Understanding the *Warp*™ Report File for Ultra37000™ Devices

Introduction

Cypress provides HDL synthesis for programmable logic with a series of software suites called *Warp*. Different versions of *Warp* carry different capabilities for design entry and verification, but they all share the core synthesis engine in common. *Warp* is actually comprised of several command-line driven executables; the graphical user interface, or GUI, that automatically spawns the elements of *Warp* is called Galaxy. *Warp* is used to synthesize code written in VHDL or Verilog formats and fit the functions into Cypress programmable logic devices. As *Warp* performs the functional synthesis, technology mapping and fitting—called compilation—it generates several output files, including a report file. This application note provides a comprehensive description of the report file.

The compilation process can be broken down to the following steps: Parsing, High-Level Synthesis, Synthesis and Optimization (library optimization), Technology Mapping and Optimization (architectural and Boolean optimization), Fitting, JEDEC File Generation, and Static Timing Path Analysis. (*Warp* will optionally generate a post-synthesis simulation model, but there is no reference to this step in the report file.) When the compiler is processing a lower-level library file, or a library project, it stops after the first two steps. The remaining five steps are only included when the top-level file in a device project is compiled. The report file carries the same file name as the top-level source file with the extension “RPT”.

Each step produces different varieties of information. This information could be useful in several ways. It could be used for understanding how high-level code is interpreted and synthesized into Boolean functions, for verifying the Boolean equations against the intended behavior, or for understanding how low-level equations are translated and fit into physical gates. It could be used for understanding the results of certain design trade-offs, for improving the fitter’s results, or even for performing low-level debugging.

This application note is divided into two sections. The **Compiler Summary** section lists each of the compiler’s major steps, along with *How Is This Useful* tips on the information included in the report file. This section is intended to provide a high-level view of the compiler functions and report file, and to help the user quickly identify where to go for specific help and information. The **Detailed Compiler and Report File Descriptions** section is numbered and subtitled like the **Compiler Summary** section, and contains the same tips, with more detail.

The report file may contain error messages and warnings, when appropriate, from any one of the compilation steps. This application note will not contain any details on those messages. Please refer to Appendix A of the *Warp* Reference Manual for explanations of error messages and warnings.

Compiler Summary

1. Parser

The parser verifies correct language syntax (grammar), some semantics (usage), and correct instantiations of library components. It also performs “Module Generation” (described in *Data Path Operators*) during this first step.

2. High Level Synthesis

This step reduces the high-level VHDL or Verilog to an intermediate format called an expression tree. It also optimizes out the I/O or entity portion of unused logic.

3. Synthesis and Optimization

The synthesis portion performs state machine synthesis. The optimization portion optimizes through substitution, removes unused logic at the gate or architecture level, and performs initial Boolean-level logic reduction.

Any logic that is defined by the source file, but has no destination for its output is removed here. Likewise, nodes that contain no logic, or nodes with two separate names are factored out of the low-level equations. If an equation is dependent on the combinatorial output of another equation, the second equation is substituted into the first in place of its output signal name. These are called expanded signals. Soft nodes are factoring points that are earmarked for conversion into physical nodes during the fitting process.

4. Technology Mapping and Optimization

This process chooses flip-flop type and equation polarity according to the options available in the target PLD to minimize product term usage. It also performs sum splitting and global resource reduction. Finally, it converts all the equations into a format appropriate to the target architecture.

If an equation is dependent on the combinatorial output of another equation, the second equation is substituted into the first in place of its output signal name. For most designs compiled in *Warp* the substitution is done earlier and this process only removes redundant buffers and wires. When expansion has been done to its limit, the resulting soft nodes, registered equations, and outputs are committed to macrocell or input register resources.

The number of product terms used to implement an XOR function can sometimes be reduced by using a T flip-flop. Additionally, since the logical inversion of both sides of an equation trades sums for products and products for sums, in some cases implementing the inverse of an equation can require fewer product terms. The sum-of-products (for D flip-flops), their inverse, and T flip-flop version of each signal is generated and evaluated. Finally, for equations that require more than the maximum number of product terms provided to a macrocell, the final sum or OR function must be split between two or more macrocells.

5. Fitting

This compilation step assigns the final equations and functions to physical hardware locations in the target device.

The most useful information in the report file is provided during this step. The final equations are listed here, in Boolean format. Next, a graphical representation of the product term array placement map is provided. Next, tables on logic bock utilization, and the I/O assignment for each logic block are provided. Finally, the report file shows the total usage of the part for every logic block and for global resources.

The following series of steps comprise the actual fitting process. The series starts with fixed pin assignments and global resources, and builds the design through more flexible resource assignments. Signals are first assigned to logic blocks, and then assigned to specific macrocells. Then product term sharing is evaluated and performed, product terms are assigned, and input pins are assigned.

The fitter lists several statistics relevant to signal and product term placement and utilization.

Design Equations

⇒ How Is This Useful?⇐

The "Design Equations" section is a good beginning place to verify that the design written in HDL eventually compiled to functions in Boolean algebra that resemble the designer's original intent. Ultimately, functional simulation will verify the design, but a cursory look at the equations at this level can identify typos that had correct syntax, but that created obviously incorrect functions.

⇒ How Is This Useful?⇐

The "Design Equations" section is very useful for determining how well the design compiled into the target device. To get the maximum performance out of the CPLD, it is usually best to minimize the number of passes through the array for each expression. The compiler will implement multiple passes for several reasons. A resulting equation may have been too complex to implement in the available product terms, so the compiler created factoring points, rather than expand the equation to its limit. Or "Attribute: synthesis_off" was used to factor a portion of an expression to a macrocell. Or the compiler flattened too many intermediate expressions into a single equation, requiring sum splitting.

Product Term Utilization Statistics

⇒ How Is This Useful?⇐

Sometimes, fixing the pinout of a design early in development will prevent the fitter from grouping like equations to take advantage of the architecture's ability to share Product Terms (PTs). In most cases, this is not a big sacrifice, but to regain optimal sharing, it is best to allow the fitter to establish the initial pinout. In some cases, the pinout may be manually changed to further improve PT sharing. Equations with greater numbers of product terms in common may be grouped. Or the product terms might be better utilized by spreading complex equations more evenly to other logic blocks, in effect distributing the burden, so no one logic block is significantly more congested.

Logic Block I/O

⇒ How Is This Useful?⇐

Inputs from the Programmable Interconnect Matrix (PIM) can be one of the limiting issues in design placement. Here are some suggestions for avoiding congestion at the inputs from the PIM. For signals that are less time critical, create factoring points early in the signal paths of complex equations. In theses cases, the implementation will pass a single signal rather than a group of inputs between logic blocks. Or, spread complex equations more evenly to other logic blocks, in effect, distributing the burden, so no one logic block is significantly more congested.

Macrocell Utilization (Overall Design Statistics)

⇒ How Is This Useful?⇐

"Macrocells Used" is one of the limitations on designs targeted to a specific architecture that determines the size of part required to implement a design. It is the most common measure of the efficiency of a design, the capabilities of the compiler, and the routability of the target architecture. Because of this, it is very useful for benchmarking synthesis tools and devices.

Preset/Reset and Output Enable Combinations

⇒ How Is This Useful?⇐

The greater the number of unique groupings of AP, AR, CLK, and OE requirements in a design, the more initial restrictions are placed on the fitter, and so the more difficult the final placement process will be. In some cases, macrocells may be rendered unusable as nodes where these resources are used.

1) It is generally best to use synchronous resets, synchronous presets, and global clocks wherever possible.

2) It is generally best to include all dependencies for registered signals in the combinatorial decode preceding the register, rather than using logic to gate its clock.

3) It is also usually best to allow the fitter to establish the initial pinout for the design. This allows the fitter to group signals that use similar sets of global resources.

6. JEDEC File Generation

The ultimate result of the compiler is the data for programming the target device. The data is generated by the fitter and output in the JEDEC file format.

⇒ How Is This Useful?⇐

If *Warp* is not creating a JEDEC file when one is expected, verify that the top-level file has been marked and is being compiled in a device project.

7. Static Timing Path Analysis

This section provides a detailed description of the delays associated with each signal. This information is very useful for verifying signals that were sum split, for identifying signals that were factored, or even for finding errors in the source code. At the end of this section, an analysis is provided which gives the worst case timing path. This may help determine the maximum clock speed of the final implementation of the design.

⇒ How Is This Useful?⇐

The "Timing Path Analysis" section is very useful for determining how well the design compiled into the target device. To get the maximum performance out of the CPLD, it is usually best to minimize the number of passes through the array for each expression. The "Design Equations" section is the first useful place to look for multiple passes, and the "Timing Path Analysis" section is the other. The signals containing multiple passes may be identified in this section, and cross-referenced and researched using the equations.

Detailed Compiler and Report File Descriptions

Opening Sequence

The first information provided in the report file includes the language being synthesized, the revision of the software and internal build number, and copyright information. This is followed by the name of the source file, and the compile options selected, presented in a command line format (*Figure 1*).

1. Parser

The first operation is parsing. The parser verifies correct language syntax (grammar), some semantics (usage), and correct instantiations of library components. It also performs "Module Generation" (described in *Data Path Operators*) during this first step. VHDLFE.EXE performs this operation for VHDL source files and VLOGFE.EXE performs this operation for Verilog source files. The output is *filename*.PRS and this file is deleted after it is passed on to the next step. See *Figure 2*.

External References

The parser calls the library for the target part, and then any libraries specified in the source files, such as the IEEE and

```
    | | | | | | |
  _______________
 -|               |-
 -|               |-
 -|               |-
 -|    CYPRESS    |-
 -|               |-
 -|               |-   Warp VHDL Synthesis Compiler: Version 5.2 IR 17
 -|               |-   Copyright (C) 1991, 1992, 1993,
  |_______________|    1994, 1995, 1996, 1997, 1998, 1999 Cypress Semiconductor
    | | | | | | |
======================================================================
Compiling:  PCI_tgt.vhd
Options:    -m -yu -e10 -w100 -o2 -ygs -fO -fP -v10 -dc37192 -pCY37192P160-154AC -u PCI-Target.hie
PCI_tgt.vhd
======================================================================
```

Figure 1. Opening Sequence

```
vhdlfe V5.2 IR 17:  VHDL parser
Tue Sep 28 12:38:24 1999

Library 'work' => directory 'lc37192'
Linking 'C:\warp\bin\std.vhd'.
Linking 'C:\warp\lib\common\cypress.vhd'.
Linking 'C:\warp\lib\common\work\cypress.vif'.
Library 'ieee' => directory 'C:\warp\lib\ieee\work'
Linking 'C:\warp\lib\ieee\work\stdlogic.vif'.
Linking 'C:\warp\lib\common\stdlogic\lpmpkg.vif'.
Linking 'C:\warp\lib\common\stdlogic\rtlpkg.vif'.
Linking 'C:\warp\lib\common\stdlogic\mod_cnst.vif'.
Linking 'C:\warp\lib\common\stdlogic\mod_mth.vif'.
Linking 'C:\warp\lib\common\stdlogic\mod_gen.vif'.
Linking 'D:\MyDesigns\PCI\lc37192\pci_pkg.vif'.
PCI_tgt.vhd (line 138, col 87):  Note: Substituting module 'cmp_vv_ss' for '='.
PCI_tgt.vhd (line 195, col 66):  Note: Substituting module 'cmp_vv_ss' for '='.
Linking 'D:\MyDesigns\PCI\lc37192\config.vif'.
Linking 'D:\MyDesigns\PCI\lc37192\pargen.vif'.

vhdlfe:  No errors.
```

Figure 2. Parser

WORK libraries. It then links the corresponding packages, plus any packages created by the user.

Data Path Operators

The parser also performs “Module Generation”. That is, the parser recognizes VHDL or Verilog coding that performs a common function like an adder or comparator. These functions are referred to as “data path operators”. It then substitutes pre-optimized code for that function. These optimized versions of common functions are stored in lower level “VIF” format (described in “High Level Synthesis”). For each optimized function, the library contains a version optimized specifically for each Cypress PLD architecture. Beyond that, the library contains two sets of these, one optimized for speed (using the shortest timing path through the PLD), and one optimized for area (using the least number of macrocell and other resources). The substituted high-level code is identified by its source file name, line and column number, and the statement, “Note: Substituting module…” (see *Figure 2*).

2. High-Level Synthesis

The next step is high-level synthesis. This step reduces the high-level VHDL or Verilog to an intermediate format called an expression tree. It also optimizes out the I/O or entity portion of the unused logic. This step is performed by TOVIF.EXE. The results are stored in *filename*.VIF. VIF stands for VHDL/Verilog Intermediate Format. Once created, these files are maintained as libraries in the *Warp* directories, and under the working directory in each target subdirectory. (See *Figure 3*.)

```
tovif V5.2 IR 17:  High-level synthesis
Tue Sep 28 12:38:25 1999

Linking 'C:\warp\bin\std.vhd'.
Linking 'C:\warp\lib\common\cypress.vhd'.
Linking 'C:\warp\lib\common\work\cypress.vif'.
Linking 'C:\warp\lib\ieee\work\stdlogic.vif'.
Linking 'C:\warp\lib\common\stdlogic\lpmpkg.vif'.
Linking 'C:\warp\lib\common\stdlogic\rtlpkg.vif'.
Linking 'C:\warp\lib\common\stdlogic\mod_cnst.vif'
Linking 'C:\warp\lib\common\stdlogic\mod_mth.vif'.
Linking 'C:\warp\lib\common\stdlogic\mod_gen.vif'.
Linking 'D:\MyDesigns\PCI\lc37192\pci_pkg.vif'.
Linking 'D:\MyDesigns\PCI\lc37192\config.vif'.
Linking 'D:\MyDesigns\PCI\lc37192\pargen.vif'.
Linking 'C:\warp\lib\common\stdlogic\mcompare.vif'

tovif:  No errors.
```

Figure 3. High-Level Synthesis

External References

This step links the same libraries and packages as the previous step. It’s clear by the filename that VIF is the file format used to link the packages.

Removing Wires

If “Detailed Report File” is selected in the Compiler Options menu, the report file may have references to removing wires. This is related to instantiations from the Module Generation, LPM, or user libraries. Some libraries are written so each component implements the most comprehensive version of its function possible. If the specific application requires a simpler version of a library component, the optimizer will optimize out the unused portion of the function. TOVIF only optimizes out the I/O or entity portion of the unused logic; it does not remove the unused logic itself.

3. Synthesis and Optimization

Synthesis and optimization are performed by TOPLD.EXE. There are two processes within TOPLD: synthesis and optimization. The synthesis portion performs state machine synthesis. The optimization portion optimizes through substitution, removes unused logic at the gate or architecture level, and performs initial Boolean-level logic reduction. This step runs only when the top-level file in a device project is compiled. The output file is *filename*.PLA. This file is deleted after it is passed on to the following step (see *Figure 4*).

External References

The libraries are called and the packages are linked as before.

Synthesis

There are two processes within TOPLD: synthesis and optimization. The synthesis portion performs state machine synthesis.

```
topld V5.2 IR 17:  Synthesis and optimization
Tue Sep 28 12:38:27 1999

Linking 'C:\warp\bin\std.vhd'.
Linking 'C:\warp\lib\common\cypress.vhd'.
Linking 'C:\warp\lib\common\work\cypress.vif'.
Linking 'C:\warp\lib\ieee\work\stdlogic.vif'.
Linking 'C:\warp\lib\common\stdlogic\lpmpkg.vif'.
Linking 'C:\warp\lib\common\stdlogic\rtlpkg.vif'.
Linking 'C:\warp\lib\common\stdlogic\mod_cnst.vif'
Linking 'C:\warp\lib\common\stdlogic\mod_mth.vif'
Linking 'C:\warp\lib\common\stdlogic\mod_gen.vif'
Linking 'D:\MyDesigns\PCI\lc37192\pci_pkg.vif'.
Linking 'D:\MyDesigns\PCI\lc37192\config.vif'.
Linking 'D:\MyDesigns\PCI\lc37192\pargen.vif'.
Linking 'C:\warp\lib\common\stdlogic\mcompare.vif'
State variable 'tstate' is represented by a
  Bit_vector (0 to 1).
State encoding (sequential) for 'tstate' is:
       idle :=        b"00";
       cmp := b"01";
       tdata :=       b"10";
Linking 'C:\warp\lib\lc370\stdlogic\c370.vif'.
```

Figure 4. Synthesis and Optimization

State Variable

State Machine Encoding

In this section the state names are mapped into the actual state bit values. Here the report shows which format was used for encoding, such as sequential, Grey, one-hot-one, or one-hot-zero (see *Figure 4*). The state machine format may be specified in VHDL using “Attribute: state_encoding” (see Chapter 6 of the Users Guide). State machine encoding is only supported in VHDL. IEEE-1363 Verilog does not support any type of symbolic encoding scheme; the user is required to explicitly encode states.

```
-----------------------------------------------------
Detecting unused logic.
-----------------------------------------------------
    User names
        mem_readwrite


Deleted 1 User equation or component.
Deleted 1 Synthesized equation or component.

-----------------------------------------------------------
Alias Detection
-----------------------------------------------------------


-----------------------------------------------------------
Aliased 0 equations, 226 wires.
-----------------------------------------------------------

-------------------------------------------------------------
Circuit simplification
-------------------------------------------------------------


-------------------------------------------------------------
Circuit simplification results:

        Expanded 55 signals.
        Turned 0 signals into soft nodes.
        Maximum expansion cost was set at 10.
-------------------------------------------------------------

-----------------------------------------------------
Alias Detection
-----------------------------------------------------


-----------------------------------------------------
Aliased 0 equations, 0 wires.
-----------------------------------------------------
Created 733 PLD nodes.

topld:  No errors.
```

Figure 5. Optimization

Optimization

There are two processes within TOPLD: synthesis and optimization. The optimization portion optimizes through substitution, removes unused logic at the gate or architecture level, and performs initial Boolean-level logic reduction (see *Figure 5*).

Detecting Unused Logic

This process is announced with a banner in the report file. If "Detailed Report File" is selected in the Compiler Options menu, the report file may have references to logic that was removed. Any logic that is defined by the source file, but has no destination for its output, is removed here. This also applies to instantiated components. Some libraries, like the LPM library, are written so each component implements the most comprehensive version of its function possible. If the specific application requires a simpler version of a library component, the optimizer will optimize out the unused portion of the logic. A common example of this function is the removal of adder logic when a vector is added to a constant.

Alias Detection

This process is announced with a banner in the report file. If "Detailed Report File" is selected in the Compiler Options menu, the report file may have references to aliasing and removing wires. In this case, nodes that contain no logic, or nodes with two separate names are factored out of the low-level equations. For example, when library components are used, the port map connects signals defined in the library

(in the component declaration) to signals in the file where the component is instantiated. These signal pairs are factored to one signal in this section. This also applies to Module Generation. There may be references to removing the "Lhs" (left-hand side) or "Rhs" (right-hand side) of aliased equations. Every signal in the source files and libraries used in the design is given a unique number identifier, shown in brackets. The number identifier is related to the signal's position in the design hierarchy - the signals are generally numbered from the top down. For aliased signal names, TOPLD removes the signal with the numerically higher identifier (lower in the hierarchy), and indicates whether it removed the Rhs or the Lhs. The results of this process are given in totals at the end of this section.

Circuit Simplification (Virtual Substitution)

This process is announced with a banner in the report file. If "Detailed Report File" is selected in the Compiler Options menu, the report file may have references to "Substituting". If an equation is dependent on the combinatorial output of another equation, the second equation is substituted into the first in place of its output signal name. After each substitution is listed, TOPLD also shows an associated "cost". This is a raw estimate of the additional product terms required to implement the substitution (before Boolean reduction or inversion is applied). The "cost" is an estimate of the cumulative effect of substituting a single equation into every location where its signal name is used. This number is used to quantify the design trade-off, in terms of the number of product terms it "cost" to attempt to reduce the number of macrocells used.

TOPLD makes several passes for this reduction technique to flatten as many levels as possible. The lengths to which TOPLD will go to flatten equations can be controlled by the "Node Cost" setting in the Compiler Options menu. Node Cost may be set to 0 through 11, and limits both the cumulative cost and the maximum cost to any individual equation where a given signal is substituted. This maximum value is referred to as "size". The actual costs and sizes associated with each setting are established according to proprietary rules, and are different for various architectures.

The line in *Figure 6* shows that the substitution violated the maximum cost of 90 with an estimated cost of 128 (before reduction), but did not violate the maximum size of 102 with an estimated size of 64. It therefore did not perform the substitution, and made the signal into a soft node (described in "*Circuit Simplification Results*").

For a more information on controlling substitution, see Chapter 6 of the Users Guide.

Circuit Simplification Results

Circuit simplification results are expressed in totals at the end of this section (*Figure 5*). Expanded signals are equations where substitution was used. The number of soft nodes generated is displayed. Soft nodes are factoring points that are earmarked for conversion into physical nodes during the fitting process. The "Node Cost" setting from the Compiler Options menu is displayed. The total number of nodes is shown. Then, if "Detailed Report File" is selected in the Compiler Options menu, the nodes that were substituted into the primary, larger equations are listed as they are removed.

4. Technology Mapping and Optimization

This step is performed by two files: DSGNOPT.EXE and MINOPT.EXE. DSGNOPT optimizes the logic structures for the target architecture while MINOPT is called as a Boolean reduction engine. DSGNOPT chooses flip-flop type and equation polarity according the options available in the target PLD to minimize product term usage. It also performs sum-splitting and global resource reduction, and finally converts all the equations into a format appropriate to the target architecture. Depending on the complexity of the design, these files may run several times in succession, always beginning and ending with DSGNOPT. This step runs only when the top-level file in a device project is compiled. They generate several temporary files named *filename*.FD*n* (see *Figure 7*).

```
Note: Virtual signal MODULE_2_g1_a0_gx_u0_eq_13 with ( cost: 128 or cost_inv: 64)  > 90 or
with size: 64 > 102 has been made a (soft) node.
```

Figure 6. Node Cost

```
PLD Optimizer Software:       DSGNOPT.EXE    02/APR/1999  [v4.02 ] 5.2 IR 17

DESIGN HEADER INFORMATION  (12:38:38)

Input File(s): PCI_tgt.pla
Device       : CY37192P160
Package      : CY37192P160-154AC
ReportFile   : PCI_tgt.rpt

Program Controls:
    COMMAND PROPERTY BUS_HOLD ENABLE

Signal Requests:
    GROUP DT-OPT ALL
    GROUP FAST_SLEW ALL

Completed Successfully
```

Figure 7. Technology Mapping and Optimization

Source, Target, and Options

The header first lists the software function, name, software revision date, software version number, and the compile time of day. It then lists the input file name, the target device and the report file name. Finally, it lists the execution options and some of the compiler options.

Process Virtual … Expanded

In this section equations are flattened or expanded. If an equation is dependent on the combinatorial output of another equation, the second equation is substituted into the first in place of its output signal name. For most designs compiled in *Warp*, however, the substitution is done earlier (by TOPLD), and this process only removes redundant buffers and wires.

Process Virtual … Converted to NODE

When expansion has been done to its limit, the resulting soft nodes, registered equations, outputs, and signals on which "Attribute: synthesis_off" was used are committed to macrocell or input register resources.

Generating Both D & T Register Equations

The number of product terms used to implement XOR functions can sometimes be reduced by using a T flip-flop. For example, the n^{th} bit of a simple counter implemented with a D flip-flop requires n product terms. For most CPLDs, that's clearly a problem for any counter wider than 16 bits. However, a simple counter that uses T flip-flops only uses one product term per bit. The sum-of-products (for D flip-flops) and T flip-flop version of each signal is generated here to be evaluated and optimized later.

Expanding XOR Equation

XOR equations are converted into sum-of-products format for use with T flip-flops.

Optimizing Logic Without Changing Polarity

DSGNOPT identifies the equations that are to be optimized by MINOPT without inversion. For example, equations that were expanded to their XOR equivalent (for use with T flip-flops) cannot use output polarity inversion. All registered signals can be forced into this category by selecting "T" flip-flop or "Keep Polarity" in the Compiler Options menu (excluding signals created in a sum-split). On subsequent passes, other signal types may appear in this list, once their optimal polarity is identified.

Optimizing Logic Using Best Output Polarity

Since the logical inversion of both sides of an equation trades sums for products and products for sums, in some cases implementing the inverse of an equation can require fewer product terms. For example, an equation like Q = A + B + C requires three product terms. The inverse is /Q = /A * /B * /C. This only requires a single product term and an inversion of Q. DSGNOPT identifies the equations for which MINOPT will optimize both the Sum-Of-Products (SOP) and Product-Of-Sums (POS) versions. All combinatorial and D registered signals will be evaluated this way by default. All signals can be forced into this category by selecting "D" flip-flop in the Compiler Options menu. All signals can be excluded from this evaluation by selecting "Keep Polarity" in the Compiler Options menu (except for signals created in a sum split).

Selected Logic Optimization OFF

DSGNOPT prevents further logic optimization for equations that have already been committed to macrocells. This includes registered equations, and equations that have been forced to terminate in a macrocell by "Attribute: synthesis_off".

Optimizing Banked Preset/Reset Requirements

Depending on the features available in the targeted device, DSGNOPT attempts to optimize the resources used for global functions like clock, output enable, and asynchronous reset and preset.

Sum-Splitting Output Logic

For equations that require more than the maximum number of product terms provided to a macrocell, the final sum or OR function must be split between two or more macrocells. This can be done in "cascaded" or "balanced" format, and the format that is used can be controlled for each signal using "Attribute: sum_split" (see Chapter 6 of the Users Guide). Balanced is the default format.

Selecting T/D Register Equation as Minimal

DSGNOPT individually selects the minimal implementation of each signal after generating inputs to a T flip-flop and both polarities for input to a D flip-flop.

Inverting Preset/Reset & Output Logic Polarity

In a case where the control signal for AR for one macrocell is also used as the control signal for AP for another, and vice versa, the control signals and output on one of them are inverted and the control signals are reversed.

Folding Logic

Where the expression for OE control is more complex than the target architecture will accommodate, the compiler will implement the expression as a combinatorial pass through the CPLD. It then uses that output as a single-term input into the OE control logic. This is referred to as folding.

5. Fitting

This compilation step assigns the final equations and functions to physical hardware locations in the target device. This step runs only when the top-level file in a device project is compiled. The resulting file will be *filename*.JED. The fitter that is actually invoked depends on the type of device that is targeted. The Ultra37000 family invokes 37KFIT.EXE.

The most useful information in the report file is provided during this step. The final equations are listed here, in Boolean format (see *Figure 8*). Next, a graphical representation of the product term array placement map is provided. Next, it provides tables on logic block utilization, and the pinout for each logic block. Finally, it shows the total usage of the part for every logic block and for global resources.

⇒ How Is This Useful?⇐

The "Design Equations" section is a good beginning place to verify that the design written in HDL eventually compiled to functions in Boolean algebra that resemble the designer's original intent. Ultimately, functional simulation will verify the design, but a cursory look at the equations at this level can identify typos that had correct syntax, but that created obviously incorrect functions.

```
PLD Compiler Software:        C37KFIT.EXE    02/APR/1999  [v4.02 ] 5.1.2 IR 06

DESIGN EQUATIONS           (09:40:54)

    tstateSBV_1.D =
          /tstateSBV_1.Q * /tstateSBV_0.Q * /framen * frame_ni.Q
```

Figure 8. Design Equations

Signal Name Conventions

First, the normal rules for signal names in VHDL and Verilog apply. In the simple case, signals are assigned the names they are given in the top-level source file. Where instantiations with buried functions are used, the buried node signal name begins with the label name of the instantiation. If there are several levels of hierarchy, the label names are concatenated to the signal name, separated by underscores. At the final substitution, the signal name in the final component is concatenated to the end, separated by an underscore. These rules apply to Module Generation (described in *Data Path Operators*). In the case of Module Generation, buried node signal names begin with "MODULE_*n*_", and intermediate module outputs that cannot be assigned to source file signal names begin with "MODIN*n*_". Next, bus order subscripts are concatenated as digits following an underscore at the end of the signal name. (In some circumstances, signals in the top-level file may maintain the parentheses for their bus order subscripts.) When there are several processes with identical variable names, where the variables terminate in nodes, TOPLD will append the label "wrp*n*", to every occurrence of the variable name after the first. If DSGNOPT must perform a sum-split, the additional nodes are named "S_*n*". Similarly, where logic is folded, the additional nodes are named "F_*n*". In the above cases, *n* is a number assigned in the order the signals or nodes are created.

Signal Name Extensions

Macrocells can have several features, such as output enable, asynchronous reset, etc. All the equations associated with a macrocell use the same signal name for macrocell feature assignments. Assignments for each feature are identified with a one, two, or three character extension to the signal name. *Table 1* describes the extensions that are used to identify equations for various macrocell features.

Equation Conventions

The final equations resulting from compilation and minimization are presented in Boolean format. The following symbols are used in the Boolean expressions:

- / represents a logical inversion.
- * represents an AND function.
- + represents an OR function.
- = represents an assignment.

VCC and GND represent constant logic values 1 and 0 respectively.

Table 1. Signal Name Extensions

Extension	Meaning
	No extension means non-registered input or combinatorial output.
.C	Clock
.D	D-type flip-flop input
.T	T-type flip-flop input
.AP	Asynchronous Preset
.AR	Asynchronous Reset
.OE	Output Enable
.Q	Registered feedback from macrocell
.CMB	Combinatorial feedback from macrocell
.PIN	Feedback from pin
.LH	Transparent Latch Enable
.CI	Input Register Clock
.QI	Output from Input Register or Input Latch

⇒ How Is This Useful?⇐

The "Design Equations" section is very useful for determining how well the design compiled into the target device. To get the maximum performance out of the CPLD, it is usually best to minimize the number of passes through the array for each expression. The compiler will implement multiple passes for several reasons. A resulting equation may have been too complex to implement in the available product terms, so the compiler created factoring points, rather than expand the equation to its limit. Or "Attribute: synthesis_off" was used to factor a portion of an expression to a macrocell. Or the compiler flattened too many intermediate expressions into a single equation, again requiring sum splitting.

There are some techniques to help the compiler to implement a design with fewer passes. If the compiler is implementing several sum splits, the "Node Cost" setting in the Compiler Options menu can be a useful tool. By default it is set to one less than its maximum value. This implies that the compiler will work strenuously to reduce the number of macrocells, at the cost of additional product terms. If the product terms for a given signal exceed the maximum available to that macrocell, a sum split must be performed, thereby increasing the number of macrocells used for that signal. Experimenting with the Node Cost setting may yield better results.

Here's another technique. Let's use a common circuit as an example—a counter that resets at a specific value. The compiler would take the expression that decodes the terminal count and flatten it into the counter equations themselves. This might work fine for a small counter, but for a large counter, the high-order bits might require more than the maximum available product terms. Each bit would be sum split into two or more passes through the array and this would also consume additional macrocells. This situation can be improved by factoring the terminal count logic to its own macrocell using "Attribute: synthesis_off" (or various other methods discussed in Chapter 6 of the Users Guide). This would only require a single extra product term, but the counter is still dependent on two combinatorial passes through the array: first to decode the terminal count and second to decode the next counter value. The final step to improving this design would be to register the output of the terminal count logic, and decrease the value of the terminal count by 1. This way, the terminal count decode arrives after the appropriate clock, and there is only one pass through the array between clocks.

Generally, to reduce the number of passes through the array, look for final expressions that are too complex to flatten intermediate expressions to a single macrocell. Factoring intermediate expressions in these cases will potentially help reduce the number of macrocells used and the number of required passes. Wherever possible, register the factored expressions to minimize the remaining multiple passes.

Design Rule Check

This step performs an evaluation of the design against the target device to check for basic incompatibilities. These might include too many macrocells, or too many I/O pins required. In some cases, synthesis may be performed by a third-party CAE vendor and *Warp* would be used only for the fitting function. The fitter therefore evaluates the synthesized equations to verify correlation to the targeted architecture (see *Figure 9*).

Partition Logic Steps

This series of steps is the beginning of the actual fitting process. The header first lists the software function, name, software revision date, software version number, and the compile time of day. The series starts with fixed pin assignments and global resources, and builds the design through more flexible resource assignments. By the end of this series, signals are assigned to logic blocks, but are not yet assigned to specific macrocells. The documentation of these steps provides adequate descriptions (see *Figure 10*).

Design Signal Placement

The actual signal placement of fitting process is performed in several steps for each logic block consecutively (see *Figure 11*).

Fitting Signals to Logic Block

Pre-placed outputs are assigned to their macrocells within the logic block. The remaining equations are evaluated to determine where they might best fit in groups of macrocells.

Assigning Signals to Macrocells

Each unplaced signal is then preliminarily assigned to a macrocell.

Improving Macrocell Assignment

This step identifies identical product terms in equations assigned to neighboring macrocells. For this purpose, a "neighbor" would be a macrocell up to three away in either direction. It takes each identical product term, implements it only once in the product term allocator for a group of four adjacent macrocells, and sends that one resulting product term signal to any of the four macrocells where it is needed. This is a feature of the Ultra37000 architecture referred to as "Product Term Sharing".

Assigning Product Terms to Allocator

After the above final logic reduction, product terms are assigned to the product term allocator.

Combining

This is a process where buried nodes at I/O macrocells are "combined" with input signals. A buried node is one where a signal is assigned a macrocell but the resulting output is only used internally and is not routed to a pin. Some architectures have fixed buried macrocells. These are macrocells that are not wired to output pins and so are permanently buried. In some architectures, the macrocells that are associated with pins may be configured as buried. These macrocells still have their input pins available to be used as inputs. After all the I/O macrocells and buried macrocells are assigned, input signals are assigned to pins, and in some cases "combined" with buried macrocells.

Logic Block Placement Diagrams

For the Ultra37000 family, the fitter prints the details of the product term allocator, then the pin assignments for each logic block consecutively (see *Figure 12*).

```
PLD Compiler Software:        C37KFIT.EXE    02/APR/1999  [v4.02 ] 5.2 IR 17

DESIGN RULE CHECK          (09:45:16)

Messages:
                  None.

Summary:
                  Error Count = 0      Warning Count = 0

Completed Successfully
```

Figure 9. Design Rule Check

```
PLD Compiler Software:        C37KFIT.EXE    02/APR/1999  [v4.02 ] 5.2 IR 17
PARTITION LOGIC           (12:38:50)
Messages:
  Information: Initializing Logic Block structures.
  Information: Forming input seeds.
  Information: Checking for duplicate NODE logic.
  Information: Forming input seeds.
  Information: Assigning fixed logic to Logic Blocks.
  Information: Processing banked global preset, reset and output enable.
  Information: Separating output logic set to GND/VCC.
  Information: Validating Logic Block's with pre-placed signals.
  Information: Separating input register logic.
  Information: Assigning initializing equations to empty Logic Blocks.
  Information: Separating output combinatorial logic.
  Information: Separating disjoint output logic.
  Information: Separating output node logic.
  Information: Assigning floating outputs to Logic Blocks.
  Information: Compacting Logic Block interconnect.
  .+.........+........+.......+....+....+.............................
  Information: Separating disjoint output logic.
  Information: Separating output node logic.
  Information: Assigning floating outputs to Logic Blocks.
  Information: Assigning floating outputs to Logic Blocks.
  Information: Assigning floating outputs to Logic Blocks.
  Information: Minimizing interconnect between Logic Blocks.
  .++.++.++.+..+...+...+++..+.+.....+...+.+..+...........
Start=12:38:52  End=12:38:54
  Information: Assigning floating outputs to Logic Blocks.
  Information: Assigning floating inputs to Logic Blocks.
Summary:
                Error Count = 0      Warning Count = 0
Completed Successfully
```

Figure 10. Partition Logic Steps

```
PLD Compiler Software:        C37KFIT.EXE    02/APR/1999  [v4.02 ] 5.2 IR 17

DESIGN SIGNAL PLACEMENT    (12:38:54)

  Information: Fitting signals to Logic Block D.
  Information: Assigning Signals to Macrocells.
  Information: Improving Macrocell Assignment
  ..+.+..+.........................
  Information: Assigning Product Terms to Allocator
  Information: Assigning Signals to Macrocells.
  Information: Improving Macrocell Assignment
  ..............
  Information: Assigning Product Terms to Allocator
  Information: Combining 'cfgaddr_2' definition with input pin 'idsel'.
  Information: Combining 'frame_ni' definition with input pin 'rstn'.
…

Summary:
                Error Count = 0      Warning Count = 0

Completed Successfully
```

Figure 11. Design Signal Placement

```
PLD Compiler Software:        C37KFIT.EXE    02/APR/1999  [v4.02 ] 5.2 IR 17
LOGIC BLOCK D PLACEMENT   (12:38:54)
Messages:

_____________________________________________________________________________
          1111111111222222222233333333334444444444555555555566666666667777777777
01234567890123456789012345678901234567890123456789012345678901234567890123456789
_____________________________________________________________________________
| 0 |ad(1)
XXXXXXXXXXXXXX++.............................................................
| 1 |[i/p]
......++++++++++++++++.......................................................
| 2 |(cfgaddr_2)idsel
..........++++++XX++++++++...................................................
| 3 |(irdy_ni)
..............X++++++++++++++.................................................
| 4 |ad(4)
..................XXXXXXXXXXXX++++.........................................
| 5 |UNUSED
......................++++++++++++++++.....................................
| 6 |[i/p]
..........................++++++++++++++++..................................
| 7 |(cfgaddr_1)
..............................X+++X++++++++++..............................
| 8 |ad(3)
..................................XXXXXXXXXXXXX++++..........................
| 9 |UNUSED
....................................++++++++++++++++........................
|10 |ad(5)
........................................XX++X+XX+XXXXXXX.....................
|11 |(cfgaddr_3)
............................................XX++++++++++++++.................
|12 |(frame_ni)rstn
................................................X++++++++++++++..............
|13 |(cfgaddr_0)
....................................................+++++X++X+++++++.........
|14 |ad(2)
.....................................................XXXX+XXXX+XX++XX......
|15>|ben(0)
.............................................................+++X++XX++X+++++
_____________________________________________________________________________

Total count of outputs placed        =  12
Total count of unique Product Terms  =  64
Total Product Terms to be assigned   =  76
Max Product Terms used / available   =  71 /  80   = 88.76 %
```

Figure 12. Logic Block Placement

Product Term Allocation and Sharing

Each column represents an individual product term, and the product terms are numbered “0” through “79” (in vertical fashion) in the upper two rows of the diagram. The row groups below represent macrocells and those are numbered “0” through “15” down two columns on the far left side. In the body of the diagram, a “+” indicates a location where a product term can be assigned, but is not used. An “X” indicates a location where a product term is programmed connected. A row of Xs represents a group of product terms that are summed (ORed) to implement the final sum of products equation. The product term (or AND) array is not shown in the report file, as is usual with SPLDs. Two or more rows that show Xs directly vertical to each other indicate signals that are sharing the same product terms (as described in “*Improving Macrocell Assignment*”). The Xs that represent shared product terms are shown in **bold** in the figure above. Buried nodes are indicated by signal names in parentheses, while outputs assigned to pins use the signal name without parentheses. Buried nodes combined with inputs show the buried signal name in parentheses with the input signal name directly following it. Macrocell locations used just as inputs are indicated by “[i/p]”, “[In-

put Latch]" or "[Input Register]" without the input signal name. If a pin location is fixed in the source file, the fixed node will be indicated by a ">" next to the node number.

Product Term Utilization Statistics

In the statistics following the product term allocation diagram, the "Total count of outputs placed" shows the total number of macrocells used for buried nodes or outputs in a Logic Block. The "Total count of unique Product Terms" shows just what it describes. The "Total Product Terms to be assigned" shows the total number of Xs on the diagram, or the total number of Product Terms (PTs) from the original low-level equations which must be included in the final design. The "Max Product terms used…" shows the number of physical PTs required with the given constraints.

The difference between Total PTs and Unique PTs is the number of PTs that are candidates for sharing. The difference between the Total PTs to be assigned and Max PTs indicates the number of PTs that were successfully shared. The difference between the number of Unique PTs and the Max PTs is the number of PTs that were candidates for sharing, but had to be repeated or implemented more than once in the array. This final difference indicates that the design constraints were not optimal for sharing product terms. This may occur when pin locations are fixed or when pins using the same macrocell features are grouped.

"Max Product Terms used / available" shows the actual number of PTs implemented over the fixed number of PTs available in the logic block of the target architecture, and the final percent usage of those available PTs.

⇒ How Is This Useful?⇐

Sometimes, fixing the pinout of a design early in development will prevent the fitter from grouping like equations to take advantage of the architecture's ability to share PTs. In most cases, this is not a big sacrifice, but to regain optimal sharing, it is best to allow the fitter to establish the initial pinout. In some cases, the pinout can be manually changed to further improve PT sharing. Equations with greater numbers of PTs in common may be grouped. Or the PTs might be better utilized by spreading complex equations more evenly to other logic blocks, in effect, distributing the burden, so no one logic block is significantly more congested.

Logic Block Global Resource (Control Signal) Listing

For the logic block, where the global resources and product term controls are used, each global resource and product term control signal is listed with its associated input signal in a table format (see *Figure 13*).

Three columns of information are provided. The first column lists the global resource or the feature optionally controlled by a product term. The second column lists the logic polarity of the controlling signal, where there is an option at the logic block level. "AH" is used for active HIGH signals and "AL" for active LOW. The third column lists the control signal name. If a logic block feature does not use a product term for control, "<not used>" is substituted for the control signal name, and the polarity is programmed to the default value of active HIGH.

If the control signal can be implemented in a single product term, the control equation will be listed in the third column according to the rules described in the "**Equation Conventions"** section. If the control product term is derived from input signals, the signals will be listed in the equation. If the control product term is derived from an additional buried macrocell, the signal fed back from the buried macrocell is listed in parentheses. If the additional buried macrocell is combined with an input at its pin, or if one of the input signals used in the control equation is combined with a buried macrocell at its pin, *both* signal names may be listed in the control equation. In some cases, it may be difficult to determine whether the input signal or the buried macrocell signal is relevant in the control equation. This ambiguity may be resolved by referring to the "DESIGN EQUATIONS" section of the report file.

```
Control Signals for Logic Block D
---------------------------------
CLK pin  19 : clk
CLK pin  22 : <not used>
CLK pin  99 : <not used>
CLK pin 102 : <not used>
CLK PT      : AH : <not used>
PRESET      : AH : <not used>
RESET       : AH : <not used>
OE 0        : AL : (ad(31)_OE)w_fifo_fulln.CMB
OE 1        : AH : <not used>
OE 2        : AL : (ad(31)_OE)w_fifo_fulln.CMB
OE 3        : AH : <not used>
```

Figure 13. Control Signals

In this example, the first four signals are the global clocks from dedicated clock inputs. The polarity of the global clocks is programmable, but is not shown here, since it is not controlled at the logic block. The next signal is the product term clock specific to the logic block listed, including the chosen polarity. Preset, reset, and output enable controls available in the macrocell, with their polarities, are shown as well.

Logic Block I/O

The second diagram for each logic block is the "black box" view of the logic block (see *Figure 14*).

From PIM

The left side of the box shows the connections to the Programmable Interconnect Matrix.

To Macrocells and Pins

The right side of the box shows the macrocell and pin assignments. Signals listed inside the box, on the right side, are those assigned to fixed buried macrocells. These are macrocells that are not wired to outputs and so are permanently buried. Signals listed outside the box are those assigned to macrocells associated with specific pin numbers. The actual pin numbers are shown at the edge of the box next to the signal name. If the signal name is in parentheses, the macrocell is still considered a buried node. Buried nodes combined with inputs show the buried signal name in parentheses with the input signal name directly following it. The remaining input and output signal names are shown without parentheses. Bus order subscripts are also shown in parentheses next their signals.

Macrocell Utilization (Per Logic Block)

The logic block I/O diagram is followed by a short table showing the usage of the maximum or fixed resources available in the target architecture (see *Figure 15*).

```
Logic Block D
                  ____________________________________________
                 |   |= >ad(3).Q                        | 23|= ad(1)
                 |   |= >ad(3)                          |   |
                 |   |= >tstateSBV_0.Q                  |   |
                 |   |= >cfgaddr_2.Q                    |   |
                 |   |= >ad(5)                          | 24|= datain(6)
                 |   |= >cfg_read.Q                     |   |
                 |   |= >irdyn                          |   |
                 |   |= >r_fifo_empt..                  |   |
                 |   |= >tstateSBV_1.Q                  | 25|= (cfgaddr_2)idsel
                 |   |= >datain(2)                      |   |
                 |   |= >ad(2).Q            (irdy_ni) =|   |
                 |   |= >ad(2)                          |   |
                 |   |= >ben(0).Q                       | 26|= ad(4)
                 |   |= >datain(3)                      |   |
                 |   |= >cben(0)          not used:939 *|   |
                 |   |= >datain(5)                      |   |
                 |   |= >cfgaddr_1.Q                    | 27|= datain(12)
                 |   |= >ad(1).Q                        |   |
                 |   |= >mem_write.Q       (cfgaddr_1) =|   |
                 |   |= >ad(5).Q                        |   |
                 |   |= >cfgaddr_4.Q                    | 28|= ad(3)
                 |   |= >framen                         |   |
                 |   |> not used:401      not used:943 *|   |
                 |   |> not used:402                    |   |
                 |   |= >ad(4).Q                        | 29|= ad(5)
                 |   |= >cfg_write.Q                    |   |
                 |   |> not used:405       (cfgaddr_3) =|   |
                 |   |= >cfgaddr_0.Q                    |   |
                 |   |> not used:407                    | 30|= (frame_ni)rstn
                 |   |= >ad(4)                          |   |
                 |   |= >cfgaddr_5.Q       (cfgaddr_0) =|   |
                 |   |= >ad(31)_OE.CMB                  |   |
                 |   |= >datain(1)                      | 32|= ad(2)
                 |   |> not used:412                    |   |
                 |   |= >ad(1)                          |   |
                 |   |= >datain(4)                      |   |
                 |   |= >cfgmemacces..                  | 33|= ben(0)
                 |   |= >cfgaddr_3.Q                    |   |
                 |   |= >mem_read.Q                     |   |
                  ____________________________________________
```

Figure 14. Logic Block I/O

```
Information: Macrocell Utilization.

    Description        Used     Max
 ____________________________________
| I/O Macrocells     |   10  |   10  |
| Buried Macrocells  |    4  |    6  |
| PIM Input Connects |   34  |   36  |
 ____________________________________
                         48  /   52   = 92  %
```

Figure 15. Macrocell Utilization (per Logic Block)

In this table, “I/O Macrocells” refers to macrocells configured as buried nodes, macrocells used as outputs, and additional macrocells used as inputs where there is no buried node. “Buried Macrocells” refers to fixed buried macrocells, not including macrocells optionally configured as buried. “PIM Input Connects” refers to the number of connections between the Programmable Interconnect Matrix and the logic block.

⇒ How Is This Useful?⇐

Inputs from the Programmable Interconnect Matrix can be one of the limiting issues in design placement. Here are some suggestions for avoiding congestion at the inputs from the PIM. For signals that are less time critical, create factoring points early in the signal paths of complex equations. In theses cases, the implementation will pass a single signal rather than a group of inputs between logic blocks. Or, spread complex equations more evenly to other logic blocks, in affect, distributing the burden, so no one logic block is significantly more congested.

Combined Fitter Statistics

After the above information is provided for each logic block, the report file provides the combined statistics for the overall design.

Device Pinout

This section begins by identifying the target device and package type. It then lists the device pinout by consecutive package pin numbers. Buried nodes are shown with the signal name in parentheses. Buried nodes combined with inputs show the buried signal name in parentheses with the input signal name directly following it. The remaining input and output signal names are shown without parentheses. Power and ground pins are labeled as well (see *Figure 16*).

Pins for which “Attribute: pin_avoid” was used are labeled consecutively (in the order they are listed in the attribute statement) starting with “Reserved1”. “Attribute: pin_avoid” instructs the compiler to avoid using the specified pin as an input or output, but the compiler may still use the specified pin for a buried node. In this case, the buried node is shown with the signal name in parentheses, with the “Reserved*n*” label directly following it.

Macrocell Utilization (Overall Design Statistics)

This table shows the total usage of the maximum or fixed resources available in the targeted architecture. This is a sum of the previous tables for individual logic blocks, plus dedicated input pins, and additional PIM input connects for global resources (see *Figure 17*).

In this table, “Dedicated Inputs” refers to input-only pins that may not also be used as global clocks, and “Clocks/Inputs” refers to input-only pins that may be configured as either inputs or global clocks. “I/O Macrocells” refers to macrocells configured as buried nodes, macrocells used as outputs, and additional macrocells used as inputs where there is no buried node. “Buried Macrocells” refers to fixed buried macrocells (not including I/O macrocells optionally configured as buried). “PIM Input Connects” refers to the number of connections between the Programmable Interconnect Matrix and all the logic blocks.

Macrocell Utilization (Total I/O and Macrocell Resource Usage)

The final list totals the usage of various global resources (see *Figure 18*). Some of these quantities require detailed explanations.

“CLOCK/LATCH ENABLE signals” are the number of global clocks used (as macrocell clocks or macrocell latch enable). The Max equals the total clocks available, including product term clocks.

“Input REG/LATCH signals” are the number of input registers used. The Max is the sum of the input registers at dedicated inputs, and the number of fixed buried macro-

```
PLD Compiler Software:        C37KFIT.EXE    02/APR/1999  [v4.02 ] 5.2 IR 17

PINOUT INFORMATION   (12:38:54)

Device:  CY37192P160
Package: CY37192P160-154AC

…

                  23  :  ad(1)
                  24  :  datain(6)
                  25  :  (cfgaddr_2)idsel
                  26  :  ad(4)
                  27  :  datain(12)
                  28  :  ad(3)
                  29  :  ad(5)
                  30  :  (frame_ni)rstn
                  31  :  GND
                  32  :  ad(2)
                  33  :  ben(0)
```

Figure 16. Device Pinout

```
PLD Compiler Software:        C37KFIT.EXE    02/APR/1999  [v4.02 ] 5.2 IR 17

RESOURCE UTILIZATION (12:38:54)


  Information: Macrocell Utilization.

                    Description        Used     Max
                  ___________________________________
                  | Dedicated Inputs   |    1  |    1  |
                  | Clock/Inputs       |    4  |    4  |
                  | I/O Macrocells     |  120  |  120  |
                  | Buried Macrocells  |   26  |   72  |
                  | PIM Input Connects |  370  |  468  |
                  ___________________________________

                                       521  /  665   = 78  %
```

Figure 17. Macrocell Utilization (Overall Design Statistics)

```
                          Required    Max (Available)
CLOCK/LATCH ENABLE signals     1           16
Input REG/LATCH signals        0           77
Input PIN signals              5            5
Input PINs using I/O cells    13           13
Output PIN signals           107          107


Total PIN signals            125          125
Macrocells Used              133          192
Unique Product Terms         702          960
```

Figure 18. Macrocell Utilization (Total I/O and Macrocell Resource Usage)

cells next to I/O macrocells that may be used as input registers.

"Input PIN signals" are the number of dedicated inputs used (or reserved using "Attribute: pin_avoid"), not including global clocks or dedicated inputs that use input registers. The Max is calculated starting with the total dedicated inputs, minus the number of global clocks used, minus the number of dedicated inputs configured to use input registers.

"Input PINs using I/O cells" are the number of I/O pins used as inputs (or reserved using "Attribute: pin_avoid") where the underlying macrocell is not used as a buried node. The Max is calculated assuming the number used is the maximum available for that purpose (see the next category, "Output PIN signals").

"Output PIN signals" are the I/O pins used for outputs or optionally buried macrocells. The Max is calculated as the total number of I/O pins, minus the number of I/O pins used as inputs where the underlying macrocell is not used as a buried node (or "Input PINs using I/O cells").

"Total PIN signals" are the total number of pins consumed. This includes outputs, buried nodes at I/O pins, inputs at I/O macrocells (not including inputs combined with buried nodes), pins reserved using "Attribute: pin_avoid" (not including those combined with buried nodes), dedicated inputs, and clocks. The Max is the number of I/O macrocells plus the number of dedicated inputs or clocks. Both the Required and the Max numbers in this category are the sum of the three preceding (pin-related) categories.

"Macrocells Used" refers to the total number of output and buried macrocells required to implement the design.

⇒ *How Is This Useful?* ⇐

"Macrocells Used" is one of the limitations on designs targeted to a specific architecture that determines the size of part required to implement a design. It is the most common measure of the efficiency of a design, the capabilities of the compiler, and the routability of the target architecture. Because of this, it is very useful for benchmarking synthesis tools and devices.

"Unique Product Terms" refers to the total number of unique PTs discovered in each logic block group. It is not necessarily the number of unique product terms in the overall design; unique product terms that are repeated in separate logic blocks are counted for each occurrence. It also is not indicative of the number of PTs implemented, since occasionally PTs that are candidates for sharing can not actually be shared to all locations where they are needed (as discussed in the "Improving Macrocell Assignment" section). The Max is 80 times the number of logic blocks in the target device. That is, the number of PTs from the

Product Term Allocator used for node logic, excluding those used for OE, AR, etc., times the number of logic blocks.

Preset/Reset and Output Enable Combinations

This section describes groups of signals that have identical combinations of asynchronous preset, asynchronous reset, asynchronous clock, and output enable requirements. Each grouping in the report file represents a unique combination of requirements, and the report file shows the number of macrocells that share each unique combination - as if each grouping represented a single bubble in a Venn Diagram of requirements.

```
PRESET: GND
RESET : /(frame_ni)rstn
CLOCK PT : NULL
Used by Logic Blocks: NONE
Total unique inputs =  77
count of output equations =  31
==>OE: GND or VCC
   count of OE equations =  30
==>OE: perr_oe.Q
   Used by Logic Blocks:
   count of OE equations =  1
```

Figure 19. Preset/Reset and Output Enable Combinations

Resources for AP, AR, CLK, and OE requirements are generally distributed one or two per group of eight or sixteen macrocells. Because the resources for these functions are scarcer than the resources provided for logic implementation, these requirements take precedence during the placement process.

⇒ How Is This Useful?⇐

The greater the number of unique groupings of AP, AR, CLK, and OE requirements in a design, the more initial restrictions are placed on the fitter, and so the more difficult the final placement process will be. In some cases, macrocells may be rendered unusable as nodes where these resources are heavily used.

1) It is generally best to use synchronous resets, synchronous presets, and global clocks wherever possible.

2) It is generally best to include all dependencies for registered signals in the combinatorial decode preceding the register, rather than using logic to gate its clock.

3) It is also usually best to allow the fitter to establish the initial pinout for the design. This allows the fitter to group signals that use similar sets of global resources.

6. JEDEC File Generation

The ultimate result of the compiler is the data for programming the target device. The data is generated by the fitter and output in the JEDEC file format. The file is created for each logic block consecutively, and the logic blocks are listed in the report file as the fitter begins work on each. The fitter also announces completion of the JEDEC file.

⇒ How Is This Useful?⇐

If *Warp* is not creating a JEDEC file when one is expected, verify that the top-level file has been marked and is being compiled in a device project.

7. Static Timing Path Analysis

This section provides a detailed description of the delays associated with each signal. This information is very useful for verifying signals that were sum split, for identifying signals that were factored, or even for finding errors in the source code. At the end of this section, an analysis is provided which gives the worst-case timing path. This may help determine the maximum clock speed of the final implementation of the design. This step runs only when the top-level file in a device project is compiled. The fitter that is actually invoked depends on the type of device that is targeted. The Ultra37000 family invokes 37KFIT.EXE.

Relevant Timing for All Signals

All output signals are listed in this section, with their input dependencies and output delays. The signals are sorted by pin number, and each is given a row group of text separated by a line of dashes. The row group for each signal starts with the signal type - registered or combinatorial - followed by the signal name, and pin number shown in brackets. As in the pinout section, buried nodes are shown with the signal name in parentheses. Buried nodes combined with inputs show the buried signal name in parentheses with the input signal name directly following it. In this case, the timing information is of course related to the buried node signal. Below the signal name and location, one or more of the input signals are listed by name, with their associated delays. The four-dash arrows indicate input or intermediate signals that contribute to the delay listed. The path can actually be traced in the equation section, using the consecutively listed signal names. Then for registered outputs, the output signal name and output delays are listed (see *Figure 20*).

Combinatorial Outputs

For combinatorial outputs, input delay times are listed. This could include t_{PD} from a combinatorial source, or t_{CO} from a registered source. If an output enable is used, t_{ER} will be listed as a potential input delay.

Registered Outputs

For registered outputs, input delay times are listed and a clock to output time is listed. Input set up times could include t_S from a combinatorial source, or t_{SCS} from a registered source. If an output enable, asynchronous preset, or asynchronous reset is used, t_{ER}, t_{PO}, or t_{RO} will be listed as potential input delays. The clock to output time listed is t_{CO}.

Additional Passes

If an equation can be implemented with a single pass through the array, the input delay times in the report file will be equal to those values listed in the data sheet for the targeted device and speed grade. In this case, the report file will state “1 pass” next to the input signal. If an equation requires more than one normal pass through the array, inserting another combinatorial decode onto the propagation delay or between clocks, the input delay value listed in the report file will be calculated in the following way. The input delay of the first pass will be equal to the data sheet input delay as above, and every additional pass will add an internal propagation delay called a “one pass adder”. Likewise, if an equation requires a combinatorial de-

```
------------------------------------------------------------------------------
Signal Name | Delay Type  |   tmax   | Path Description
------------------------------------------------------------------------------
…
------------------------------------------------------------------------------
reg::par[143]
inp::par
              tS              5.0 ns     1 pass
inp::ad(20).Q
---->parmap_p7
---->parmap_p8
---->parmap_p9
              tSCS            20.0 ns    4 passes
out::par
              tCO             4.5 ns
inp::par_oe.Q
              tER             12.5 ns    1 pass
------------------------------------------------------------------------------
cmb::r_fifo_enn[144]
inp::r_fifo_emptyn
              tPD             7.5 ns     1 pass
inp::MODIN1_23.Q
              tCO             13.5 ns    2 passes
------------------------------------------------------------------------------
…
------------------------------------------------------------------------------
reg::ben(1)[158]
inp::framen
---->framen
              tS              5.0 ns     1 pass
inp::tstateSBV_1.Q
              tSCS            6.5 ns     1 pass
out::ben(1)
              tCO             4.5 ns
------------------------------------------------------------------------------
```

Figure 20. Relevant Timing for All Signals

code after a register, the t_{CO} value listed will be calculated the same way. The "one pass adder" takes approximately the same path as a combinatorial input signal, but excludes the delays of the input and output buffers. The value of the "one pass adder" is approximated by the compiler and is not listed in the data sheet, but is approximately equal to ($t_{CO2} - t_{CO}$).

⇒ How Is This Useful?⇐

The "Timing Path Analysis" section is very useful for determining how well the design compiled into the target device. To get the maximum performance out of the CPLD, it is usually best to minimize the number of passes through the array for each expression. The "Design Equations" section is the first useful place to look for multiple passes, and the "Timing Path Analysis" section is the other. The signals containing multiple passes may be identified in this section, and cross-referenced and researched using the equations. Please see the detailed suggestions in the "Design Equations" section.

Worst Case Timing Path Summary

The compiler finds the worst case path for combinatorial and registered outputs, and provides one example for each of the worst cases for t_{PD}, t_S, t_{SCS}, t_{CO}, t_{ER}, t_{AP} , and t_{AR} where they apply. From this list, the worst-case maximum clock frequency is equal to ($1/t_{SCS}$) (see *Figure 21*).

Only one worst-case is reported for each of the clock-related parameters, regardless of how many clocks are used in the overall design. If there is more than one clock in the design, the estimated maximum frequencies for the remaining clocks are equal to or higher than the one that is reported. The maximum frequencies for the remaining clocks can be determined by manually reviewing the maximum frequency (minimum t_{SCS}) of each signal that uses the clock in question.

```
Worst Case Path Summary
-----------------------

                    tPD = 7.5 ns for r_fifo_enn
                     tS = 5.0 ns for ad(1).D
                   tSCS = 20.0 ns for par.D
                    tCO = 13.5 ns for r_fifo_enn
                    tRO = 11.5 ns for MODIN1_31.AR
                    tER = 17.0 ns for ad(1).OE



Summary:
                Error Count = 0      Warning Count = 0

Completed Successfully
```

Figure 21. Worst Case Timing Path Summary