Designing with FPGAs
An Introduction to Cypress's pASIC380 Family of FPGAs and the *Warp3*™ Design Tool

Introduction

Field Programmable Gate Arrays (FPGA) borrow the sea of gates concept from the gate array semicustom integrated circuit and add field programmability. The similarity of the FPGA to the semicustom approaches opens many possibilities for the design engineer. With a large number of gates available, complex designs can be implemented into a single device. In the semicustom approach it may be many weeks from sign off to prototypes. Moreover, simulation is usually exhaustive (due to the cost of a design change), taking many weeks of the design cycle. With field programmability, a design may be realized in a device in a week or two of design time, in contrast to the many months with a semicustom approach. The FPGA brings gate-array-like possibilities to many design projects. The lack of a non-recurring engineering charge for FPGAs makes this technology financially available to a large number of developments.

This application note is intended to be an introduction to using FPGAs by taking the reader through a complete design. The first part of this note presents the design tools for FPGA designs. Here a design flow is followed from design entry in its multiple forms. The design flow is top down. That is, the process starts from a description of high-level abstract entry of the design and progresses to adding more hardware-specific details as required for realization in a device. To illustrate the back-end part of the design process, a DRAM memory controller is presented. In this part, details of speed optimization, simulation, and device specifics required in the design description are discussed.

Design Development and Cypress's *Warp3*™ Design Tool

FPGA devices are resource-rich entities capable of implementing designs that use from 1K to 20K or more gates. These designs will not be simple, consequently the development of the design, its debugging, and final performance analysis will be a complicated task. Fortunately, gate array design and analysis tools can be used to make these tasks comparatively easy. The design proceeds beginning with the entry of the design description. This may be in schematic form or in a high level language form, such as VHDL. After entry, the VHDL code (or VHDL equivalent of the schematic) can be simulated directly to verify the functionality of the design description. The design is then synthesized and committed to a particular device. At this point the performance of the implementation can be analyzed and optimized if necessary to achieve a design target. Lastly, the actual device is programmed.

Warp3 is a modern, self-contained CPLD and FPGA design tool that supports all phases of the design process. At the front end, it includes VHDL and schematic entry design capture tools for efficient and convenient user entry. The synthesis tools include native compilation of VHDL (for accurate VHDL interpretation), and Cypress device-specific compilers for maximum utilization of device architectural features. The schematic capture, simulation, and the framework are ViewLogic™ tools

adapted specifically for PLD, CPLD, and FPGA designs.

VHDL Design

VHDL is a rich and powerful language for the description of logic circuits. The language offers the capability to use different styles of design entry. There are three styles of logic description that can be used in any combination: behavioral, dataflow, and structural. Behavioral descriptions are C- or Pascal-like constructs that specify the action of the logic in high level abstract terms. Language constructs such as the IF/ELSE statement or the CASE statement specify the behavior. Dataflow descriptions include Boolean equations that can be used to describe logic circuits. Tabular descriptions are also possible. These can be considered a subset of the behavioral descriptions but where the action of the logic is specified by a truth table. The structural description is much less abstract and can be considered a verbal description of a schematic. In one version of this form of VHDL, gates, flip-flops, and other primitives are instantiated, and their interconnection described through signals that tie the output of one primitive to the input of others. Conversely, several entities can be instantiated in a structural description of their interconnect, but the description of any of the individual entities can be behavioral.

VHDL is a hierarchical language. Just as most programming languages support subroutines, functions, and procedures, VHDL supports components, packages, and the ability to combine a set of entities into a higher-level entity. A complex VHDL design can be built by successively combining building blocks in related layers. This allows two, very powerful design approaches. First, a complex design can be done from a top-down approach. In this method, the whole of the design can be described in very abstract, high level terms. Then the design is decomposed—breaking the design apart into specific functional blocks that are described in more specific terms. This moves the design from concept to implementation, from abstract description to near hardware level realization and optimization. At the top level, the designer need not be concerned with the exact details of the design. The concern at this top level is the conformance of the design to the given functional specification. Once this is achieved, the design can be decomposed for realization purposes while being assured that the overall functional requirements are still met. Even at the top level, the design can be built up of manageable entities which can be debugged separately. Second, the design can be done from a bottom up approach. After a block diagram is sketched out, the individual blocks can be designed according to their function and the interfaces to the other blocks. The individual block designs can be done at the most detailed level. This can consist of schematics using components instantiated from a library or the design can be a structural VHDL description. With each block fully designed, they are then connected together to build the complete design.

As an example of VHDL, consider a 12-bit wide 4-to-1 multiplexer. A version of this multiplexer is used in the design example section of this application note. The first input is a 12-bit bus that is the column address for the DRAM. The second input is a select signal which controls whether the row (row_ad) or column (col_ad) is selected. The third input is a 12-bit bus that is the row address for the DRAM. The forth input is a 12-bit bus that is the refresh address for the DRAM, and the last input is the state of the controller finite state machine. The output of the multiplexer is a 12-bit address to be sent to the DRAM memory devices. When the finite state machine is in states *refad*, *wr1*, or *wr2*, the refresh address is placed on the multiplexer output. This selection is independent of the state of *col_sel*. When the finite state machine is not in states *refad*, *wr1*, or *wr2*, COL_SEL controls the multiplexing. When COL_SEL is 0, row_ad is placed on the output of the multiplexer, and when COL_SEL is 1, col_ad is placed on the output of the multiplexer. The VHDL code to implement this function is given in *Figure 1*. The code is compact and simple. One of the twelve synthesized logic equations is given in *Figure 2*. These logic equations are available in the report file generated during compilation. The equivalence of the behavioral code and the logic equation should be self evident: in the logic equation, the state machine state vector bits (for the states in the if statement) are ANDed with the re-

```
mux: process(col_ad,col_sel,row_ad,re_ad,state)
  begin
    if(state = refad or state or state = wr1 or state = wr2) then
      rc_ad <= re_ad;
    elsif(col_sel = '1') then
      rc_ad <= col_ad;
    else
      rc_ad <= row_ad;
    end if;
  end process;
```

Figure 1. Behavioral Description of Multiplexer

```
rc_ad_11_ = /controller_state_12_.Q * /controller_state_11_.Q *
            /controller_state_10_.Q * /controller_state_11_.Q *
            stored_ad_11_DFF.Q * col_sel.Q
           +
            /controller_state_12_.Q * /controller_state_11_.Q *
            /controller_state_10_.Q * /controller_state_11_.Q *
            stored_ad_23_DFF.Q * /col_sel.Q
           +
            controller_state_12_.Q * ref_ad_11_DFF.Q *
           +
            controller_state_11_.Q * ref_ad_11_DFF.Q *
           +
            controller_state_10_.Q * ref_ad_11_DFF.Q *
           +
            controller_state_9_.Q * ref_ad_11_DFF.Q *
```

Figure 2. Logic Equations for the Multiplexer

fresh address bit (REF_AD(11)) and the compliment of these bits are a factor in the remaining product terms. One of the other two product terms ANDs COL_SEL with col_ad(11); the other product term ANDs COL_SEL with row_ad(11). The logic equations refer to stored_ad_11 instead of col_ad(11) and stored_ad_23 instead of row_ad(11). This is because col_ad and row_ad are aliases for these signals. Refer to the appendix for the alias definition.

Schematic Entry

In some cases VHDL may not be the preferred method of capturing the design. A discrete implementation of the design may already exist with the objective of reducing size and improving performance by putting the circuit into an FPGA. Many designers may feel more comfortable with schematic design capture than with a high-level language description.

Mixed Mode

Some functions are difficult to describe directly in schematic form. A state machine, for example, is far easier to describe in terms of a transition table (possible in VHDL) or VHDL conditional constructs (for example, a CASE statement). Not only is the description easier but design changes and debug-

ging are also far easier in nonschematic form. It is therefore important for a tool to be capable of mixed mode design description. In its most probable form, it is desirable to place and connect a component into a schematic where the function of the component is described in VHDL.

Whether the design is done in VHDL, schematic, or mixed mode, *Warp3* transforms the user's captured form of the design into VHDL. From this VHDL description, the design synthesis and compilation takes place. This is important since a schematically captured design is collapsed in the synthesis process. Several layers of elementary gates are combined where possible into a single AND/OR plane, thus removing redundant gates. The final implementation may therefore look quite different than the original schematic.

Source Level Design Verification

It is very convenient to verify the functionality of the design at the VHDL source code level. Clearly, debugging at this stage saves considerable time and effort in that the design does not have to be synthesized and fitted to a device before simulation can take place. *Warp3* features a VHDL source level debugger that will simulate VHDL code and produce functional results. The results can be as graphical waveform displays, active line indication in the source code, and tabular displays of variable values. Various other debug facilities are included. The VHDL code can be conveniently debugged at this level leaving the post compilation simulation to speed optimization.

Synthesis, Optimization, and Place and Route

After the design is captured, the software produces a hardware realization of the design description. This process involves three steps: synthesis, optimization, and place and route, all of which are relatively transparent to the user. The user may interface with these processes to apply synthesis directives (constraints), or timing driven constraints for place and route.

If the design was captured with schematics, then the schematics are translated to a structural VHDL netlist. The netlist is flattened (i.e., hierarchy and intermediate nodes are removed). If the design was entered in behavioral VHDL, then it is converted into a flattened register-transfer-level netlist, which describes the interconnection of components. Behavioral constructs are translated to gates. Operator inferencing is used to instantiate arithmetic components.

Up to this point the internal design description is still device independent. Optimization is based on the target device. Different algorithms are used for different device families to produce an optimized netlist for use with the place and route software. The place and route software may perform some additional optimization, if necessary, to pack the gates into logic cells. The software then places logic cells in locations that will minimize total routing delays. After placement, routing software chooses the best path among many comparable solutions to route signals between I/O and logic cells, logic cells and logic cells, and logic cells and I/O.

Directive Driven Synthesis and Place and Route

In some cases, the designer will want to supply additional information to the synthesis and place and route processes to effect specific performance or resource utilization results. Synthesis directives can be used to provide buffering of high fanout signals, or to specify an area-optimized or performance-optimized implementation of a module (be it a counter, adder, or other arithmetic circuit). These optimization directives can help to eliminate any unnecessary delays due to either routing or the levels of logic required to implement a function in the critical path. Synthesis directives may also be used to dictate pad assignment so that high fanout signals will utilize high drive pads or clock pads as necessary. State encoding can also be affected by using a synthesis directive.

Another optimization technique often used in high-speed ASIC designs is pipelining. Pipelining allows complex functions to be performed over multiple clock cycles while operating at high speeds. Pipelining is not an option that is automatically performed by synthesis software, but is an option to the designer when capturing a design.

Place and route constraints can be used to affect place and route results. A path analysis tool within the place and route tool enables the designer to examine set-up and clock-to-output timing as well as maximum operating frequency. Constraints can be placed on specific paths in order to effect a more optimal placement of a given signal (for example, to improve the clock-to-output delay of a given signal).

Refer to the *Warp*™ documentation for a complete description of the available synthesis and place and route options.

Automatic Test Vector Generation

Some programmers are equipped to exercise a programmed device with a user supplied functional test program. Such programmers have enough hardware to permit driving and sensing all pins of a device. *Warp3* can generate test vector files for these programmers.

A Design Example

A design example is presented in order to illustrate the significant features of the pASIC380 family of FPGAs and the *Warp3* design tools. The design is a DRAM memory controller that interfaces to a system address and control bus on one side and a DRAM memory array on the other. The controller includes a slave bus interface, a DRAM address generator with burst transfer capability, and a state machine to effect the DRAM control signal timing, refresh, and bus handshake. This example was chosen because of its wide variety of implementations: a state machine, counters, registers, signal multiplexing, and decoding. This example is not meant to be a full featured DRAM controller. Certain obvious functionality is left out so as not to obscure the illustration objectives of the example. Specifically,

- There are variable splits in the row/column address multiplexing depending upon the DRAM type populating a given address range.
- The RAS/CAS that is asserted depends upon the address range occupied by a given DRAM type that happens to be present
- The column address does not use bus addresses [3:0] for a 64-bit data bus because these are byte addresses
- The actual refresh counter is left out of the synthesized design

The design is done as a high-level behavioral design with no hierarchy. The design is divided into four sections: the address and control register, the state machine, the refresh counters, and the address multiplexer. This is a natural division producing design tasks that are easy to grasp. Each section is coded in VHDL and simulated separately to verify the functionality of each section. Simulation at these simpler levels is easier to accomplish and can be more thorough than simulation of the entire design. Simulations at this level attempt to verify all details of the operation of the particular section. After each section is functionally verified, they are combined into the top-level controller description, which is then functionally simulated again except with a different focus. After the design verification, it is then compiled and performance analyzed. The following paragraphs will describe the design process, present representative samples of the outputs at each step, and highlight significant results of tool and device architecture capabilities.

Detailed Design

The entire controller is described as a single entity with a behavioral level architecture description. Each section of the design is coded as a separate process. These sections are described below. The entity declaration includes hardware-specific attribute statements. These specify the target device and particular pin assignments to take advantage of device-specific features for clock distribution and high fanout signal distribution. Other non-behavioral declarations are made later in the design for optimization purposes. In all other respects, the design is purely abstract.

The address register interfaces to the main system bus and captures the address and control information for the memory transaction. The register is synchronous. The system bus clock is applied to the register, which captures bus data on the rising edge of the system bus clock while the address strobe is

asserted. The output of the register supplies the address and control information to the rest of the circuit. The VHDL description of the register is given in *Figure 3*. The register circuitry includes handshake flags to communicate with other parts of the controller. When an address strobe is detected, a flag is set to inform the state machine that a bus transaction has started. The handshake is closed by the state machine clearing the flag, indicating that it has been recognized. The flag is necessary since, as will be seen in the later parts of the design, the state machine may not always be able to respond to (recognize) this one clock cycle event. The address register circuitry also includes comparator circuitry to see if the address is valid for the memory array. This comparator produces the match result combinatorially from the register output. There are other architectural options: making the address register a transparent latch and/or placing the address comparator inputs on the input side of the latch. For simplicity, these options are not pursued.

A refresh counter is required to provide a refresh address to the DRAM memory as well as to make request of the controlling state machine to periodically execute a refresh operation. The refresh counter does not add any instructional value to the description here. It is therefore not included in the example details. The refresh circuitry provides a refresh request to the state machine and a refresh address to the address multiplexer. These are set to zero in the example VHDL source code.

```
adreg: process
   begin
     wait until clk = '1';
     if(reset = '1') then
       as_flag <= '0';
     elsif(as = '0') then
       as_flag <= '1';
     elsif(clr_as = '0') then
       as_flag <= '0';
     endif;

     if(as = '0') then
       stored_ad <= address;
       burst_stored <= burst;
     end if;
   end process;
```

Figure 3. VHDL Behavioral Description of Address Register

The column address counter produces addresses for a burst transaction. In these transactions the system provides the first address of the burst and the remaining addresses are expected to be generated by the slave. The column address counter is loaded, under control of the state machine, with the low-order portion of the incoming address and is advanced also under control of the state machine. The counter also serves to hold the column address in those cases where a new address is received and the state machine is not yet finished with the present address. The counter is a fixed width for a given transaction burst length. The counter provides an address which wraps around the maximum length of the counter. This is consistent with burst transactions that do not start the transaction at the lowest address of the items in the data burst. The counter is configured to count in Intel order (refer to the *i486 Hardware Reference Manual*).

The address multiplexer is combinatorial logic that produces twelve address outputs for the DRAM array. Under control of the state machine, the multiplexer can output the refresh counter address, the column address, or the row address. The column address is the lowest-order bits of the address contained in the address register. The row address is a selected set of the upper-order bits of the address contained in the address register. The particular upper-order bits depend upon the DRAM type used in the array.

The state machine controls all actions of the DRAM controller including the timing for the DRAM access. The state diagram is given in *Figure 4* and the VHDL code is given in *Figure 5*. The state machine is designed so that the outputs come directly from a dedicated flip-flop, principally so that the bus acknowledge back to the system has a favorable set-up time for high-speed buses. Upon recognizing the assertion of the address flag from the address register, the state machine advances to the next state where the validity of the address is checked. The additional clock cycle from the receipt of the address flag to the test of the address validity allows

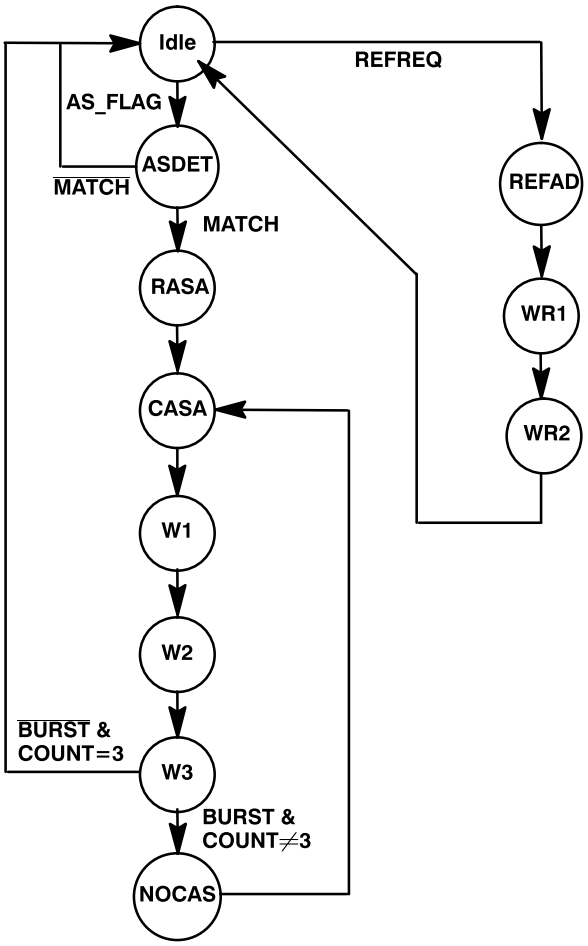

Figure 4. Controller State Machine State Diagram

time to test the address after it has entered the address register. If the upper address bits are 0, the access is taken to be valid for the DRAM memory and the state machine advances to start the memory cycle. If there is no address match, the state machine asserts the address flag clear signal to prepare for a subsequent transaction.

From the state machine *IDLE* state, the address multiplexer is set to select the upper portion of the stored address (the ROW address). With an address strobe detected, the state machine advances to the *ASDET* state where the address match is examined. If there is no match, the state machine returns to the *IDLE* state and if there is a match, the state machine advances to the *RASA* state. Here the state machine asserts RAS and simultaneously causes the address multiplexer to select the lower portion of the stored address (the COLUMN address). In the next cycle, the state machine advances to the *CASA* state where the DRAM CAS signal is asserted. The next cycles, W1, W2, and w3, are wait states keeping CAS and RAS asserted for the required time given in the DRAM data sheets. Depending upon the clock speed, W1 and/or W2 may be omitted. In state W3, a decision is made to return to the idle state if no burst is to be performed, or to advance to the state *NOCAS* if the transaction is a burst. In the *NOCAS* state, CAS is deasserted and then the state machine then loops back to the *casa* state to continue with the burst. The burst is ended with an examination of the burst counter in state w3. If the burst is completed, the state machine returns to the *IDLE* state. Refreshes are performed by advancing from the *IDLE* state to the *REFAD* state. Here, the address multiplexer is directed to output the address from the refresh address counter. In the next states, RAS is asserted for three cycles: *REF_RAS*, *WR1*, and *WR2*. Refresh is completed, and the state machine returns to the idle state. In the IDLE state the code gives priority to refresh requests over bus related requests (address strobe detection).

Although this is a high-level behavioral design, some aspects of the hardware have been taken into account. The statement

```
ATTRIBUTE state_encoding OF state:type IS
    ONE_HOT_ONE;
```

directs the synthesizer to use one flip-flop per state for the state machine. A portion of the synthesized logic is shown in *Figure 6*. Note the simplicity of the resulting logic equations. Outputs and transition terms include only the one flip-flop for the state in question. This results in a great simplification of the logic since complex state decoding is not required. Although this approach uses more flip-flops than encoded states, the decoding is far simpler. Most states and the related decoding should be realizable in one logic cell.

The state machine, by virtue of the VHDL coding, forces the outputs to be derived directly from the output of a flip-flop. This gives signals such as BACK (bus acknowledge) a short clock-to-output

```
-- State Machine process

     control:process
     begin
       wait until clk_in = '1';
       if(rs_in = '1') then
         state <= idle;
         cas_en <= '1';
         ras_en <= '1';
         back <= '1';
         col_sel <= '0';
         ref_sel <= '0';
       else
       case state is
         when idle =>
           cas_en <= '1';
           ras_en <= '1';
           back <= '1';
           if (ref_req = '1') then
             state <= refad;
             ref_sel <= '1';
           elsif (as_flag = '1') then
             state <= asdet;
             clr_as <= '1';
           end if;
         when asdet =>
           clr_as <= '0';
           if (match = '1') then
             state <= rasa;
             ras_en <= '0';
             col_sel <= '1';
             burst_flag <= burst_stored;
           else
             state <= idle;
           end if;
         when rasa =>
             state <= casa;
             cas_en <= '0';
             --back <= '0';

         when casa =>
             state <= w1;
             --back <= '1';

         when w1 =>
             state <= w2;
```

Figure 5. State Machine Behavioral Description

```
            when w2 =>
                state <= w3;
                back <= '0';

            when w3 =>
              cas_en <= '1';
              back <= '1';
              if(burst_flag = '1' and bst_cnt /= "11") then
                state <= nocas;
                col_sel <= '1';
              else
                state <= idle;
                ras_en <= '1';
                col_sel <= '0';
              end if;

            when nocas =>
                state <= casa;
                cas_en <= '0';
    -- Refresh
            when refad =>
                state <= wr1;
                ras_en <= '0';

            when wr1 =>
                state <= wr2;
            when wr2 =>
                state <= idle;
                ref_sel <= '0';
                ras_en <= '1';

          end case;
          end if;
        end process;
```

Figure 5. State Machine Behavioral Description (continued)

propagation delay, allowing it to meet the requirements of a high speed bus.

Design Analysis

The final step in the design process is to analyze the device performance and make adjustments to the

```
controller_0_state_bv_4_.D =
   controller_0_state_bv_3_.Q
 + controller_0_state_bv_8_.Q * burst_flag.Q * /bst_cnt_0__BEH_i42_0_DFF.Q
 + controller_0_state_bv_8_.Q * burst_flag.Q * /bst_cnt_1__BEH_i42_0_DFF.Q
```

Figure 6. A Portion of the Synthesized State Machine Logic Equations (State CASA)

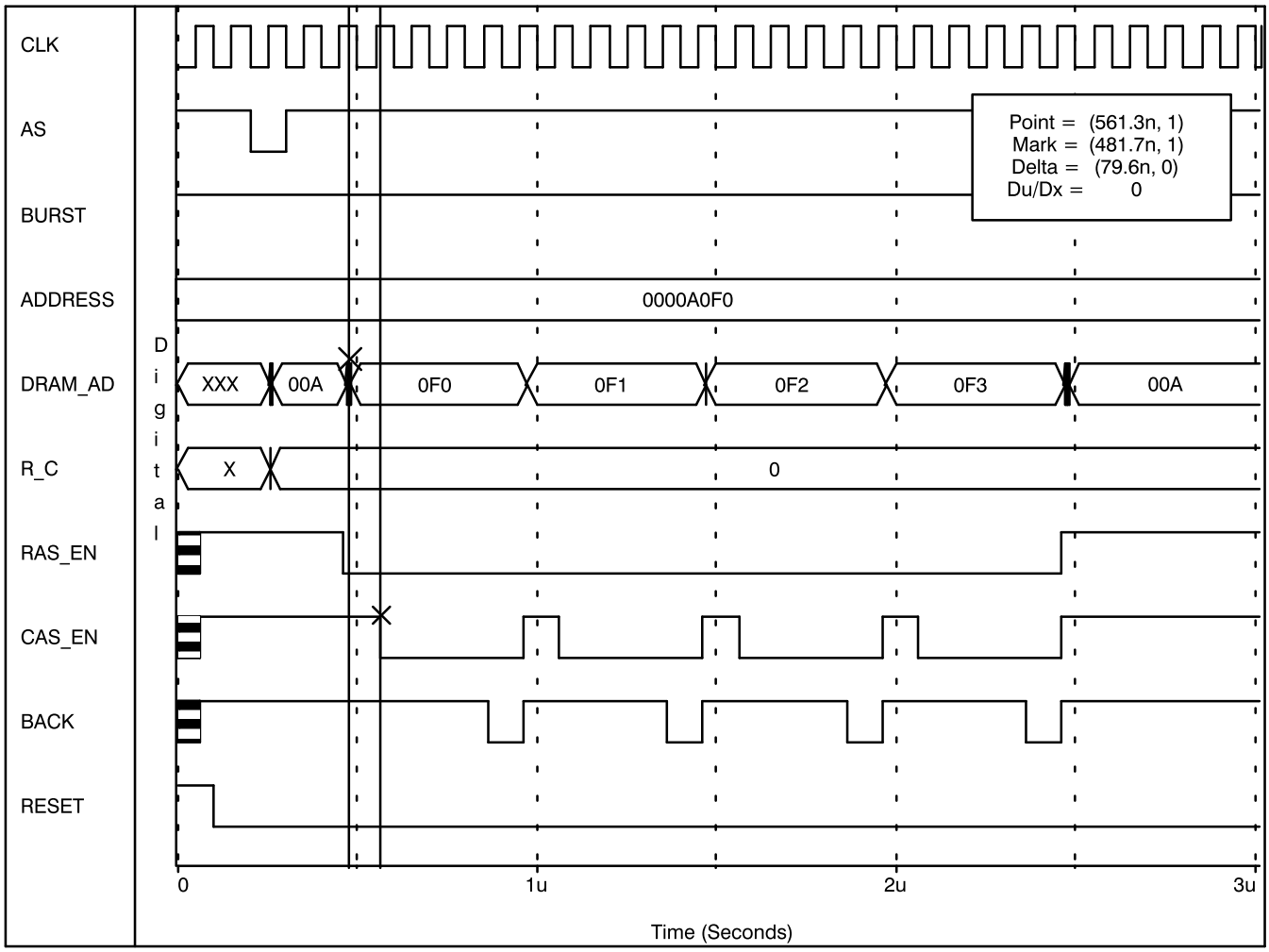

Figure 7. Graphic Output of the Simulator

design to optimize certain speed paths. At this point, the functionality of the device should be correct per the design specification as verified by the VHDL source level debugger. The performance optimization is accomplished using the timing analyzer. Timing measurements can be made in the simulation as well. There are several concerns in this design: the set-up time for the address and control information to the input register, the clock-to-output delay for the BACK signal, and the delay in the column address output. This latter concern is to determine if the multiplexing from the row to the column address (in state *rasa*) will cause the column address to be too close to the assertion of CAS (state *casa*) potentially violating the DRAM column address set up time. *Figure 7* illustrates a timing measurement being made in the simulation environment. The figure shows one burst transaction. A Mark and Point are placed at the falling edge of CAS_EN (the CAS enable output) and the place where the DRAM address (DRAM_AD) has switched to the column address and stabilized. The small window displays the Mark and Point times and DELTA is the time difference between these points.

As an example of an optimization the clock to output of the BACK signal is examined. *Table 2* shows the output of the Path Analyzer. The table shows a timing analysis result selecting the flip-flop to PAD options in the analyzer. The flip-flop to BACK path is then selected in the table of signal delays. The tabular results show a delay of 6.3 ns from the BACK

flip-flop output to the pad. The schematic-like representation (the physical view) of the device shows the generalized placement and signal routing in the device. The Physical View is shown in *Figure 8*. The selection of the flip-flop output to PAD path in the analyzer results table has caused that path to be highlighted in the schematic.

An improvement in this delay is sought by placing a timing constraint in the analyzer results table and rerunning the place and route. *Table 2* shows the new result after the place and route is rerun. The new place and route Physical View results are shown in *Figure 9*. Comparing *Figures 8* and *9* it can be observed that the flip-flop storing BACK has been moved closer to the pad resulting in a near 1 ns improvement in the clock to output delay for this signal. The analyzer shows that the variation in the clock pad to flip-flop delay is less than 100 ps, therefore it is not fruitful to attempt to improve the clock pad to BACK flip-flop delay. The other signal paths are similarly analyzed and optimized if necessary.

After the optimization is completed, the final design is simulated. There are two ways to define the stimulation for the simulation: graphical and textual. For designs with wide bus inputs or a large number of inputs, the most convenient method is textual input. The textual input of commands to drive the simulation is shown in *Figure 10*. Note in particular the command lines beginning with 'wfm'. These lines describe the waveform of the input signals. Of particular interest is the line beginning with wfm address, which describes the input address signal. The entire vector, address, can be assigned values in hex—a task which would be very cumbersome in a graphic input only environment. This has been done for the simulation result shown in *Figure 7*. The figure shows the graphical results of the simulation output along with the input stimuli.

Table 2. SpDE Path Analyzer

Path #	Delay	Delay Path	Constraint
−1−	4.0	RAS_CAS_3_−I2 −− RAS_CAS_3_	
−2−	4.0	RAS_CAS_0_−I2 −− RAS_CAS_0	
−3−	4.0	RAS_CAS_1_−I2 −− RAS_CAS_1	
−4−	4.3	RAS_CAS_2_−I2 −− RAS_CAS_2	
−5−	5.7	RAS_EN−I2 −− RAS_EN	
−6−	6.0	CAS_EN−I2 −− CAS_EN	
−7−	6.3	BACK−I2 −− BACK	
−8−	6.8	STORED_AD_19_ −− RC_AD_7_	
−9−	7.0	STORED_AD_16_ −− RC_AD_4_	
−10−	7.0	STORED_AD_23_ −− RC_AD_11_	
−11−	7.0	STORED_AD_21_ −− RC_AD_9_	
−12−	7.1	STORED_AD_15_ −− RC_AD_3_	
−13−	7.2	STORED_AD_13_ −− RC_AD_1_	
−14−	7.4	STORED_AD_14_ −− RC_AD_2_	
−15−	7.4	STORED_AD_18_ −− RC_AD_6_	
−16−	7.5	STORED_AD_20_ −− RC_AD_8_	
−17−	7.8	STORED_AD_12_ −− RC_AD_0_	
−18−	7.8	STORED_AD_22_ −− RC_AD_10_	
−19−	8.0	STORED_AD_17_ −− RC_AD_5_	
−20−	8.7	STORED_AD_19_ −− RC_AD_7_	
−21−	8.8	STORED_AD_23_ −− RC_AD_11_	
−22−	8.8	STORED_AD_16_ −− RC_AD_4_	
−23−	8.8	STORED_AD_21_ −− RC_AD_9_	
−24−	8.9	COL_AD_9_ −− RC_AD_9_	

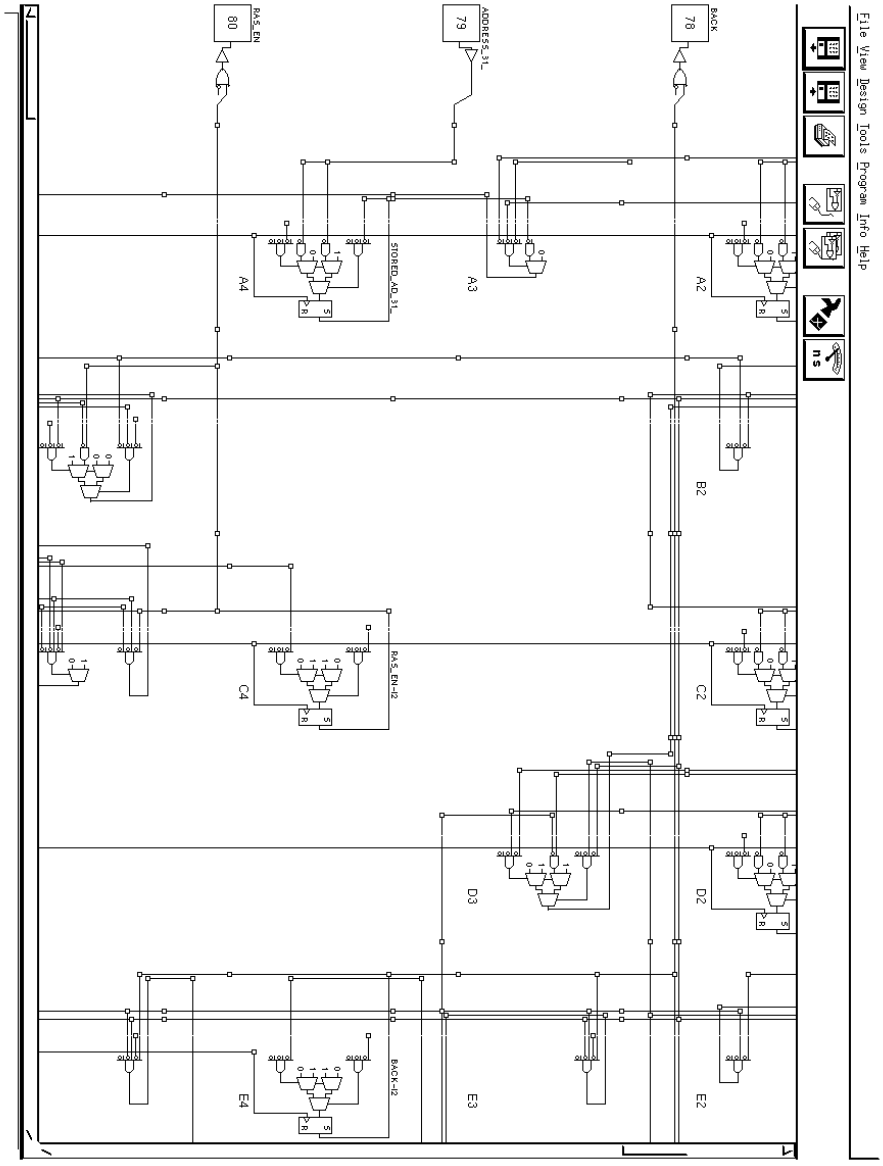

Figure 8. Physical View of Initial Design

Table 3. SpDE Path Analyzer with Applied Constraint

Path #	Delay	Delay Path	Constraint
−1−	4.0	RAS_CAS_3_−I2 −− RAS_CAS_3_	
−2−	4.0	RAS_CAS_0_−I2 −− RAS_CAS_0	
−3−	4.0	RAS_CAS_1_−I2 −− RAS_CAS_1	
−4−	4.3	RAS_CAS_2_−I2 −− RAS_CAS_2	
−5−	5.5	BACK−I2 −− BACK	4.0
−6−	5.7	RAS_EN−I2 −− RAS_EN	
−7−	6.0	CAS_EN−I2 −− CAS_EN	
−8−	6.8	STORED_AD_19_ −− RC_AD_7_	
−9−	7.0	STORED_AD_16_ −− RC_AD_4_	
−10−	7.0	STORED_AD_23_ −− RC_AD_11_	
−11−	7.0	STORED_AD_21_ −− RC_AD_9_	
−12−	7.1	STORED_AD_15_ −− RC_AD_3_	
−13−	7.2	STORED_AD_13_ −− RC_AD_1_	
−14−	7.4	STORED_AD_14_ −− RC_AD_2_	
−15−	7.4	STORED_AD_18_ −− RC_AD_6_	
−16−	7.5	STORED_AD_20_ −− RC_AD_8_	
−17−	7.8	STORED_AD_12_ −− RC_AD_0_	
−18−	7.8	STORED_AD_22_ −− RC_AD_10_	
−19−	8.0	STORED_AD_17_ −− RC_AD_5_	
−20−	8.7	STORED_AD_19_ −− RC_AD_7_	
−21−	8.8	STORED_AD_23_ −− RC_AD_11_	
−22−	8.8	STORED_AD_16_ −− RC_AD_4_	
−23−	8.8	STORED_AD_21_ −− RC_AD_9_	
−24−	8.9	COL_AD_9_ −− RC_AD_9_	

Summary

This application note is an introduction to FPGAs and high-level design tools. Specifics of the Cypress pASIC380 FPGA family of devices were presented along with the *Warp3* design tool set. A DRAM memory controller was presented to illustrate the global flow of a complete development. This was not meant to be a design tutorial for the design tools, schematic entry, VHDL encoding or design optimization, but rather it is intended as an overall map as to how a designer would proceed and the options available. The pASIC380 family FPGAs and the *Warp3* tool set is a powerful combination of device architecture and development tool that can help a designer achieve design success in a very short period of time.

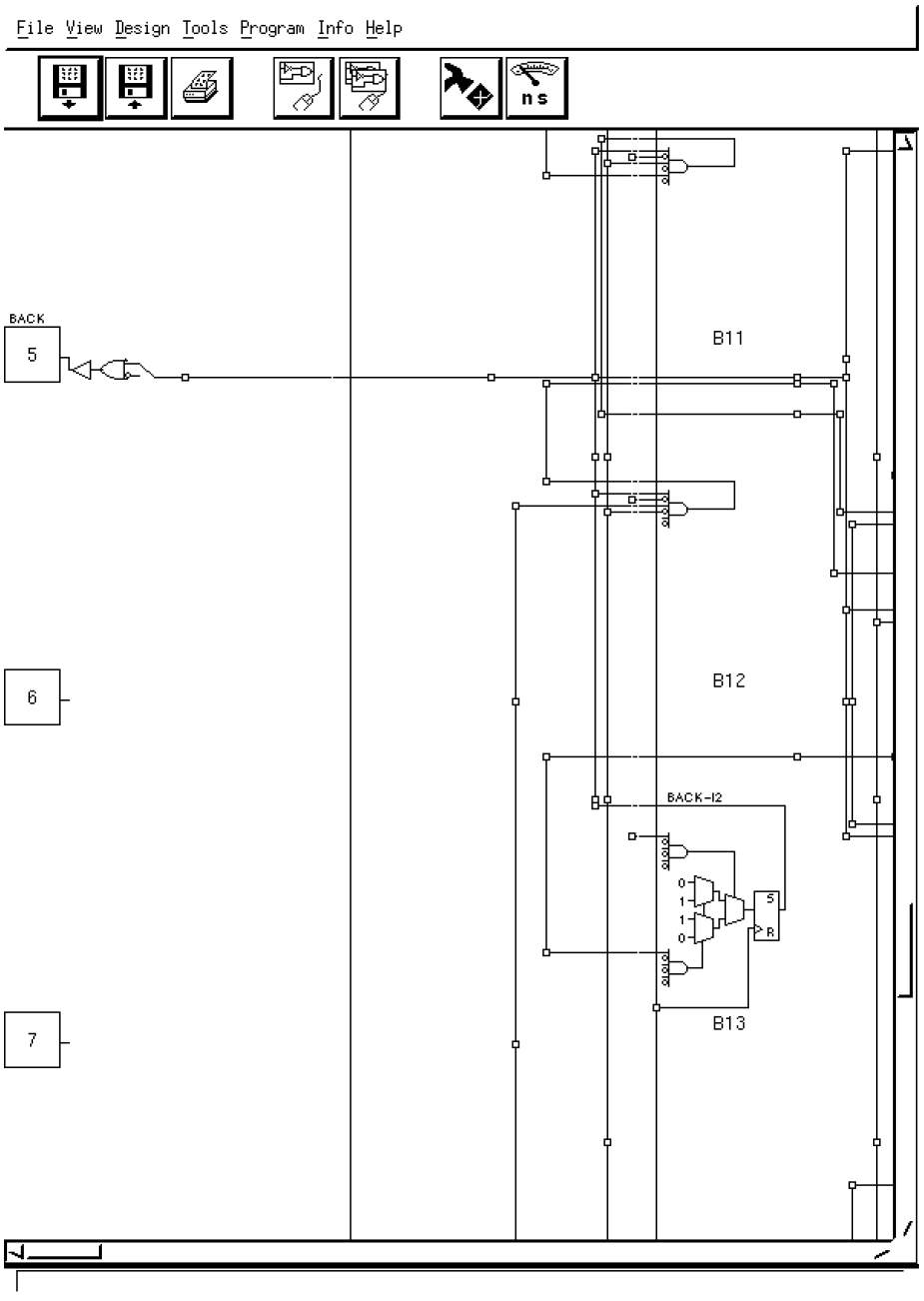

Figure 9. Physical View of Design Re−Placed and Routed with Constraint

```
| controller command file
logfile controller.log
stepsize 50ns
defaults –bignet –cmdfile –time
wave
vector address address[31:0]
radix hex address

vector dram_ad RC_AD[11:0]
radix hex dram_ad

vector r_c RAS_CAS[3:0]
radix hex r_c

wave controller.wfm clk as burst address dram_ad r_c ras_en cas_en back
       reset

clock clk 0 1

wfm reset @0=1 100ns=0
wfm as @0=1 200ns=0 100ns=1
wfm burst @0=1
wfm address @0=a0f0\h
cycle 200
log
```

Figure 10. Simulation Command File

Appendix A. Complete Design Behavioral Description

```
entity controller is
  port(clk,burst,as,reset: in bit;
    address: in bit_vector(31 downto 0);
    back,ras_en,cas_en: out bit;
    ras_cas: out bit_vector(3 downto 0);
    rc_ad: out bit_vector(11 downto 0));
attribute part_name of controller:entity is ”C384”;
end controller;

use work.bv_math.all;
use work.rtlpkg.all;

architecture behavior of controller is

  type name is (idle,asdet,rasa,casa,w1,w2,w3,nocas,refad,wr1,wr2);
  ATTRIBUTE state_encoding OF name: type IS ONE_HOT_ONE

  signal state: name;
  signal bst_cnt: bit_vector(1 downto 0);
  signal col_sel, ref_sel, as_flag, burst_flag, burst_stored, match,
      clr_as, ref_req, clk_in, ck_x, ck_y, as_x, as_in, rs_x, rs_in: bit;
  signal stored_ad: bit_vector(31 downto 0);
  signal re_ad, col_ad: bit_vector(11 downto 0);

  alias top_ad: bit_vector(3 downto 0) is stored_ad(31 downto 28);
  alias row_ad: bit_vector(11 downto 0) is stored_ad(23 downto 12);
  alias bank: bit_vector(3 downto 0) is stored_ad(27 downto 24);

begin

-- Special IO ports for device

  ck1:PAckcell
    PORT MAP(clk, ck_x, ck_y, clk_in);

  hd1:PAincell
    PORT MAP(as, as_x, as_in);

  hd2:PAincell
    PORT MAP(reset, rs_x, rs_in);

-- Address register process

  adreg:process
  begin
    wait until clk_in = ’1’;
```

Appendix A. Complete Design Behavioral Description (continued)

```
    if(rs_in = '1') then
      as_flag <= '0';
    elsif(as_in = '0') then
      as_flag <= '1';
    elsif(clr_as = '1') then
      as_flag <= '0';
    end if;

    if(as_in = '0') then
      stored_ad <= address;
      burst_stored <= burst;
    end if;

  end process;

-- Match Comparator

  match <= '1' when top_ad = "0000" else '0';

-- DRAM address multiplexer
  re_ad <= "000000000000";
  ref_req <= '0';

  mux:process(col_ad,col_sel,row_ad,re_ad,state)
  begin
    if(state = refad or state = wr1 or state = wr2) then
      rc_ad <= re_ad;
    elsif(col_sel = '1') then
      rc_ad <= col_ad;
    else
      rc_ad <= row_ad;
    end if;
  end process;

-- Encoded RAS / CAS select

  ras_cas <= bank;

-- Column Address, Intel Order

  col_ad(11 downto 2) <= stored_ad(11 downto 2);
  col_ad(1) <= stored_ad(1) xor bst_cnt(1);
  col_ad(0) <= stored_ad(0) xor bst_cnt(0);

-- State Machine process

  control:process
  begin
    wait until clk_in = '1';
```

Appendix A. Complete Design Behavioral Description (continued)

```
if(rs_in = '1') then
  state <= idle;
  cas_en <= '1';
  ras_en <= '1';
  back <= '1';
  col_sel <= '0';
  ref_sel <= '0';
else
case state is
  when idle =>
    cas_en <= '1';
    ras_en <= '1';
    back <= '1';
    if (ref_req = '1') then
      state <= refad;
      ref_sel <= '1';
    elsif (as_flag = '1') then
      state <= asdet;
      clr_as <= '1';
    end if;
  when asdet =>
    clr_as <= '0';
    if (match = '1') then
      state <= rasa;
      ras_en <= '0';
      col_sel <= '1';
      burst_flag <= burst_stored;
    else
      state <= idle;
    end if;
  when rasa =>
      state <= casa;
      cas_en <= '0';
      --back <= '0';

  when casa =>
      state <= w1;
      --back <= '1';

  when w1 =>
      state <= w2;

  when w2 =>
      state <= w3;
      back <= '0';
```

Appendix A. Complete Design Behavioral Description (continued)

```
      when w3 =>
        cas_en <= '1';
        back <= '1';
        if(burst_flag = '1' and bst_cnt /= "11") then
          state <= nocas;
          col_sel <= '1';
        else
          state <= idle;
          ras_en <= '1';
          col_sel <= '0';
        end if;

      when nocas =>
          state <= casa;
          cas_en <= '0';
      when refad =>
          state <= wr1;
          ras_en <= '0';

      when wr1 =>
          state <= wr2;
      when wr2 =>
          state <= idle;
          ref_sel <= '0';
          ras_en <= '1';

    end case;
    end if;
  end process;

-- Burst counter

  burst_count:process
  begin
    wait until clk_in = '1';
    if(state = idle) then
      bst_cnt <= "00";
    elsif(state = w3) then
      bst_cnt <= inc_bv(bst_cnt);
    end if;
  end process;

end behavior;
```