CY3125

Warp® CPLD Development Tool for UNIX

Features

- **VHDL (IEEE 1076 and 1164) and Verilog (IEEE 1364) high-level language compilers with the following features:**
  - **Designs are portable across multiple devices and/or EDA environments**
  - **Facilitates the use of industry-standard simulation and synthesis tools for board and system-level design**
  - **Support for functions and libraries facilitating modular design methodology**
- **IEEE Standard 1076 and 1164 VHDL synthesis supports:**
  - **Enumerated types**
  - **Operator overloading**
  - **For... Generate statements**
  - **Integers**
- **IEEE Standard 1364 Verilog synthesis supports:**
  - **Reduction and conditional operators**
  - **Blocking and non-blocking procedural assignments**
  - **While loops**
  - **Integers**
- **Several design entry methods support high-level and low-level design descriptions:**
  - **Behavioral VHDL and Verilog (IF...THEN...ELSE; CASE...)**
  - **Boolean**
  - **Structural Verilog and VHDL**
  - **Designs can include multiple entry methods (but only one HDL language) in a single design.**
- **UltraGen™ Synthesis and Fitting Technology:**
  - **Infers "modules" such as adders, comparators, etc., from behavioral descriptions and replaces them with circuits pre-optimized for the target device.**
  - **User-selectable speed and/or area optimization on a block-by-block basis**
  - **Perfect communication between synthesis and fitting**
  - **Automatic selection of optimal flip-flop type (D type/T type)**
  - **Automatic pin assignment**
- **Supports for the following Cypress Programmable Logic Devices:**
  - **PSI™ (Programmable Serial Interface™)**
  - **Delta39K™ CPLDs**
  - **Quantum38K™ CPLDs**
  - **Ultra37000™ CPLDs**
  - **FLASH370i™ CPLDs**
  - **MAX340™ CPLDs**
  - **Industry-standard PLDs (16V8, 20V8, 22V10)**
- **VHDL and Verilog timing model output for use with third-party simulators**
- **Static Timing Report:**
  - **Provides timing information for any path broken down by the different steps of the path**
- **Architecture Explorer and Dynamic Timing Analysis for PSI, Delta39K and Quantum38K devices:**
  - **Graphical representation of exactly how your design will be implemented on your specific target device**
  - **Zoom from the device level down to the macrocell level**
  - **Determine the timing for any path and view that path on a graphical representation of the chip**
- **Workstation support for Sun Solaris™**
- **On-line documentation and help**

Functional Description

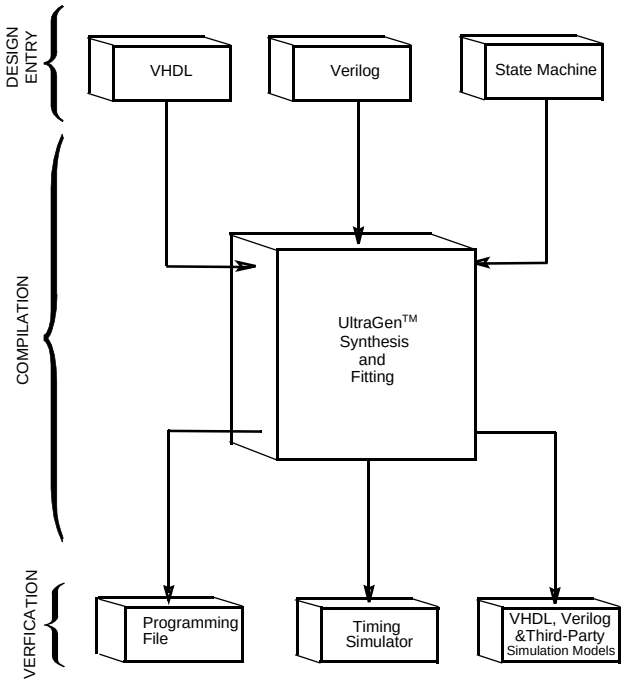

Figure 1. *Warp*® VHDL Design Flow

Warp® is a state-of-the-art HDL compiler for designing with Cypress's Complex Programmable Logic Devices (CPLDs). *Warp* utilizes a subset of IEEE 1076/1164 VHDL and IEEE 1364 Verilog as its Hardware Description Languages (HDL) for design entry. Then, it synthesizes and optimizes the entered design, and outputs a JEDEC or Intel hex file for the desired PLD or CPLD (see *Figure 1*). Furthermore, *Warp* accepts VHDL or Verilog produced by the Active-HDL FSM graphical Finite State Machine editor. For simulation, *Warp* provides a timing simulator, as well as VHDL and Verilog timing models for use with third party simulators.

VHDL and Verilog Compilers

VHDL and Verilog are powerful, industry standard languages for behavioral design entry and simulation, and are supported by all major vendors of EDA tools. They allow designers to learn a single language that is useful for all facets of the design process.

VHDL and Verilog offer designers the ability to describe designs at many different levels. At the highest level, designs can be entered as a description of their behavior. This behavioral description is not tied to any specific target device. As a result, simulation can be done very early in the design to verify correct functionality, which significantly speeds the design process.

The *Warp* syntax for VHDL and Verilog includes support for intermediate level entry modes such as state tables and Boolean entry. At the lowest level, designs can be described using gate-level descriptions. *Warp* gives the designer the flexibility to intermix all of these entry modes.

In addition, Verilog and VHDL allow you to design hierarchically, building up entities in terms of other entities. This allows you to work either “top-down” (designing the highest levels of the system and its interfaces first, then progressing to greater and greater detail) or “bottom-up” (designing elementary building blocks of the system, then combining these to build larger and larger parts) with equal ease.

Because these languages are IEEE standards, multiple vendors offer tools for design entry and simulation at both high and low levels and synthesis of designs to different silicon targets. The use of device-independent behavioral design entry gives users the freedom to easily migrate to high-volume technologies. The wide availability of VHDL and Verilog tools provides complete vendor independence as well. Designers can begin their project using *Warp* for Cypress CPLDs and convert to high volume ASICs using the same VHDL or Verilog behavioral description with industry-standard synthesis tools.

The VHDL and Verilog languages also allow users to define their own functions. User-defined functions allow users to extend the capabilities of the language and build reusable files of tested routines. VHDL and Verilog provide control over the timing of events or processes. They have constructs that identify processes as either sequential, concurrent, or a combination of both. This is essential when describing the interaction of complex state machines.

VHDL and Verilog are rich programming languages. Their flexibility reflects the nature of modern digital systems and allows designers to create accurate models of digital designs. Because they are not verbose languages they are easy to learn and compile. In addition, models created in VHDL and Verilog can readily be transported to other EDA Environments. *Warp* supports IEEE 1076/1164 VHDL including loops, for/generate statements, full hierarchical designs with packages, enumerated types, and integers as well as IEEE 1364 Verilog including loops, reduction and conditional operators.

A VHDL Design Example

Design Entry

Warp descriptions specify:

- The behavior or structure of a design, and
- The mapping of signals in a design to the pins of a PLD/CPLD (optional)

The part of a *Warp* description that specifies the behavior or structure of the design is called an entity/architecture pair. Entity/architecture pairs, as their name implies, are divided into two parts: an entity declaration, which declares the design’s interface signals (i.e., defines what external signals the design has, and what their directions and types are), and a design architecture, which describes the design’s behavior or structure.

The entity portion of a design file is a declaration of what a design presents to the outside world (the interface). For each external signal, the entity declaration specifies a signal name, a direction and a data type. In addition, the entity declaration specifies a name by which the entity can be referenced in a design architecture. This section shows code segments from five sample design files. The top portion of each example features the entity declaration.

Behavioral Description

The architecture portion of a design file specifies the function of the design. As shown in *Figure 1*, multiple design-entry methods are supported in *Warp*. A behavioral description in VHDL often includes well known constructs such as If...Then...Else, and Case statements. Here is a code segment from a simple state machine design (soda vending machine) that uses behavioral VHDL to implement the design:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY drink IS
 PORT (nickel,dime,quarter,clock:#in
std_logic;
   returnDime,returnNickel,giveDrink:out

std_logic);
END drink;


ARCHITECTURE fsm OF drink IS

TYPE drinkState IS (zero,five,ten,fifteen,
twenty,twentyfive,owedime);
SIGNAL drinkstatus:drinkState;

BEGIN

PROCESS BEGIN

 WAIT UNTIL clock = ’1’;

  giveDrink <= ’0’;
  returnDime <= ’0’;
  returnNickel <= ’0’;

  CASE drinkStatus IS

  WHEN zero =>
    IF (nickel = ’1’) THEN
      drinkStatus <= five;
    ELSIF (dime = ’1’) THEN
      drinkStatus <= Ten;
    ELSIF (quarter = ’1’) THEN
      drinkStatus <= twentyfive;
    END IF;
  WHEN five =>
```

```
    IF (nickel = '1') THEN
      drinkStatus <= ten;
    ELSIF (dime = '1') THEN
      drinkStatus <= fifteen;
    ELSIF (quarter = '1') THEN
      giveDrink <= '1';
      drinkStatus <= zero
    END IF;

  -- Several states are omitted in this
  -- example. The omitted states are ten,
  -- fifteen, twenty, and twentyfive.

  WHEN owedime =>
    returnDime <= '1';
    drinkStatus <= zero;

  when others =>
  -- This makes sure that the state
  -- machine resets itself if
  -- it somehow gets into an undefined state.
    drinkStatus <= zero;
  END CASE;
  END PROCESS;

END FSM;
```

VHDL is a strongly typed language. It comes with several predefined operators, such as + and /= (add, not-equal-to). VHDL offers the capability of defining multiple meanings for operators (such as +), which results in simplification of the code written. For example, the following code segment shows that "count <= count +1" can be written such that count is a std_logic_vector, and 1 is an integer.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

ENTITY sequence IS
  port (clk: in std_logic;
    s : inout std_logic);
end sequence;

ARCHITECTURE fsm OF sequence IS

SIGNAL count: std_logic_vector(3 downto 0);

BEGIN

PROCESS BEGIN

  WAIT UNTIL clk = '1';

    CASE count IS

      WHEN x"0" | x"1" | x"2" | x"3" =>
        s <= '1';
        count <= count + 1;
      WHEN x"4" | x"5" | x"6" | x"7" =>
        s <= '0';
        count <= count + 1;
      WHEN x"8" | x"9" =>
        s <= '1';
        count <= count + 1;
      WHEN others =>
        s <= '0';
        count <= (others => '0');
    END CASE;

END PROCESS;

END FSM;
```

In this example, the + operator is overloaded to accept both integer and std_logic arguments. *Warp* supports overloading of operators.

Functions

A major advantage of VHDL is the ability to implement functions. The support of functions allows designs to be reused by simply specifying a function and passing the appropriate parameters. *Warp* features some built-in functions such as ttf (truth-table function). The ttf function is particularly useful for state machine or look-up table designs. The following code describes a seven-segment display decoder implemented with the ttf function:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.table_std.all;

ENTITY seg7 IS
  PORT(
    inputs: IN STD_LOGIC_VECTOR (0 to 3)
    outputs: OUT STD_LOGIC_VECTOR (0 to 6)
  );
END SEG7;

ARCHITECTURE mixed OF seg7 IS

CONSTANT truthTable:
  ttf_table (0 to 11, 0 to 10) := (
-- input&    output
-- ----------------------
  "0000"&  "0111111",
  "0001"&  "0000110",
  "0010"&  "1011011",
  "0011"&  "1001111",
  "0100"&  "1100110",
  "0101"&  "1101101",
  "0110"&  "1111101",
  "0111"&  "0000111",
  "1000"&  "1111111",
  "1001"&  "1101111",
  "101-"&  "1111100", --creates E pattern
  "111-"&  "1111100"
  );

BEGIN

    outputs <= ttf(truthTable,inputs);

END mixed;
```

Boolean Equations

A third design-entry method available to *Warp* users is Boolean equations. *Figure 2* displays a schematic of a simple one-bit half adder. The following code describes how this one-bit half adder can be implemented in *Warp* with Boolean equations:

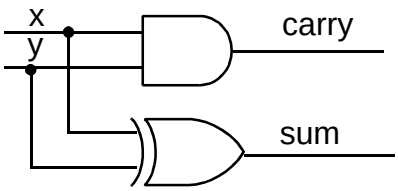

Figure 2. One-Bit Half Adder

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

--entity declaration
ENTITY half_adder IS
  PORT (x, y : IN std_logic;
    sum, carry : OUT std_logic);
END half_adder;
--architecture body
ARCHITECTURE behave OF half_adder IS
BEGIN
  sum <= x XOR y;
  carry <= x AND y;
END behave;
```

Structural VHDL

While all of the design methodologies described thus far are high-level entry methods, structural VHDL provides a method for designing at a very low level. In structural descriptions, the designer simply lists the components that make up the design and specifies how the components are wired together. *Figure 3* displays the schematic of a simple 3-bit shift register and the following code shows how this design can be described in *Warp* using structural VHDL:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.rtlpkg.all;

ENTITY shifter3 IS port (
    clk : IN STD_LOGIC;
    x : IN STD_LOGIC;
    q0 : OUT STD_LOGIC;
    q1 : OUT STD_LOGIC;
    q2 : OUT STD_LOGIC);
  END shifter3;

ARCHITECTURE struct OF shifter3 IS
  SIGNAL q0_temp, q1_temp, q2_temp : STD_LOGIC;
  BEGIN
    d1 : DFF PORT MAP(x,clk,q0_temp);
    d2 : DFF PORT MAP(q0_temp,clk,q1_temp);
    d3 : DFF PORT MAP(q1_temp,clk,q2_temp);
    q0 <= q0_temp;
    q1 <= q1_temp;
    q2 <= q2_temp;
  END struct;
```

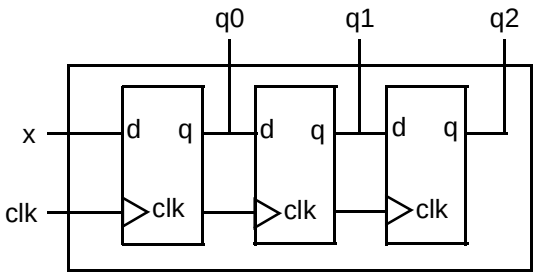

Figure 3. Three-Bit Shift Register Circuit Design

All of the design-entry methods described can be mixed as desired. VHDL has the ability to combine both high- and low-level entry methods in a single file. The flexibility and power of VHDL allows users of *Warp* to describe designs using whatever method is appropriate for their particular design.

A Verilog Design Example

Design Entry

Warp descriptions specify:

- The behavior or structure of a design, and
- the mapping of signals in a design to the pins of a PLD/CPLD (optional)

The part of a *Warp* description that specifies the behavior or structure of the design is called a module. The module declares the design's interface signals (i.e., defines what external signals the design has, and what their directions and types are).

The module portion of a design file is a declaration of what a design presents to the outside world (the interface). For each external signal, the module specifies a signal name, a direction and a data type. In addition, the module declaration specifies a name by which the entity can be referenced in other modules. This section shows code segments from four sample design files. The top portion of each example features the module declaration.

Behavioral Description

The module portion of a design file specifies the function of the design. As shown in *Figure 1*, multiple design-entry methods are supported in *Warp*. A behavioral description in Verilog often includes well known constructs such as If…Else, and Case statements. Here is a code segment from a simple state machine design (soda vending machine) that uses behavioral Verilog to implement the design:

```
MODULE drink (nickel, dime, quarter, clock,
            returnDime, returnNickel,
            giveDrink);

  INPUT nickel, dime, quarter, clock;
  OUTPUT returnDime,returnNickel,giveDrink;
  REG returnDime, returnNickel, giveDrink;

  PARAMETER zero = 0, five = 1, ten = 2,
  fifteen = 3, twenty = 4, twentyfive = 5
  owedime = 6;

REG[1:0] drinkStatus;

ALWAYS@ (POSEDGE clock)
```

```
BEGIN

 giveDrink = 0;
 returnDime = 0;
 returnNickel = 0;

 CASE(drinkStatus)

   zero: BEGIN
      IF (nickel)
        drinkStatus = five;
      ELSE IF (dime)
        drinkStatus = ten;
      ELSE IF (quarter)
        drinkStatus = twentyfive;
   END

   five: BEGIN
      IF (nickel)
        drinkStatus = ten;
      ELSE IF (dime)
        drinkStatus = fifteen;
      ELSE IF (quarter)
        BEGIN
          drinkStatus = zero;
          giveDrink = 1;
        END

   END

   // Several states are omitted in this
   // example. The omitted states are ten
   // fifteen, twenty, and twentyfive.

   owedime: BEGIN
      returnDime = 1;
      drinkStatus = zero;
   END

   default: BEGIN
   // This makes sure that the state
   // machine resets itself if
   // it somehow gets into an undefined state.
      drinkStatus = zero;
   END

 ENDCASE
 END
 ENDMODULE
```

Verilog is not a strongly typed language. The simplicity and readability of the following code is increased by use of the CASEX. The CASEX command accepts “Don’t Cares” and chooses the branch depending on the value of the expression.

```
MODULE sequence (clk, s);
  INPUT clk;
  INOUT s;
  WIRE s;
  REG temp;
  REG[3:0] count;

ALWAYS@(POSEDGE clk)
  CASEX(count)
4'b00XX: BEGIN
             temp=1;
             count=count+1;
             end
4'b01XX: BEGIN
             temp=0;
             count=count+1;
             end
4'b100X: BEGIN
             temp=1;
             count=count+1;
             end
default: BEGIN
             temp=0;
             count=0;
             end

ENDCASE
ASSIGN s=temp;
ENDMODULE
```

Boolean Equations

A second design-entry method available to *Warp* Verilog users is Boolean equations. *Figure 4* displays a schematic of a simple one-bit half adder. The following code describes how this one-bit half adder can be implemented in *Warp* with Boolean equations:

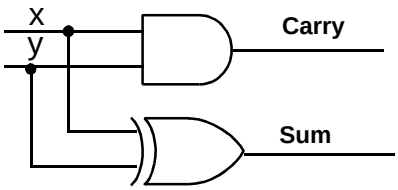

Figure 4. One-Bit Half Adder

```
MODULE half_adder(x, y, sum, carry);
  INPUT x, y;
  OUTPUT sum, carry;

ASSIGN sum = x^y;
ASSIGN carry = x&y;
ENDMODULE
```

Structural Verilog

While all of the design methodologies described thus far are high-level entry methods, structural Verilog provides a method for designing at a very low level. In structural descriptions, the designer simply lists the components that make up the design and specifies how the components are wired together.
Figure 5 displays the schematic of a simple 3-bit shift register and the following code shows how this design can be described in *Warp* using structural Verilog.

```
MODULE shifter3 (clk, x, q0, q1, q2);
  INPUT clk, x;
  OUTPUT q0, q1, q2;
  WIRE q0, q1, q2;
  REG q0_temp, q1_temp, q2_temp;

    DFF d1(x,clk,q0_temp);
    DFF d2(q0_temp,clk,q1_temp);
    DFF d3(q1_temp,clk,q2_temp);
    ASSIGN q0 = q0_temp;
    ASSIGN q1 = q1_temp;
    ASSIGN q2 = q2_temp;

ENDMODULE;
```

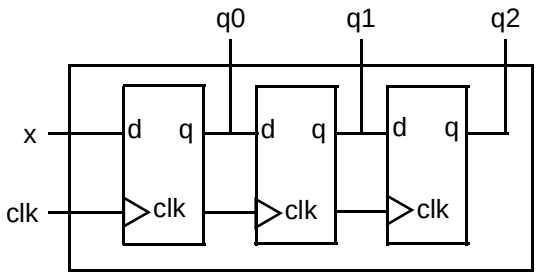

Figure 5. Three-Bit Shift Register Circuit Design

All of the design-entry methods described can be mixed as desired. Verilog has the ability to combine both high- and low-level entry methods in a single file. The flexibility and power of Verilog allows users of *Warp* to describe designs using whatever method is appropriate for their particular design.

Compilation

Once the VHDL or Verilog description of the design is complete, it is compiled using *Warp*. Although implementation is with a single command, compilation is actually a multistep process as shown in *Figure 1*. The first part of the compilation process is the same for all devices. The input description is synthesized to a logical representation of the design. *Warp* synthesis is unique in that the input languages support device-independent design descriptions. Competing programmable logic compilers require very specific and device-dependent information in the design description.

Warp synthesis is based on UltraGen technology. This technology allows *Warp* to infer adders, subtractors, multipliers, comparators, counters and shifters from the behavioral descriptions. *Warp* then replaces these operators internally with an architecture-specific circuit. This circuit or “module” is also pre-optimized for either area or speed. *Warp* uses the appropriate implementation based on user directives.

The second step of compilation is an iterative process of optimizing the design and fitting the logic into the targeted device. Logical optimization in *Warp* is accomplished using Espresso algorithms. The optimized design is automatically fed to the *Warp* fitter for targeting a PLD or CPLD. This fitter supports the automatic or manual placement of pin assignments as well as automatic selection of D or T flip-flops. After optimization and fitting, *Warp* creates a JEDEC or Intel hex file for the specified PLD or CPLD.

Automatic Error Tracking

Warp features automatic error location that allows problems to be diagnosed and corrected in seconds. Errors from compilation are displayed immediately in a window. If the user highlights a particular error, *Warp* will automatically open the source code file and highlight the offending line in the entered design. If the device fitting process includes errors, a window will again describe them. A detailed report file is generated indicating the resources required to fit the input design and any problems that occurred in the process.

Simulation

Warp outputs standard VHDL and Verilog timing models that can be used with third-party simulators to perform functional and timing verifications of the synthesized design.

Architecture Explorer

The Architecture Explorer graphically displays how the design will be implemented on the chip. It provides a view of the entire device to show what memory elements and logic clusters have been used for what part of the design. This gives the designer an idea of what resources are free. The Architecture Explorer allows you to zoom in multiple times. At maximum zoom it displays the logic gate implementation in each macrocell. The Architecture Explorer is available for PSI, Delta39K and Quantum38K devices.

Timing Analyzer

The Timing Analyzer gives the time across any path as well as the breakdown of what steps are causing the timing delays. This tool does not simply display the general specification for the target device but a worst-case simulation of the actual path being taken through the device. When you highlight a path on the timing analyzer, the source and destination of that path are displayed on the Architecture Explorer. The timing analyzer is also available for PSI™, Delta39K™ and Quantum38K™ devices.

Programming

Cypress’s FLASH370i, Ultra37000, Quantum38K, Delta39K, and PSI In-System Reprogrammable™ (ISR™) devices can be programmed on board with an ISR programmer. For PSI, Delta39K and Quantum38K CPLDs *Warp* produces an Intel hex file. The ISR programmer converts this file into STAPL and programs the device. For Ultra37000 and FLASH370i devices, *Warp* produces a JEDEC file. For Ultra37000, the ISR programmer converts this file into JAM/STAPL and programs the device. For FLASH370i, the JEDEC file is used directly to program the device.

The JEDEC and Intel hex files produced by *Warp* can also be used with any qualified third party programmer to program Cypress CPLDs.

For more information on Cypress’s ISR software see the ISR Programming Kit (CY3900i) data sheet.

System Requirements

- 32 MB of RAM (64 MB recommended)
- 110 MB Disk Space
- CD-ROM drive
- Solaris 2.5 or later

Warp includes:

- CD-ROM with *Warp* and on-line documentation (Getting Started Manual, User's Guide, HDL Reference Manual)
- *VHDL for Programmable Logic* Textbook
- Registration Card

Product Ordering Information

Product Code	Description
CY3125R62	*Warp* development system for UNIX

Warp Enterprise, UltraGen, Ultra37000, Quantum38K, Delta39K, PSI, Programmable Serial Interface, MAX340, ISR, In-System Reprogrammable, and FLASH370i are trademarks of Cypress Semiconductor Corporation.
Warp is a registered trademark of Cypress Semiconductor Corporation.
Solaris is a trademark of Sun Microsystems Corporation.
Active-HDL is a trademark of Aldec Incorporated.

Document Title: CY3125 *Warp*® CPLD Development Tool for UNIX Document Number: 38-03046				
REV.	**ECN NO.**	**Issue Date**	**Orig. of Change**	**Description of Change**
**	109903	09/22/01	SZV	Change from Spec number: 38-01033 to 38-03046
*A	111243	01/22/02	CNH	Update product code