FPGA Design Entry Using *Warp3*™

This application note is intended to demonstrate hierarchical as well as mixed-mode design entry for FPGAs using the *Warp3*™ software package. *Warp3* eases and speeds up the design process by featuring both schematic and VHDL design entry methods. Complex designs may be broken up into manageable pieces and each piece may be described behaviorally (VHDL) or structurally (VHDL and schematic). All the lower-level blocks are then put together to create the top level. In this application note, a general-purpose DMA controller is designed to further familiarize the reader with the *Warp3* design process.

Warp3 Interface and the Cockpit Overview

Running on both IBM PC/AT™-compatible platform and Sun SPARCstation™, *Warp3* provides all the tools necessary to quickly and efficiently convert complex designs into functional silicon. *Warp3* uses ViewLogic as its front-end. *Figure 1* shows the Powerview cockpit which appears when you invoke *Warp3* on Unix workstations. To the upper right is a collection of icons, one for each *Warp3* tool.

Viewdraw is used to create schematics, as well as symbols that can be instantiated on other schematics. It gives you the ability to capture schematics utilizing standard 74XXX TTL functions, generic logic gates, or user-defined custom functions.

VHDL designs may be entered using ViewText or any other text editor. VHDL can be used to describe the entire design or just a portion of it. It allows for state machine, Boolean equation, IF/ELSE type constructs, tabular, and many other design description styles. Packages allow designs to be integrated into higher levels of the hierarchy.

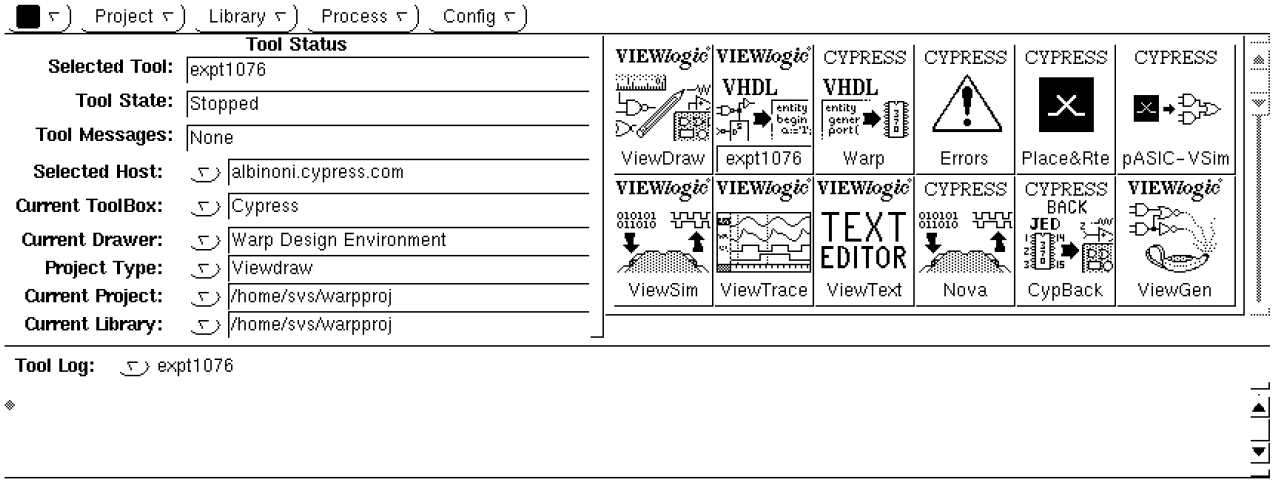

Figure 1. Powerview CockPit

ViewGen generates schematic symbols from schematic drawing. The resulting symbol could then be instantiated on other, higher-level schematics.

All designs (schematic and VHDL) are converted to VHDL, so for designs containing schematics, Expt1076 is run to translate the viewdraw schematic into one or more VHDL models.

The VHDL files are then compiled and synthesized using *Warp*™. *Warp* produces JEDEC files (used to program PLDs), HEX files (used to program PROMs), or QDIF files (used by the Place&Route tools when targeting pASIC™ FPGAs).

For FPGA devices, the Place&Rte tool is used to perform automatic place and route, delay modeling, critical-path timing analysis, automatic test vector generation, and device programming and test.

After compiling the design, ViewSim can be used to determine the design's functionality and worst case timing characteristics. ViewSim automatically brings up ViewTrace, which allows you to view the simulated waveforms.

After compilation, if the same pin assignment is desired to be kept, CypBack can be used for back annotation.

This section was intended to provide an overview of the Cockpit. For additional information please refer to the *Warp3* documentation.

Cypress pASIC380 Family FPGA Architectures

The previous architecture discussions have pointed out the strong relationship between the technology, the architecture of the FPGA, and the device characteristics. The 380 family possesses a unique technology which impacts all of the remaining architecture trade offs positively. The discussion of the 380 family begins, therefore, with a presentation of the interconnect technology.

pASIC380 Family Fuse Technology

In usual integrated circuits two crossing metal lines that are on different layers may be connected by a via. A via is a small hole in the insulating glass that lies between the two layers of metal. This small hole, which is about the size of the metal lines themselves, is filled with metal from above making the connection to the underlying metal line. The programmable via is a modified via used in standard CMOS semiconductor processing. The modification consists of depositing a thin layer of amorphous silicon in the via hole so that the silicon separates the two layers of metal. As manufactured, this special via has a resistance in excess of 1 gigaohm and an insignificantly small capacitance (about 1 fF). Its size is no larger than the standard via normally used to connect two layers of metal. A cross section of the programmable via is shown in *Figure 2*. A programming pulse applied across the programmable via causes a change in the characteristics of the silicon layer forming a bidirectional conductive link between the top and bottom metal. This programmed

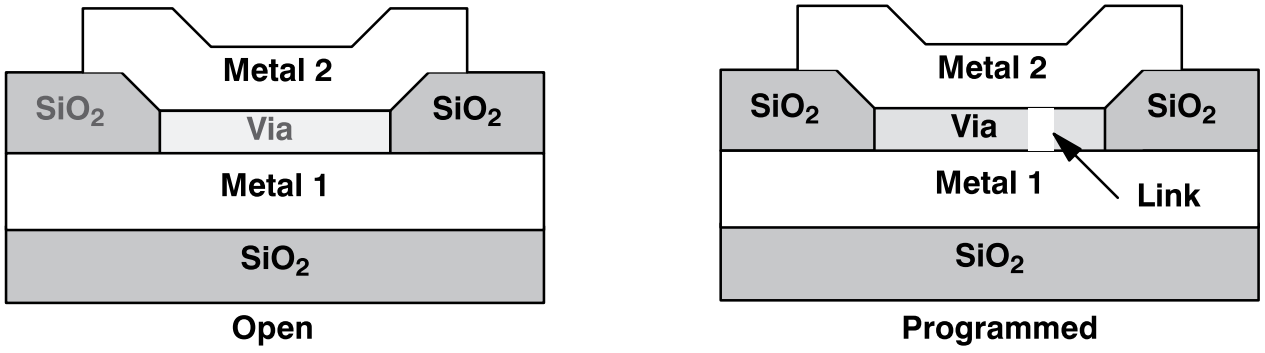

Figure 2. The ViaLink

link has a series resistance of about 52 ohms and in practice is no more than 65 ohms. The parasitic capacitance is no larger than a normal metal to metal via. The technology is appropriately termed "ViaLink™."

Routing

ViaLink technology has significant impact on FPGA architecture. Since the programmable site is no larger than the associated metal interconnect wires, there is no real restriction on the number of interconnect points (fuses) and no fuse related restrictions on the number of wires in the interconnect channels. The pASIC380 family takes advantage of this freedom with a generous routing structure.

Four types of signal wires are employed in the routing channels:

- segmented wires
- quad segmented wires
- express wires
- clock wires

Segmented wires are wires that extend only from one routing channel to the next, both vertically and horizontally. At the channel junction, a horizontal segmented wire may be programmed to interconnect to a vertical segmented wire at points called cross links. In *Figure 3*, programmable cross links are denoted by the open circle at intersections of vertical and horizontal wires. Also at the channel juncture, the segmented wire may be continued in the original horizontal or vertical direction by connection to another segmented wire running in the same channel. This connection is provided by a pass link. These links are denoted by an "x" in the figure. Segmented wires are most applicable for local wiring around or between adjacent logic cells.

Quad segmented wires are similar to the segmented wires described above except that the wire extends across four logic cells before it is segmented. Like segmented wires, the quad segmented wires may be continued to the next quad segmented wire by a pass link. The quad segmented wires are applicable to signal distribution over a larger but still local group of logic cells.

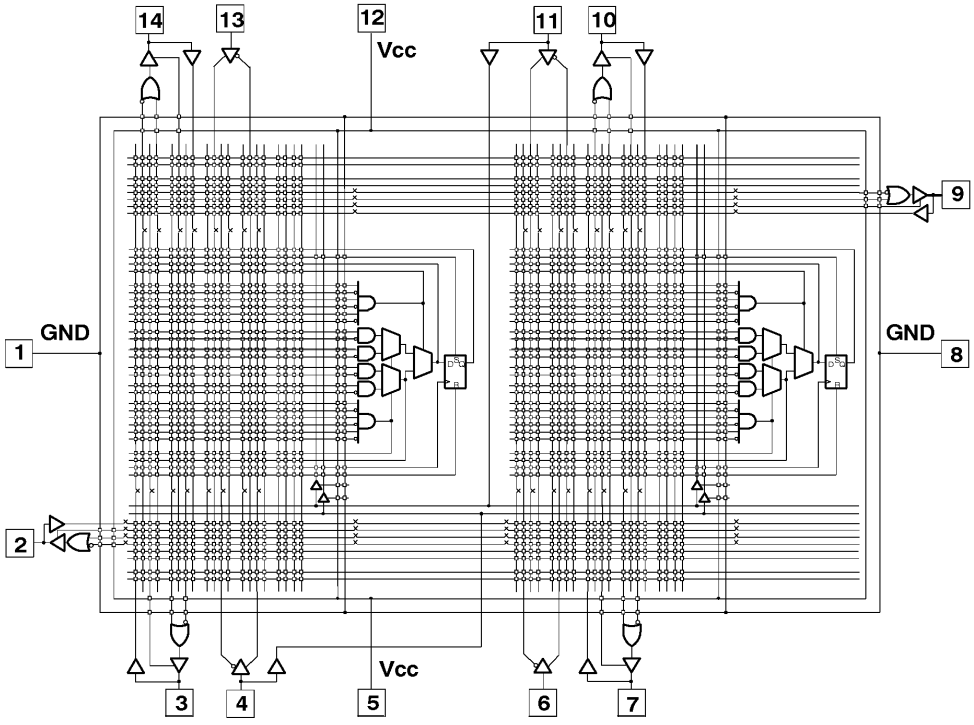

Figure 3. Simplified pASIC380 Family Model

Express wires are similar to segmented wires except they do not include pass links. An express wire will therefore run the entire length of the device. These wires are most suitable for global signals within the device.

Routing software with specific knowledge of the device architecture will automatically route signals over the appropriate wire type.

Clock wires are special signal lines that include an array of buffers for minimal skew. Clock wires are similar to express wires except that the cross links are limited. This is to insure that the clock wires are lightly loaded by programmable interconnects and can be used maximally in routing high-speed clocks or reset signals globally throughout the device with minimal skew. The source of the signal on the clock wires is specific device pins with the designation "I/CLK." After passing through the special input buffers, the signal is routed horizontally across the center of the die, as shown in *Figure 4*. There are four high drive buffers. One pair drive clock 1 and clock 2 to the upper half of the column of logic cells, and the other pair drive the two clocks to the lower half column of logic cells. There is a cluster of these buffers for each column of logic cells in the array. The buffers can be enabled to drive the clock lines or disabled if a clock is not required in a given column.

Vertical channels include all three wire types plus V_{CC} and ground wires. The V_{CC} and ground connections allow unused inputs of any logic cell to be tied to an appropriate logic level. The vertical channels run to the left of each logic cell column and extend the full height of the device. The I/O wires, which run from each of the logic cells to the right of the vertical channel, intersect the wires of the vertical channel with cross links at all segmented wires and at

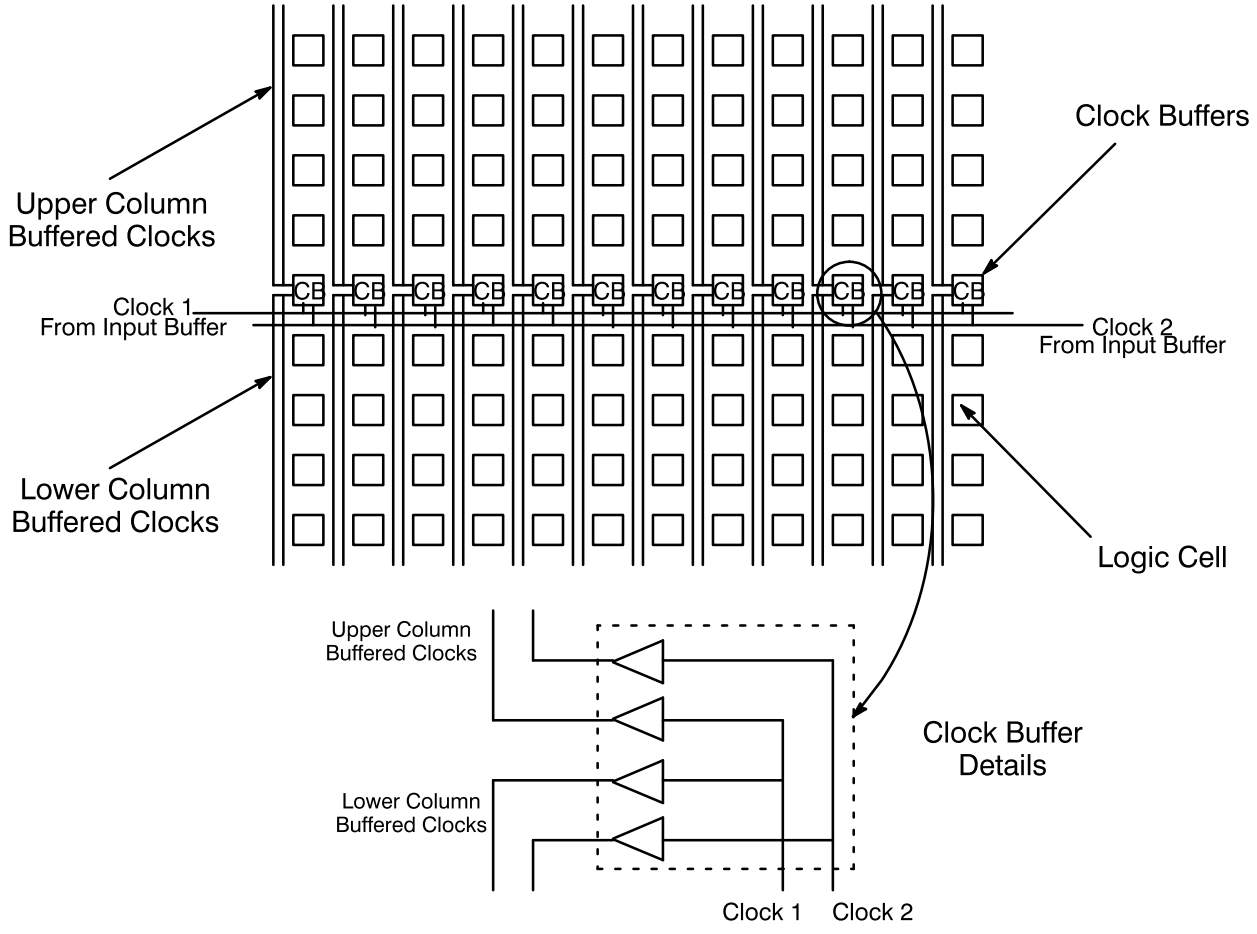

Figure 4. pASIC380 Family Clock Distribution

judicious points for express wires. At the extreme ends of the vertical channels are I/O cells that connect to the device pins. The number of wires in the vertical channel is chosen to be commensurate with the number of inputs and outputs of a logic cell, the added wires for V_{CC}, ground, and the I/O cells at the device periphery. There are 24 of these wires.

Horizontal channels provide connection by way of cross links from vertical channel to vertical channel and from the vertical channels to I/O cells on the left and right periphery of the device. All wire types are included in the horizontal channels (which contain 12 wires each) except for the clock wires. (These are the dedicated wires that carry the clocks to the buffers.)

I/O Cells

There are three types of interface buffers that connect the internal array to the device pins. The dedicated input buffer provides high drive internally and generates both true and complementary versions of the input signal. This high drive capability allows signals coming from these input only buffers to fan out to a larger number of cells than the normal I/O cell. The clock input buffer is similar to the dedicated input buffer except that it provides a third output that is routed to the internal clock distribution buffers described previously. The I/O cell provides a bidirectional connection to the devices pins. The cell can be used as input only, output only, or a bidirectional pin connection. Internally the cell has an output enable, an input data connection, and two output data connections which are ORed together to produce the output. This cell is shown schematically in *Figure 5*. The output driver provides 8 mA drive level (I_{OH} and I_{OL}).

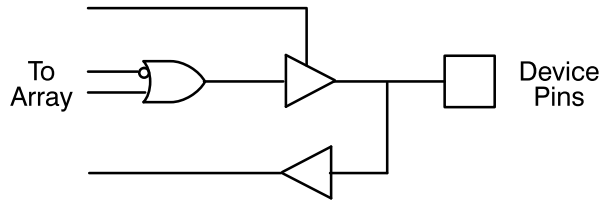

Figure 5. Bidirectional I/O Buffer

Logic Cells in the pASIC380 Family

Since the routing resources of the 380 family are abundant and without expectation of being interconnect constrained, there is freedom in the logic cell architecture to choose the optimum complexity. The 380 family logic cell is shown in *Figure 6*. This cell has been optimized to maintain the speed advantage of the ViaLink technology while insuring maximum logic flexibility.

The logic cell consists of two 6-input AND gates, four 2-input AND gates, three 2-to-1 multiplexers and a D flip-flop. This cell represents approximately 30 gate equivalents of logic capability. The cell has 23 logic and control inputs and 5 outputs. The arrangement of the gates permits 14-bit-wide gating functions and can realize all possible Boolean transfer functions of up to three variables. The D flip-flop possesses asynchronous set and reset inputs to independently control the output state. The multiplexer and logic feeding the D input allow the flip-flop to be configured as D, T, JK, or SR.

The outputs of the logic cell include the Q output of the flip-flop (QZ) plus four other outputs tapped at selected points within the logic cell. The OZ output is the same as the D input to the flip-flop. The OZ

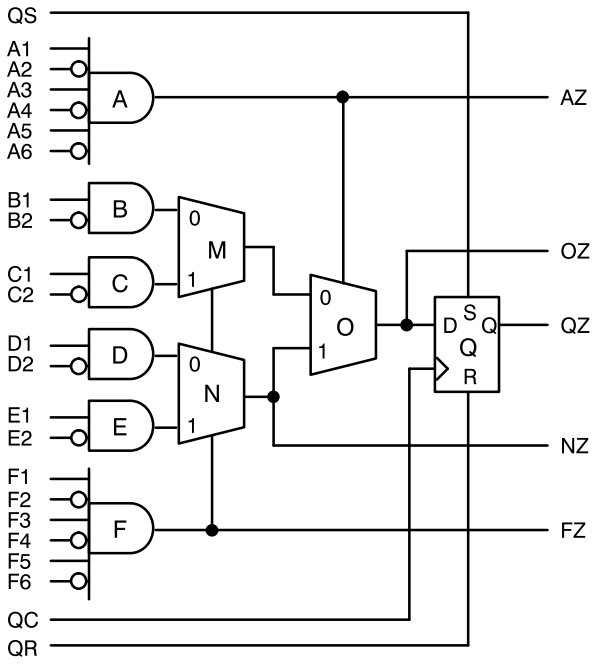

Figure 6. pASIC380 Internal Logic Cell

output facilitates combinatorial functions. The three other combinatorial outputs tap the logic cell at selected places. If simple logic functions are to be implemented, the multiple outputs permit more than one of these functions to be realized in a single logic cell. Maximum use of the available logic can be made. Note the ability to provide this multifunction utilization without any significant impact on routing. The additional utilization factor is obtained for free. When implementing multiple functions, the flip-flop may still be employed in many cases.

The logic cell is not so complex as to adversely impact propagation delay. The internal multiplexers are positioned to participate in implementing logic functions. Since the multiplexers are all in the path to the D input of the flip-flop, they contribute significantly to combinatorial logic function realization and are not expended on signal steering. The logic cell is also noticeably symmetric and regular. Combinatorial delays are thus also symmetric. That is, input to output delays tend to be roughly the same, although the AZ and FZ output will be faster than the others. Whereas some architectures bypass large sections of cell logic by the multiplexing, thereby making the cell delay dynamically changeable, the pASIC380 logic cell delay is not subject to this condition.

Design Example

The application example described here is a general-purpose, 16-bit direct memory access controller (DMAC). Direct Memory Access facilitates maximum I/O data rate and maximum concurrence. For DMA transfers, the Central Processing Unit (CPU) must have a DMA feature. Additional external logic is also necessary. This additional logic, the DMA controller, contains its own address register, word count register, and logic for reading or writing data to or from memory. *Figure 7* illustrates the basic components of a DMA controller.

The CPU loads the DMAC with a starting address for the memory transfer and the number of words to transfer. When an I/O device requires data from memory or needs to transfer data to memory, it must request service from the DMAC by asserting a DMA request (DREQ). The DMAC then activates its hold request (HLDREQ) output. The DMAC then waits until it receives a hold acknowledge (HLDA) signal from the CPU. At this time the CPU floats its address and data buses and appropriate control lines. It suspends any processing that re-

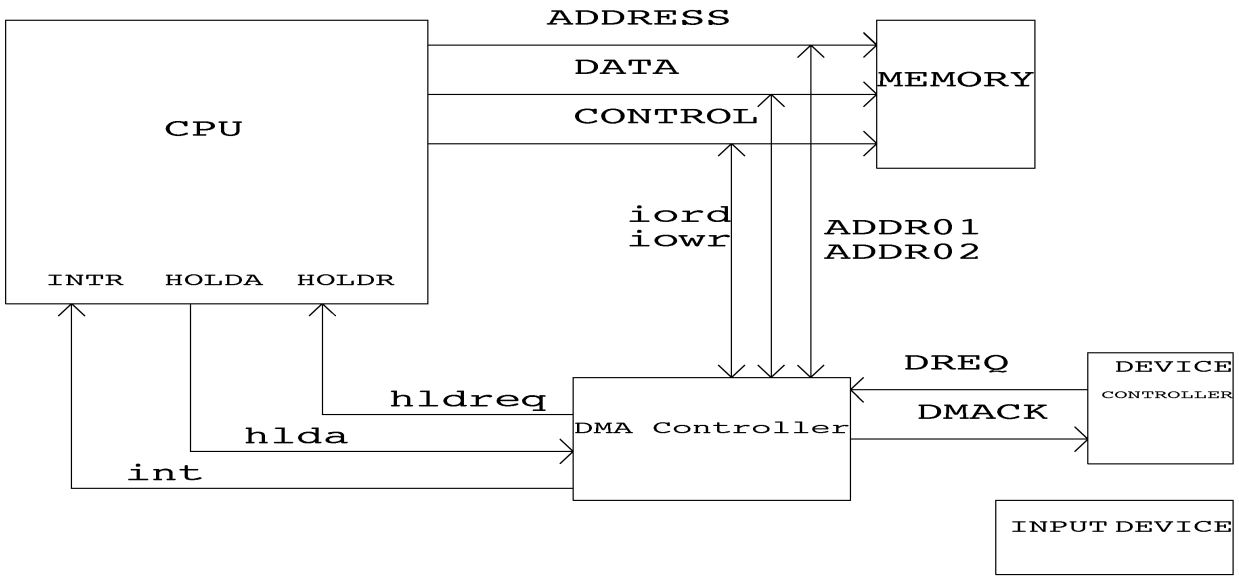

Figure 7. DMA Controller Controlling an Input Device

quires use of the address and data bus. The DMA controller then provides address and control strobes to read or write memory. The I/O device provides or accepts the data on the data bus.

Data transfers between memory and I/O devices can occur as single-word operations or as bursts of words under CPU program control. A 16-bit counter is decremented every transfer. When the required number of words have been transferred (a count of zero is reached), the DMAC terminates the DMA request and interrupts the CPU to indicate that the DMA transfer is complete.

In this implementation of the DMA controller, it is partitioned into six smaller blocks, as follows: the CPU Decoder, Control register, address generator, word counter, output multiplexer, and the DMA state machine. *Figure 8* shows the DMAC block diagram. Since some blocks are easier to describe in schematic and some others in VHDL, mixed mode design entry is selected here. For example state machine or the CPU decoder modules are easier to describe in VHDL using Behavioral and Tabular design entry methods.

The DMAC building blocks are described here in detail including an explanation of the design methodology chosen for each implementation.

CPU Decoder

The DMAC is configured by the CPU via address bits (A01 and A02), and control signals IOWRI, and IORDI. The CPU decoder receives these interface signals from the CPU and decodes them into internal write strobes and a read enable. The write strobes latch incoming parameters form the data bus into Control register, Word Counter, and Address registers. The read enable (RD_ENABL) sig-

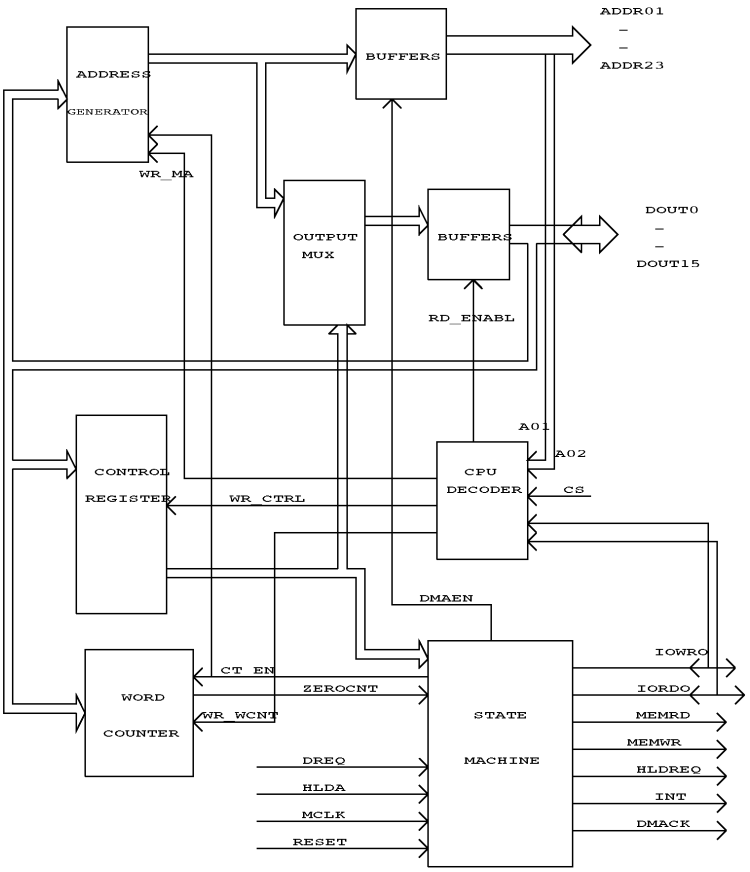

Figure 8. DMAC Block Diagram

nal and address line A01 (from the CPU) allow internally selected address registers to be multiplexed onto the data bus (during a CPU read operation). *Table 1* shows the decoding of the CPU address lines and I/O instructions by the DMAC.

Figure 9 shows the VHDL code for the CPU Decoder Block. The Entity section declares the design's inputs, outputs, and their types. VHDL provides several ways to specify a design's operation, a truth table is used to describe the CPU Decoder to express which outputs are active when specific inputs are asserted. In order to use the Tabular method of behavioral description, the "use work.table_bv.all" statement must be included. The body of architecture "arctbl" of entity "cpudec" contains the truth table. Signal TABLE_OUT is defined to hold the truth table's output signals values. Since there are 5 outputs, this signal is defined as BIT_VECTOR(0 to 4). The table is defined as constant "dectable," indicating the number of rows (0 to 4=5) and columns (0 to 8=9) it contains, followed by the bit values of the table itself.

The process "machine" then calls the TTF() function to produce outputs from the design's inputs. Since the CPUDEC.VHD (file name) is a lower-level piece of our DMA controller design, and it needs to be instantiated into our top-level DMA controller, it needs to be put in a Package. This is easily accomplished by copying and then slightly modifying the Entity section. The Package section is then placed at the top of CPUDEC.VHD file and is then recompiled. The last step is to run VHDL−> SYM (found in Viewdraw) which analyses our VHDL model and automatically generates a symbol. The symbol and the VHDL design file have the same name as the VHDL Entity name with an extension of ".1".

Control Register

The Control register configures the DMAC and controls the DMA controller's operation. The CPU writes to the control register block. The Control register has control bits to enable or disable the DMAC, enable an interrupt when the word count equals zero, clear the word counter, enable burst or single-byte transfers, and define the transfer direction (memory to I/O or I/O to memory). The bit definitions for each DMAC function appear in *Table 2*.

Warp3's schematic capture capability is used to implement the control register block. The registers can be cleared using the RESET or CLRENB signal from the state machine. The write control signal (WR_CTRL) from the CPU Decoder block clocks in the data bit values. After the design is entered in ViewDraw, Export1076 is run to convert the schematic to its VHDL model. The VHDL model is then compiled using the *Warp* compiler (Galaxy). Finally Viewgen is used to create a symbol for this lower-level design. The Control register schematic is shown in *Figure 10*.

Table 1. DMAC CPU Signals Decoding

A02	A01	CS	IORDI	IOWRI	Description
X	X	0	X	X	
0	0	1	0	1	Write Control Register
0	1	1	0	1	Write Word Count
1	0	1	0	1	Write Low Mem Address
X	0	1	1	0	Read Low Mem Address
1	1	1	0	1	Write High Mem Address
X	1	1	1	0	Read High Mem Address and DMAC Status

```
package cpu_dec is

component cpudec
  port(iowri,iordi,cs,a01,a02:in bit;
       wr_ctrl,wr_wcnt,wr_ma_0,wr_ma_1,rd_enabl:out bit);
end component;

end cpu_dec;

entity cpudec is
  port(iowri,iordi,cs,a01,a02:in bit;
      wr_ctrl,wr_wcnt,wr_ma_0,wr_ma_1,rd_enabl: out bit);

end cpudec;

use work.table_bv.all;

architecture arctbl of cpudec is

  signal table_out :bit_vector(0 to 4);
  constant dectable:x01_table(0 to 4, 0 to 8):= (
  -- inputs    outputs
  --
     "xx10" & "00001",  -- read status reg or i/o
     "0001" & "10000",  -- write control register
     "0101" & "01000",  -- write word count
     "1001" & "00100",  -- write low mem address
     "1101" & "00010"); -- write high mem address

begin

 machine: process (cs)
  begin
  if cs = '1' then
    table_out <= ttf(dectable,a02&a01& iordi& iowri);
  end if;
  end process;

wr_ctrl <= table_out(0);
wr_wcnt <= table_out(1);
wr_ma_0 <= table_out(2);
wr_ma_1 <= table_out(3);
rd_enabl <= table_out(4);

end arctbl;
```

Figure 9. CPU Decoder VHDL Design File

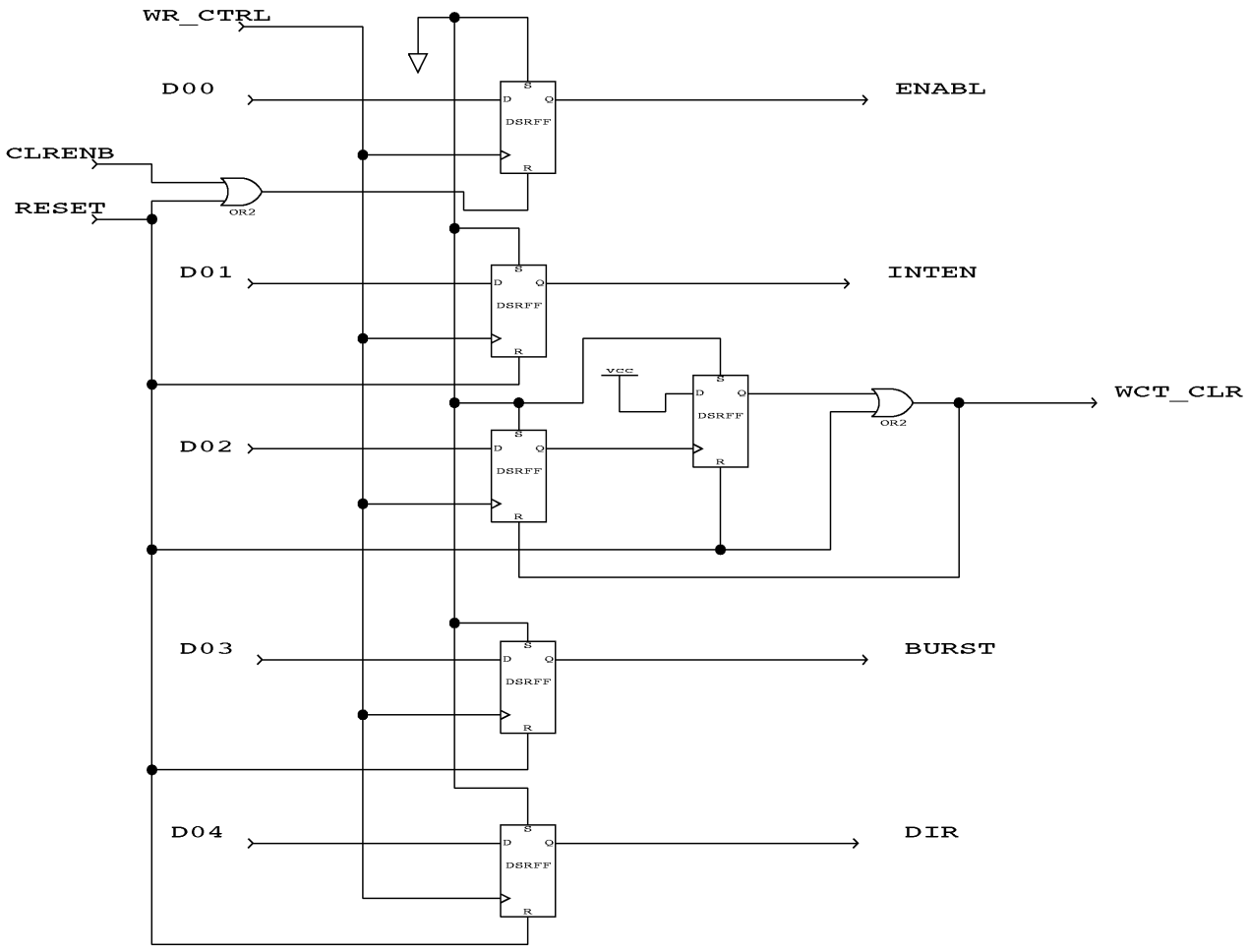

Figure 10. Control Register Schematic

Table 2. DMAC Control Register Bit Definitions

BIT	DEFINITION
0	DMA Controller Enabled (ENABL)
1	Interrupt Enabled (INTEN)
2	Clear Word Counter (1 clears Word Counter and bit 2 to zero)
3	Burst/Single Word Transfer Mode (0=single, 1=Burst)
4	Transfer Direction (0=Mem to I/O, 1=I/O to Mem)
5−15	Not Used

Address Generator

The Address Generator block is a 23-bit synchronous counter that provides the system memory address for the data transfer operation. The 23 address registers are initialized by loading the registers with the address of the first memory location to be accessed. The CPU places the 23-bit starting address on the 16-bit data bus in two operations, one for the lower 16 bits and one for the upper 7 bits. This is controlled by WR_MA_0 and WR_MA_1 signals. After each memory transaction, the state machine block asserts the CT_EN signal which enables the counter to increment. This guarantees that the address is set for the following transfer.

Figure 11 shows the Address Generator diagram. Using the 74XXX TTL functions available in *Warp3*, the address generation function is implemented with six 4-bit, 74161 counters. These counters are arranged so that when each 4-bit counter increments to a binary count of 1111, its ripple carry out output (RCO) enables the next higher 4-bit counter via the ENT and ENP inputs (tied together). The 23 address lines must be three-stated when the CPU has ownership of the system bus. The state machine block generates an output signal called DMAEN which at the appropriate time (during data transfer) enables the DMAC address lines. The three-state buffers must be implemented in the top-level design and they correspond to the internal three-state buffers of the pASIC devices.

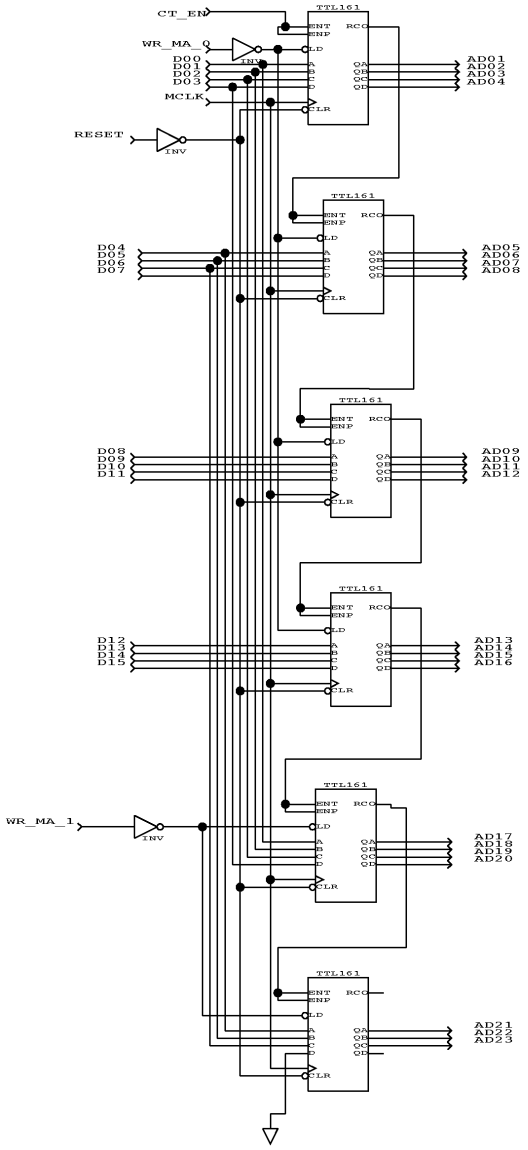

Figure 11. Address Generator Schematic

A symbol is generated for this block as was done for the Control register block.

Word Counter

Because each transfer operation requires a word count, a 16-bit counter monitors the number of words that are transferred. The CPU initializes the Word Counter to a value representing one less than the number of words to be transferred. This value allows the counter to reach zero before the last transfer and terminate the operation at the proper time. Four 74161 counters are used to construct this counter as shown in *Figure 12*. The WR_WCNT signal from the CPUDEC block and the data bits D00 through D15 initialize this counter. The data bits are inverted as they are loaded. Therefore this counter is actually decremented instead of incremented. At the end of each transfer (states *mem2* and *io2*) the CT_EN signal is asserted HIGH. This signal enables both the Address register and the Word Counter blocks. Each time the Address register is incremented, the Word Counter is decremented. The Word Counter is cleared using the RESET signal from the CPU or the WCT_CLR signal from the Control register block written by CPU address and control lines as shown in *Table 1*.

Output Multiplexer

The CPU must have access to the DMAC's internal registers to monitor operation. Therefore, the CPU has the capability of reading the DMAC's current status and configuration. This is signaled to the DMAC by asserting the A01 address line and the IORDI signal HIGH (see *Table 1*). When these two signals along with the CS signal go HIGH, the CPU decoder asserts the RD_ENABL signal HIGH. The required data is then driven to the CPU data bus. The bit definitions for the control signals are essentially the same as those for the control word (on different data bits) and are shown in *Table 3*. A multiplexing scheme is used here to enable the CPU to read either the address generator's lower 16 bits

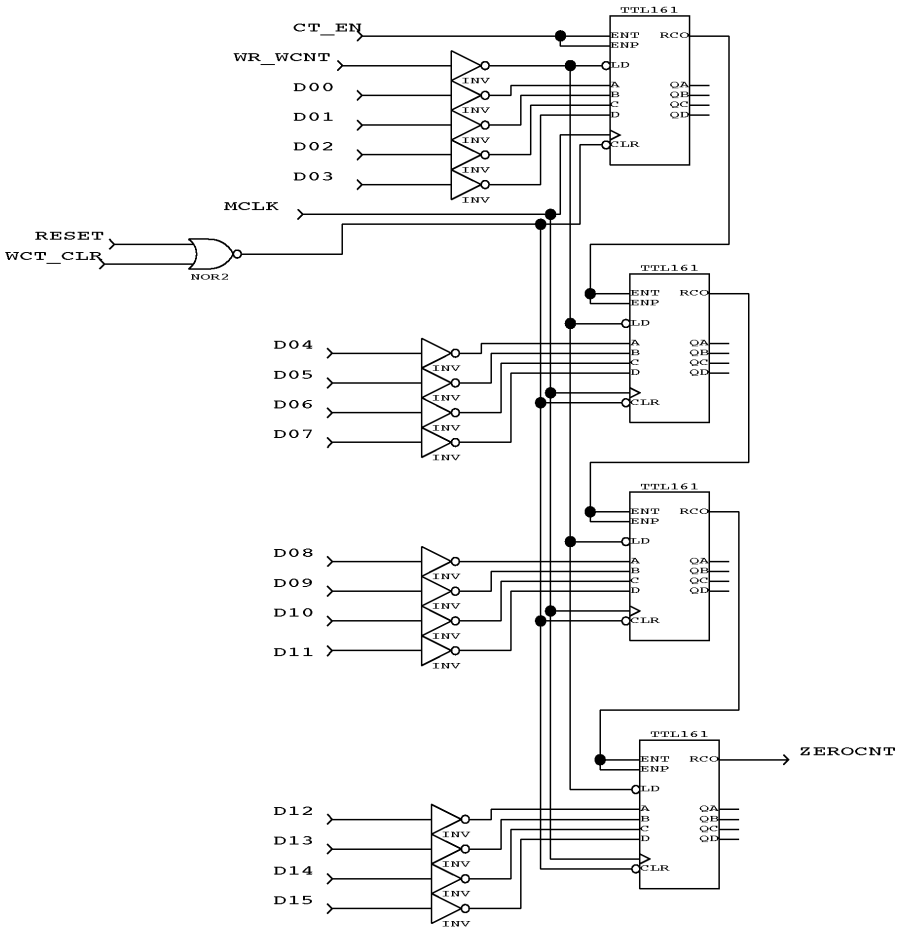

Figure 12. Word Counter Implemented in *Warp3* Schematic Capture Tool

(DATA00−DATA15 when A01=0) or upper 7 bits (DATA00−DATA06) and the DMAC status information (DATA08−DATA09, DATA11−DATA12 when A01=1). *Figure 13* shows the Output Multiplexer.

Table 3. DMAC Status Register Definition

BIT	DEFINITION
8	DMA Controller Enabled (ENABL)
9	Interrupt Enabled (INTEN)
10	Not Used
11	Burst/Single-Word Transfer Mode (0=Single, 1=Burst)
12	Transfer Direction (0=Mem to I/O, 1=I/O to Mem)
13−15	Not Used

DMA Control State Machine

Figure 14 shows the state diagram for the DMA controller. The state machine consists of 9 states: IDLE, HOLD, DIRCT, MEM, IO, ENDST, ENDHLD, CLENB, INTRPT. In IDLE state, the controller waits for an ENABL signal from the Control Register Module. Upon receiving this signal, it goes to the HOLD state and waits for the HLDA signal from the CPU. In DIRCT state, the DMAEN signal is asserted, which enables the three-state buffers that control the address lines. This signal stays asserted through state ENDST. Depending on the Control register content (written by the CPU), data is transferred between the memory and the I/O device. In states IO2 and MEM2, CT_EN is asserted, which in turn increments the Address registers and decrements the Count registers after each transfer. In state ENDST, if all words have been

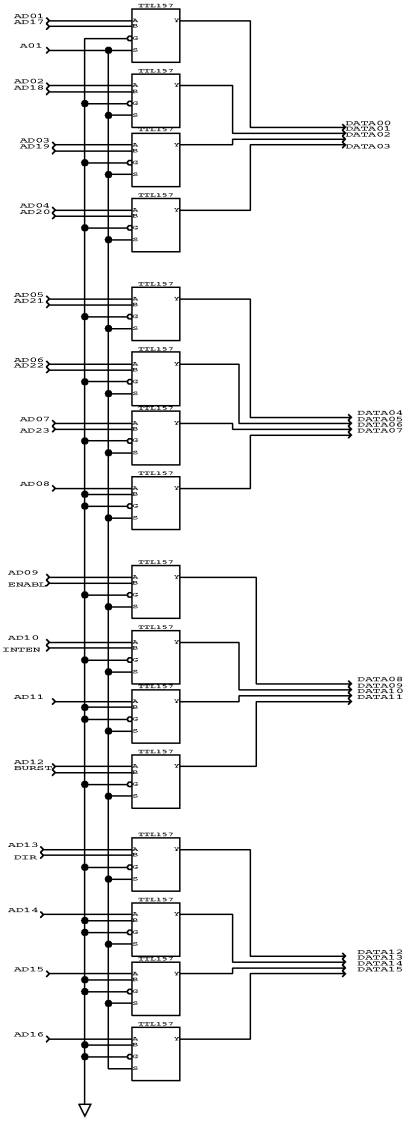

Figure 13. The Output Multiplexer

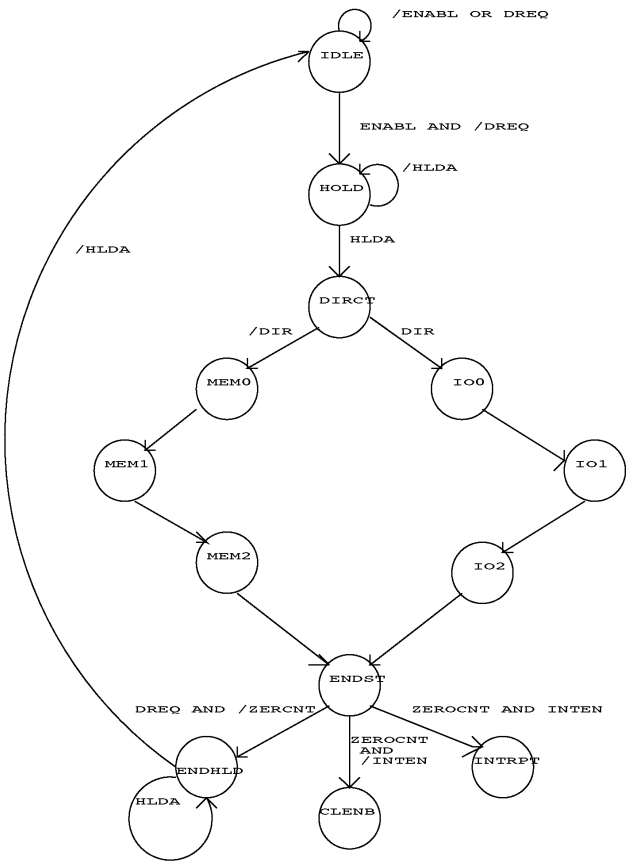

Figure 14. DMA Control State Machine

transferred and there is no Interrupt enable signal from the Control register, then the Control register is cleared.

Figure 15 shows the behavioral description of the DMAC state machine implemented in VHDL. This is a Moore state machine, since the outputs are only a function of the states.

In the Architecture section, we have declared a signal which is a vector that is 11 bits wide. It is called STATE. In this state machine, all the outputs are encoded within state bits. Since there are 10 outputs, we need at least 10 state variables. The 11th bit is used to make all state definitions unique. The operation of the state machine is described in the Process section. Notice that behavioral description uses a combination of CASE-WHEN and IF-THEN-ELSE statements. The state machine can asynchronously go to state IDLE. All of the inputs and outputs are defined as BITS in the Entity section. *Warp* assumes that for BIT types '1' is true and '0' is false. After the Process section, all the outputs are assigned to state bits.

Top-LEVEL DMAC Design

After creating lower-level block, each design was compiled and a symbol was created. It's time now to incorporate all the symbols in the DMAC's top-level schematic (*Figure 16*). To accomplish this, each symbol is called and placed on the schematic. To

```
package dma_ctrl is
component  dmas
  port(reset,dreq,hlda,zerocnt,enabl,inten,dir,burst,mclk:in bit;
       ct_en,memw,memr,iowr,iord,dack,dmaen,hreq,setint,clrenb:out bit);

end component;
end dma_ctrl;

entity dmas is
  port(reset,dreq,hlda,zerocnt,enabl,inten,dir,burst,mclk:in bit;
       ct_en,memw,memr,iowr,iord,dack,dmaen,hreq,setint,clrenb:out bit);

end dmas;

architecture machin of dmas is

signal state:bit_vector(10 downto 0);

constant idle  :bit_vector(10 downto 0)  := "00000000000";
constant hold  :bit_vector(10 downto 0)  := "00000001000";
constant dirct :bit_vector(10 downto 0)  := "00000011000";
constant mem0  :bit_vector(10 downto 0)  := "00100111000";
constant mem1  :bit_vector(10 downto 0)  := "00110111000";
constant mem2  :bit_vector(10 downto 0)  := "00000111001";
constant io0   :bit_vector(10 downto 0)  := "00001111000";
constant io1   :bit_vector(10 downto 0)  := "01001111000";
constant io2   :bit_vector(10 downto 0)  := "10000111001";
constant endst   :bit_vector(10 downto 0)  := "10000011000";
constant intrpt  :bit_vector(10 downto 0)  := "00000001100";
constant endhld  :bit_vector(10 downto 0)  := "10000000000";
constant clenb :bit_vector(10 downto 0)  := "00000000010";

begin

dma: process (mclk,reset)

  begin

    if reset = '1' then
      state <= idle;
    elsif (mclk'event and mclk = '1') then
```

Figure 15. VHDL Code for DMAC State Machine

```
case state is
  when idle =>
     if (enabl='1' and dreq ='0')  then
    state <= hold;
     end if;

  when hold =>
     if hlda ='1'  then
    state <= dirct;
     end if;

  when dirct =>
     if dir= '1'  then
    state <= io0;
     else
    state <= mem0;
     end if;

  when mem0 =>
    state <= mem1;

  when mem1 =>
    state <= mem2;

  when mem2 =>
    state <= endst;

  when io0 =>
    state <= io1;

  when io1 =>
    state <= io2;

  when io2 =>
    state <= endst;

  when endst =>
     if (dreq='0' and zerocnt='0' and burst='1') then
    state <= dirct;
     elsif (dreq='1'  and zerocnt='0') then
    state <= hold;
     elsif (zerocnt='1' and inten='1') then
    state <= intrpt;
     elsif (zerocnt='1' and inten='0')  then
    state <= clenb;
     end if;
```

Figure 15. VHDL Code for DMAC State Machine (continued)

```
          when intrpt =>
            state <= clenb;

          when clenb =>
            state <= endhld;

          when endhld =>
             if (hlda='0')  then
            state <= idle;
             end if;

          when others =>
            state <= idle;

       end case;
     end if;
end process;

-- assign state outputs to state bits

ct_en <= state(0);
clrenb <= state(1);
setint <= state(2);
hreq <= state(3);
dmaen <= state(4);
dack <= state(5);
iord <= state(6);
iowr <= state(7);
memr <= state(8);
memw <= state(9);

-- bit 10 is to make all state definitions unique.
end machin;
```

Figure 15. VHDL Code for DMAC State Machine (continued)

connect signals, it is sufficient to give them the same names rather than connecting them by wires. The external inputs and outputs are connected to input and output ports. Since the RESET signal is a high fanout signal, an HDPAD is used for distributing this signal across the device. Using an HDPAD insures usage of a dedicated input pin for the signal, giving it twice the current drive capability of the I/O pads. In addition a CKPAD is used for the clock input (MCLK). This uses a clock pin for this signal. The Clock/input pin drives a low-skew, fan-out independent clock tree that can connect to clock, set, or reset inputs of the logic-cell flip-flops. Next *triout* and *bufoe* components are used to implement three-state buffers. The *triout* component has three ports: DATA_IN, ENABLE, and DATA_OUT. The *bufoe* component has four ports: DATA_IN, ENABLE, DATA_OUT, and FEEDBACK. These two types of buffers must be connected to bidirectional pins. In this design, when the CPU has ownership of the system bus, the DMAC's address, memory and I/O control lines are in a high-impedance state. The data bus must also remain in a high-impedance state unless the CPU is reading the DMAC's internal reg-

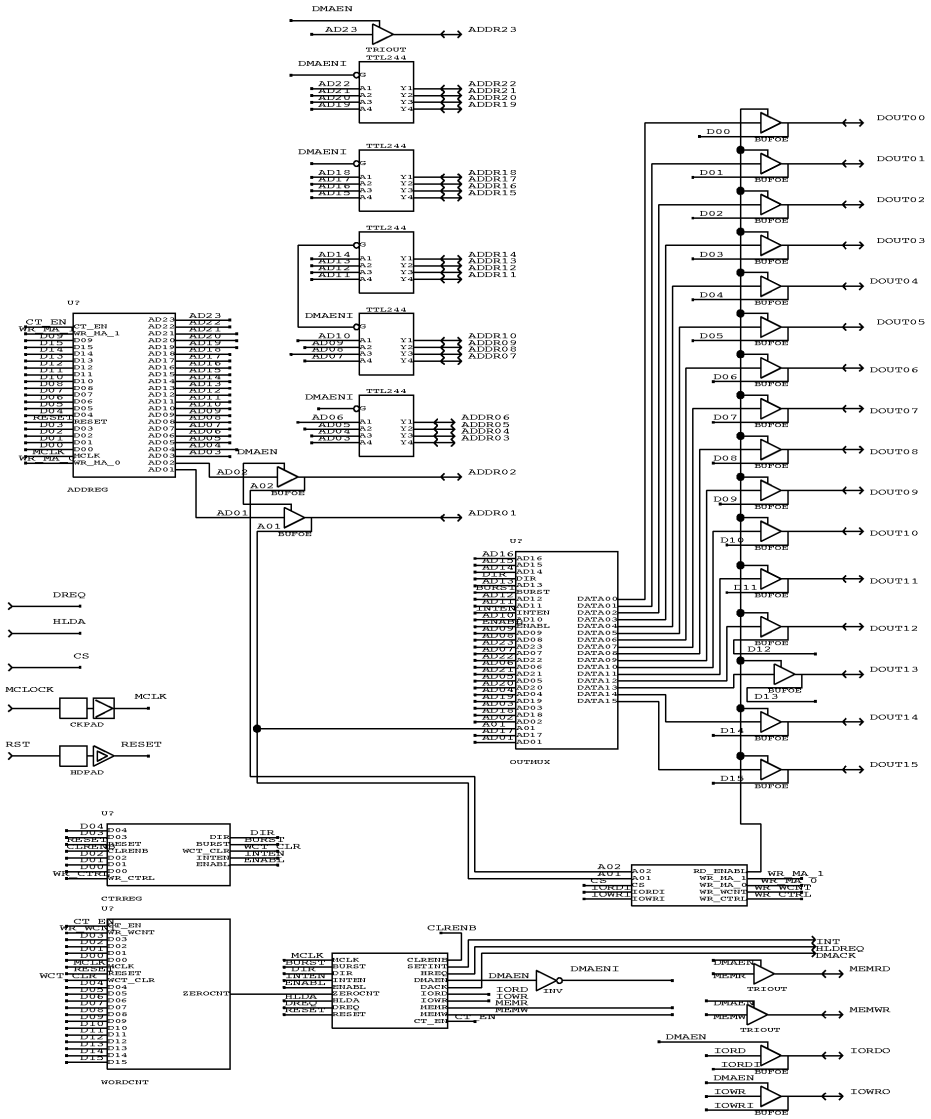

Figure 16. DMAC Top-Level Schematic

isters. The state machine's output, DMAEN, enables the address bus, IORDO, IOWRO, MEMRDO, and MEMWRO outputs. The data outputs are enabled by a RD_ENABL signal from the CPU decoder module. Since signals A01, A02, IORDO, IOWRO, and DATA bus (DOUT00−DOUT15) may be driven by the CPU to initialize the DMAC, *bufoes* (rather than *triout*) are used to connect these signals to bidirectional pins.

A VHDL model for the top-level schematic is then created using EXPT1076 from the *Warp3* Cockpit. The final task remaining is to compile the overall DMAC design and automatically place and route it into a pASIC device. This design easily fits into a CY7C383A. It uses 81 percent of the Logic Cells and 79 percent of the Pad cells.