



Application Note

M1 Constraints Guide

April 1997

Version M1.2

Summary

This guide will help you convert your existing designs from previous versions of XACT^{step} 5.X to XACT^{step} M1.X software.

Xilinx Families

XC4000E/L, XC4000EX, XC4000XL

Table of Contents

Introduction	3
Converting a Logical Design into a Physical Design	7
Efficient Use of Timespecs	9
Appendix A: Table of M1 supported Constraints	13
Appendix B: Constraining LogiBLOX RAM/ROM with the Synopsys M1 Design Flows	16

Introduction

The Use of Layout and Timing Constraints in M1

In order to offer the most robust implementation tools in the programmable logic industry, Xilinx has provided users with mechanisms to control the processing behaviors of the core tool algorithms. This quick reference guide provides a simplified description of how these mechanisms work.

The guide begins with a section titled "CONSTRAINT ENTRY MECHANISMS", which describes the types of constraints and constraint files that are available within M1 and how they are applied. The second section "EFFICIENT USE OF TIMESPECS" identifies the mechanics of how to constrain your design to work best for the M1 tools, with particular emphasis on getting the required results in less runtime.

This document does NOT describe all of the constraints of the M1 system, instead the complete constraint reference guide may be found as part of the on-line documentation, in the libraries chapter. Users should consult the list of known issues distributed with each software release to become familiar with any issues which may exist between the desired behavior described in this document, and the reality of a particular software release.

Several Appendices are included to provide a quick reference to the valid syntax of the constraints and to illustrate the changes they undergo as they are mapped from the

source design (logical constraint representation) to the compiled design (physical constraint representation)

Constraint Entry Mechanisms

The M1 version of the Xilinx Alliance and Foundation design tools allow users to control the implementation of a design through constraints which affect the mapping and layout of the physical circuit. Additionally, a user must specify the "path" timing requirements of the circuit to obtain the best results, and allow the implementation tools to determine the layout which satisfies these requirements.

The various design constraints available for use within "M1" can be entered at design creation (i.e. the logical domain) or after the design is mapped to the physical domain. Constraints entered in the logical domain are either "captured" within a schematic, or "applied" to a synthesis process then "forward annotated" through the netlisting mechanism (DC2NCF for Synopsys FPGA and Design Compiler flows, XNF file generation for the Synopsys FPGA Express flow).

Constraints entered in the "physical" domain are entered directly into the Physical Constraints File (PCF). These constraints are conceptually the same as those entered during design creation, however they are directly related to objects within the physical design database, and are therefore applied using the PCF syntax.

Figure 1 illustrates the constraints entry mechanisms of the M1 version of the Xilinx tools. The paragraphs below describe the steps involved in implementing a design within the M1 tools, with specific focus on how constraints are entered at each stage of design processing.

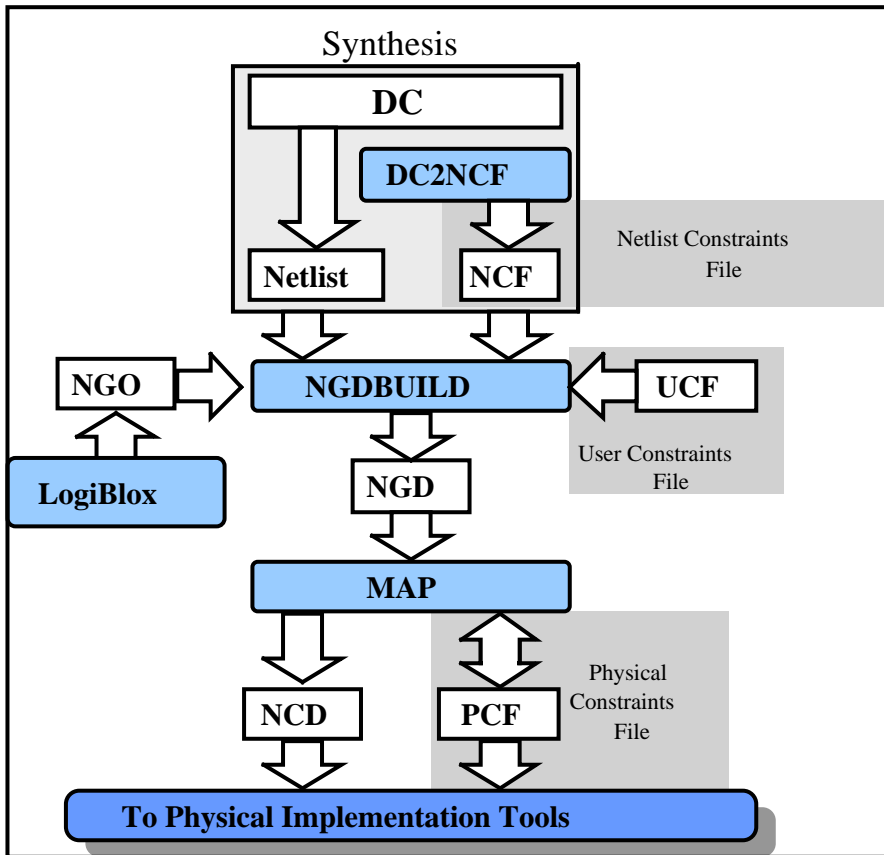


Figure 1: Constraint Entry Flow

Translating and Merging Logical Designs

The process of implementing a design within the M1 tools starts by constructing a Logical Design file (NGD) which represents the design created by the NGDBUILD application (as shown in Figure 1).

The NGD file contains all of the design's logic structures (gates) and constraints. The NGD file is produced through the NGDBUILD process which controls the translation and merging of all of the related logic design files. All design files are translated from industry standard netlists into intermediate NGO files by one of two netlist translation programs XNF2NGD or EDIF2NGD. The exception to this rule is logic which is created through the use of LogiBlox

components. LogiBlox components may be compiled directly in memory¹, and are therefore never written to disc as a separate intermediate NGO file.

The Netlist Constraint File (NCF)

The Netlist Constraint File was developed as an alternative means for third party vendors to provide the M1 tools with design constraints. Historically, these constraints are in the design netlist and are annotated to the equivalent elements within the designs NGO file. In M1, translating a logic design netlist to a NGO file includes annotating constraints present in the NCF file to the NGO design elements. The NCF file is local in scope and therefore must have the same root name as the netlist being translated. In addition, the

1. Whether or not an NGO is written to disc for a LogiBlox component is dependent on the specific design creation technology being used. Many synthesis flows require that an NGO file be written to disc during Component specification, while the typical schematic flow provides for on-the-fly compilation of NGO files.

local scope of the netlist allow entries within the NCF file to refer to specific nets or symbols without prefixing the entire hierarchical path of the net or symbol.

Figure 1 illustrates the Synopsys workstation design flow which leverages the NCF file. The program DC2NCF converts the Synopsys constraints into NCF syntax. Refer to *XSI Design Flow* section of the *Synopsys (XSI) Interface / Tutorial Guide* for detailed information on using this flow.

The User Constraint File (UCF)

The User Constraint File was developed to provide users an easy mechanism for constraining a logical design without returning to design capture tools. The process of building the complete logical design representation (NGD files) is the job of *NGDBUILD*. In developing this complete design database, *NGDBUILD* annotates design constraints which are present within a UCF file¹. The syntax for the UCF constraints file is identical to the syntax of the Netlist Constraints File (NCF). The two main differences between the NCF and UCF files are how the objects within the file are identified (as previously mentioned), in addition, to processing constraint name collisions differently. A constraint which is being applied via the UCF file must specify the complete hierarchical path name for the instance or net being constrained, while an NCF constraint need only reference the specific net or symbol within the associated netlist. The difference in hierarchical path name requirements arises from the scope of the design files themselves; NCF files have a local scope and can therefore tolerate local references, while UCF files have a global scope and therefore require full hierarchical path names.

The second difference is that NCF constraint annotation employs no resolution mechanism. Because the Netlist Constraint File (NCF) is considered an alternative mechanisms for design creation packages to pass constraints to the M1 implementation tools, no conflicts should occur. In contrast, the User Constraint File (UCF) annotation includes a resolution mechanism which allows for UCF constraints to over-write constraints present in the NCF. UCF constraints are considered more significant due to their later appearance in the design flow, and provide a mechanism for establishing or modifying logical design constraints without requiring the user to re-enter a schematic or synthesis tool.

FROM:TO style Timespecs:

Several examples of FROM:TO Timespecs using UCF file syntax are shown in Figure 2.

```
# This is a comment line
# UCF FROM : TO style Timespecs
NET DATA_EN TNM = PIPEA ;
TIMEGRP BUSPADS = PADS(BUS*) ;
TIMESPEC TS01 = FROM:BUSPADS:TO:PIPEA:20;
# Spaces or colons (:) may be used as field separators
TIMESPEC TS02 = FROM FFS TO RAMS 15 ;
```

Figure 2: UCF FROM:TO Timespec Examples

The first line Figure 2 illustrates the application of the TNM (Timing Name) "PIPEA" to the net named "DATA_EN". TNMs are used by the M1 tools in the same way as the XACT 5.2 tools - identification of a group of design objects which are to be referenced within a Timespec. The collection of design objects with the same TNM attributes attached to them is known as a Timegroup, and are referenced within Timespecs as a "TIMEGRP". The M1 tools determine timegroup membership by tracing forward from the specified net to various "Synchronous Timing Points" within the design. "Synchronous Timing Points" are PADS, FFS, LATCHES, and RAMS (Refer to the M1 Users Guide for more information on the forward tracing mechanism).

The second line of Figure 2 illustrates the TIMEGRP design object formed using a name matching mechanism in conjunction with the pre-defined TIMEGRP "PADS". In this example, the TIMEGRP named "BUSPADS" will include only those PADS with names which start with "BUS".

Each of the user defined Timegroups is then used to define the object space constrained by the timing specification (TIMESPEC) named TS01. This timing specification states that all paths from each member of the "BUSPADS" group to each member of the PIPEA group can have path timing which is no greater than 20 nano-seconds (ns are the default units for time). The TIMESPEC TS02 illustrates a similar type of timing constraint using the pre-defined groups FFS and RAMS.

It is worthwhile noting that all FROM:TO Timespecs must be relative to a TIMEGRP. Figure 2 illustrates that TIMEGROUPS may be defined by the user either explicitly (TIMEGRPs) or implicitly (TNMs), or they may be pre-defined groups (PADS).

1. Versions prior to M1.2 required the -uc switch to identify a User Constraint File which is to be annotated to the design. Versions M1.2 and later allow UCF file annotation to be performed by default if the UCF file has the same base name as the input netlist.

PERIOD style Timespecs:

In addition to FROM:TO Timespecs, there are several other forms of Timespecs provided by the M1 tools. Of particular significance is the PERIOD Timespec. Figure 3 illustrates the use of the PERIOD Timespec referenced to a time-groups CLK2_GRP and CLK3. In addition, Figure 3 demonstrates how constraints and nets may be named the same because they occupy separate name-spaces. Finally, shows the constraint syntax support which allows the definition of one Timespec relative to another is shown (the value of TS04 is declared to be two times that of TS03).

UCF PERIOD style Timespecs

```
NET CLK2 TNM = CLK2_GRP;  
NET CLK3 TNM = CLK3;  
TIMESPEC TS03 = PERIOD CLK2_GRP 50 ;  
TIMESPEC TS04 = PERIOD CLK3 TS03 * 2;
```

Figure 3: UCF PERIOD Constraint Using Timespecs

The PERIOD constraint covers all timing paths which start or end at a register, latch or synchronous RAM which is clocked by the referenced net. The singular exception to this rule are paths to output pads which are not covered by the PERIOD constraint. (Input pads which are the source of a "Pad-to-Setup" timing path for one of the specified synchronous elements are covered by the PERIOD constraint.)

In addition to the TIMESPEC form of the PERIOD constraint, there is a NET form of this constraint. The

TIMESPEC form of the PERIOD constraint allows flexibility in group definitions and allows the definition of clock timing relative to another TIMESPEC¹. The flexibility of the TIMESPEC form of the PERIOD constraint arises from being able to modify the contents of the TIMEGRP once the design has been mapped. By adding or removing objects from the TIMGRP, which are listed in the PCF file, the paths which are covered by the PERIOD constraint may be altered.

If the flexibility of the TIMESPEC form is not required, the NET form of the PERIOD constraint may be used. The syntax for the NET form of the PERIOD constraint is more simple than the TIMESPEC form, while continuing to provide the same path coverage. Figure 4 illustrates the syntax of the NET form of the PERIOD constraint.

Net form of the PERIOD timing constraints (no TSIdentifier)

```
NET CLK PERIOD = 40 ;
```

Net form of the OFFSET timing constraint

```
NET ADD0_IN OFFSET = IN 14 AFTER CLK ;
```

Figure 4: UCF Example of PERIOD Using Net and OFFSET Timing Constraints

The OFFSET Constraint, shown in Figure 4, is applied to a net connecting to a PAD (see Figure 5). It defines the delay of a signal relative to a clock, and is only valid for registered data paths. The OFFSET constraint specifies the signal delay external to the chip, allowing the implementation tools to automatically adjust relevant internal delays (CLK Buffer and distribution delays) to accommodate the external delay specified with this constraint.

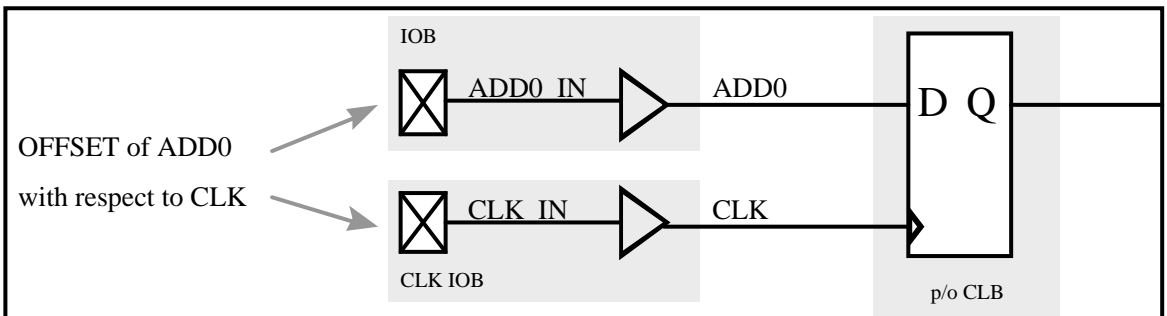


Figure 5: Typical Implementation Which Uses an OFFSET Constraint

1. Relative Timespecs can only be applied to similar Timespecs. For example, a PERIOD TIMESPEC may be defined in terms of another PERIOD TIMESPEC, but not a FROM:TO TIMESPEC.

In analyzing OFFSET paths, the M1 timing tools adjust the PERIOD associated with the constrained synchronous element based on both the timing specified in the OFFSET constraint and the delay of the referenced clock signal. In Figure 5, assume a delay of 8ns for the signal CLK to arrive at the CLB in addition to a 5 ns setup time, and a 14 ns OFFSET delay for the signal ADD0. The M1 tools would then allocate 29 ns for the signal ADD0 to arrive at the CLB input pin ($40\text{ns} - 14\text{ns} + 8\text{ns} - 5\text{ns} = 29\text{ns}$).

This same timing constraint may be applied using the FROM:PADS:TO:FFS timing constraint. However, using a FROM:TO methodology would require the designer to know the intrinsic CLK net delay, and the user would have to adjust the value assigned to the FROM:TO Timespec. The internal CLK net delay is implicit in the OFFSET / PERIOD constraint. Furthermore, migrating the design to another speed grade or device, will require modification of the FROM:TO Timespec to accommodate the new intrinsic CLK net delay. It should be noted that an alternative solution is to use the flip-flop in the IOB of certain FPGA architectures (XC4000E/EX, for instance), as the clock-to-setup time is specified in the Data Book.

Layout Constraints

The mapping constraint of Figure 6 illustrate some of the capabilities for controlling the implementation process for a design. The OPTIMIZE attribute is attached to the block of logic associated with the instance "GLUE". All of the combinatorial logic within the block GLUE will be optimized for speed (minimizing levels of logic) while other aspects of the design will be processed by the default mapping algorithms (assuming the design based optimization switches are not issued).

Mapping Constraint

```
INST GLUE OPTIMIZE = SPEED;
```

Layout Constraint

```
NET IOBLOCK/DATA0_IN LOC = P12;
```

Figure 6: UCF Mapping and Layout Constraints

The Layout constraint of Figure 6 illustrates the use of a full hierarchical path name for the net named DATA0_IN in the application of the I/O location constraint. In this example, IOBLOCK is a hierarchical boundary which contains the net DATA0_IN. Location constraints applied to "Pad nets" are

used to constrain the Location of the PAD itself, in this case to site P12.¹

Converting a Logical Design into a Physical Design

The process of mapping translates a design from the logical design domain to the physical design domain. As illustrated in Figure 1, the MAP process creates both the physical design components (CLBs, IOBs, etc.) and the physical design constraints (layout and timing). The Physical design components are written into a Native Circuit Description (NCD) file. The Physical design constraints are written into the Physical Constraints File (PCF).

As the design flow of Figure 1 shows, MAP not only writes a PCF file, but also reads a specified pre-existing PCF file. MAP reads an existing PCF file in order to facilitate the over-riding of constraints which are contained within a logic design using the "last one wins" resolution mechanism provided by the PCF file. The following paragraphs briefly describe this mechanism.

Last One Wins Resolution

MAP creates new Physical Design Constraints each time it converts a logical design into a physical design. The constraints which are created during this process are written into the "Schematic" section of the PCF file. This section is recreated each time MAP is run based on the constraints which are contained within the NGD file. The schematic section is always written to the beginning of the PCF file, and constraints which are in the PCF file but outside of the Schematic section (after the line "SCHEMATIC END") are considered to be in the "User" section of the PCF file. The User section is read, syntactically checked, and rewritten each time MAP is run. As these constraints always follow those written into the Schematic section, they will always take precedence in the "last-one-wins" rule.

The PCF File

The Physical design is the "domain" of the layout tools, and as such the PCF file is written in terms which these tools can readily work with. Layout and Timing constraints are written in terms of the physical design's components (COMPs), fractions of these COMPs (BELs) and collections of COMPs (Macros). Because of this different design viewpoint, the PCF syntax is not necessarily the same as the logical design constraint files (UCF/NCF). Furthermore, as the PCF file is written for use by the physical implementation tools, its syntax is not as intuitive as the

1. If the design contains a PAD, the constraint could have been just as easily applied to it directly (some design flows do not provide explicit I/O Pads in the design netlist).

SCHEMATIC START ;

TIMEGRP "PIPEA" = BEL REGA0 BEL REGA1 BEL REGA2 BEL REGA3;

TIMEGRP "BUSPADS" = BEL BUS0 BEL BUS1 BEL BUS2 BEL BUS3;

TIMEGRP FFS = FFS(".*");

TIMEGRP RAMS = RAMS(".*");

TS01 = MAXDELAY FROM TIMEGRP "BUSPADS" TO TIMEGRP "PIPEA" 20 nS ;

TS02 = MAXDELAY FROM TIMEGRP "FFS" TO TIMEGRP "RAMS" 15 nS ;

SCHEMATIC END;

// user added PCF directives

TS01 = MAXDELAY FROM TIMEGRP **BUSPADS** TO TIMEGRP **PIPEA 30**;

COMP **BUS0** LOCATE = SITE **P12**;

Figure 7: PCF Example Including Timespecs

UCF file¹. Regardless of the syntactical challenges associated with using a PCF file, many designers will choose to work at the physical level of design abstraction because:

1. It is readily modified and immediately applicable to the task at hand - implementing an FPGA (i.e., there is no need to re-run NGDBUILD or MAP in order to run layout or analysis tools)
2. Implication of logical design structures on the physical designs implementation only become obvious once the design is evaluated using the physical tools. Altering the PCF file for "what-if" analysis can be desirable.
3. Certain constraints are only available within the PCF file

Figure 7 shows both the Schematic and User sections of a PCF. In this example the Timespec TS01 would have a value of 30 ns when utilized by the various physical design and analysis tools because of the "last-one-wins" resolution mechanism. The PCF entries within the Schematic section correspond to the Timespecs illustrated in Figure 2 and Figure 5.

It is important to note that each of the TIMEGRPs used within Timespecs TS01 and TS02 are defined prior to their use. This is a system requirement, which will always be

met for PCF files which are written by the mapper. Furthermore, notice that even the predefined groups have been defined within the PCF file (using a wildcard name matching mechanism). The specification of these predefined groups is also a system requirement.

With respect to the design processing constraints of Figure 5, the OPTIMIZE attribute is a mapper directive which serves no purpose to the physical design tools. Therefore, this attribute is not written to the PCF file. The layout constraint LOC is moved from the NET which it was applied to in the NGD file to the IOB COMP (named BUS0) in the physical design². As can be seen, the syntax has changed slightly to represent the format native to the physical layout tools.

Constraint Precedence Order

A user may assign a precedence to timespecs only within a certain class of constraints. For example, a user may specify a priority for one FROM TO specification over another, but may not specify a FROM TO constraint to have priority over a TIG constraint. Figure 8 illustrates the explicit assignment of priorities between two same-class timing constraints, the lowest number having the highest priority.

Priority UCF Example

TIMESPEC **TS01** = FROM **GROUPA** TO **GROUPB 40** PRIORITY **4**;

TIMESPEC **TS02** = FROM **GROUP1** TO **GROUP2 35** PRIORITY **2**;

Figure 8: UCF Priority Specification Example

1. The creation of a common UCF/PCF syntax remains the goal of core tools marketing and development. However, planned modifications to the entry mechanisms for PCF constraints will likely obsolete this need in the M2 timeframe.
2. A record of this name change is provided in the Map Report file in the Symbol Cross Reference section.

Across Constraint Sources	
<i>Lowest Priority</i>	Input Netlist / Netlist Constraint File
	User Constraint File
<i>Highest Priority</i>	Physical Constraint File
Within Constraint sources	
<i>Lowest Priority</i>	“Allpaths” type constraints
	Period Specification
	FROM:FFS:TO:FFS Specification
	FROM:FFS:THRU:TM1:TO:FFS Spec
<i>Highest Priority</i>	TIG (Timing Ignore)

Table 1: Order of Constraint and Constraint File Precedence

Table 1 illustrates the order of precedence for constraint files and timing constraints.

Layout constraints also have an inherent precedence which is based on the type of constraint and the site description provided to the tools. To learn more about these please refer to the M1 User Guide.

Efficient Use of Timespecs

The previous section described the mechanisms available for constraining a designs implementation within the M1 tools. The Appendices which follow summarize each of the constraints that are available for use. The result of all of this capability is an incredibly versatile system for controlling the design implementation process. In fact, this system is so robust that there are many ways to describe the same set of timing requirements to the tools. The natural question which will arise is “How should I describe my requirements to the tools?”

The robust nature of the language enables a designer to define their design requirements at the highest level of abstraction first, and then fine tune the timing requirements using more specific TIMESPECs as necessary. This is the methodology that should be followed. The examples in Figure 9, Figure 10, Figure 11, and Figure 12 provide insight into what is meant by these statements.

The following bullets serve to identify the reasons why this methodology should be followed from a tool runtime perspective.

- The use of explicit timegroups causes slower runtimes than the use of implicit timegroups which result from the use of constraints such as PERIOD.
- The processing of larger Timegroups takes longer than the processing of smaller Timegroups.
- The use of many specific Timespecs results in slower runtimes than the use of a smaller set of more general Timespecs.
- The use of unqualified “TO” specifications runs faster than specifications with a “FROM” qualifier (TW traces back from timing end-points).

Given these statements, it may be clear that design runtime will be improved if a “Qualified Global” timing methodology is employed as compared to a “Thorough-detailed” timing methodology.

The “Starter Set” of Timing Constraints

The following examples clearly identify the “preferred” mechanism for controlling the timing of your design. The preferred method assumes a goal of getting the required results in the fastest run-time possible. If the design has a single clock and required I/O timing which equals the clock period, all that is needed are the three constraints shown in Figure 9.

```
# Global UCF Example
NET CLK1 PERIOD = 40;
NET OUT* OFFSET = OUT 13 AFTER CLK;
TIMESPEC TS01 = FROM PADS TO PADS 40;
```

Figure 9: Global UCF File Example

```
# Global PCF example
SCHEMATIC START;
. . .
NET PERIOD "CLK_IN" = 40 nS HIGH 50.00% ;
COMP "OUT1_PAD" OFFSET = OUT 40 ns AFTER COMP "CLK";
COMP "OUT2_PAD" OFFSET = OUT 40 ns AFTER COMP "CLK";
TS01 = MAXDELAY FROM TIMEGRP "PADS" TO TIMEGRP "PADS" 40000 pS PRIORITY 0;
SCHEMATIC END;
```

Figure 10: Global PCF Example

The PERIOD constraint covers all pad-to-setup and clock-to-setup timing paths. The OFFSET constraint covers the clock-to-pad timing for each of the output nets beginning with *OUT*. Both the OFFSET and PERIOD constraints accounts for the delay of the Clock Buffer/Net in the I/O timing calculations.

Figure 10 illustrates the differences in syntax between the NCF/UCF and PCF languages. In addition to the syntactical changes, it is important to note that net and instance names may change. As an example, one of the net matches resulting from the UCF "NET *OUT*" constraint is now applied to "COMP *OUT1_PAD*". The name OUT1_PAD is the name assigned to the pad instance. In addition to name changes, another difference to note is the verbosity of the PCF. In the PCF there is added syntax for "MAXDELAY", "TIMEGRP" and "Priority". These are all

optional qualifications of the Timespec within the UCF, but written explicitly to the PCF file illustrating the full flexibility of the language.

Figure 11 illustrates the use of both Global Constraints (PERIOD, OFFSET) for generally constraining the design, and detailed TimeSpecs (FROM:THRU:TO) for providing fast and slow exceptions to the general timing requirements. Because the amount of constraints placed on a design directly effect run time, it is recommended that users first apply global constraints, then apply individual constraints only to those elements of the design which require additional constraints (or an exception to a constraint). The more global the constraints, the better the runtime performance of the tools.

```
# Sample UCF file
# Specify target device and package
CONFIG PART = XC4010e-PQ208-3;

# Global Constraints
NET CLK1 PERIOD = 40;
NET DATA_OUT* OFFSET = OUT 15 AFTER DCLK;
TIMESPEC TS01 = FROM PADS TO PADS 40;

# Layout Constraints
NET SCLINF LOC = P125;

# Detailed Constraints
# Exception to cover X_DAT and Y_DAT buses
NET ?_DAT* OFFSET = OUT 25 AFTER CLK_IN;
# Ignore timing on reset net
NET RESET_N TIG;
# Slow Exception for data leaving INA FFs
TIMESPEC TS02 = FROM FFS(INA*) TO FFS 80;
# Faster timing required for data leaving RAM
TIMESPEC TS03 = FROM RAMS TO FFS 20;
# Form special time groups related to RAMs
INST $1I64 TNM = SPDRAM;
NET RAMBUS0 TPTHU = RAMVIA;
NET RAMBUS1 TPTHU = RAMVIA;
# Specify timing for this special timing path
TIMESPEC TS04 = FROM SPDRAM THRU RAMVIA TO FFS 45;
```

Figure 11: Global / Detailed UCF Example

Standard Block Delay Symbols

Table 2 defines the block delay symbols and their description. There is a one-to-many correspondence between these symbol names and data book symbol names. For those symbols listed as having a default value of disabled, no timing analysis will be performed on paths which have segment composed of symbol path. As an example, paths which have a set/reset to output path will not be analyzed. Any of the block delays (Symbol) listed in Table 2 may be explicitly enabled or disabled using the PCF.

Figure 12 gives an example of the PCF syntax which would be used to enable the path tracing for all paths which contain RAM data to out paths. Note that this PCF directive is placed in the user section of the PCF.

```
SCHEMATIC END;
// This is a PCF Comment line
// Enable RAM data to out path tracing
ENABLE = ram_d_o;
```

Figure 12: PCF Example to Enable Block Delays

Symbol	Default	Description
reg_sr_q	Disabled	Set/Reset to output propagation delay
lat_d_q	Disabled	Data to output transparent latch delay
ram_d_o	Disabled	Ram data to output propagation delay
ram_we_o	Enabled	Ram write enable to output propagation delay
tbuf_t_o	Enabled	TBUF tri-state to output propagation delay
tbuf_i_o	Enabled	TBUF input to output propagation delay
io_pad_I	Enabled	IO pad to input propagation delay
io_t_pad	Enabled	IO tri-state to pad propagation delay
io_o_I	Enabled	IO output to input propagation delay. Disabled for tri-stated IOBs.
io_o_pad	Enabled	IO output to pad propagation delay.

Table 2. Timing Symbols and Their Default Values

Appendix A: Table of M1 supported Constraints

For further explanation and examples of each of the constraints, please see the *Syntax Summary* section of the *Development System Reference Guide*.

Constraint	Merge 1.0			
	Schematic	NCF	UCF	PCF
ADD	Y	N	N	N
BASE	Y	N	N	N
BLKNM	Y	Y	Y	N
BUFG	Y	Y	Y	N
COLLAPSE	Y	Y	Y	N
CONFIG	Y	N	N	N
Config. Constraints	Y	Y	Y	N
D_INVERT	Y	N	N	N
DECODE	Y	Y	Y	N
DIVIDE1_BY DIVIDE2_BY	Y	Y	Y	N
DOUBLE	Y	N	N	N
DROP_SPEC	N	Y	Y	Y **
EQUATE_F EQUATE_G	Y	N	N	N
FAST	Y	Y	Y	N
FILE	Y	N	N	N
HBLKNM	Y	Y	Y	N
HU_SET	Y	Y	Y	N
INIT	Y	Y	Y	N
IO	Y	Y	Y	N
KEEP	Y	Y	Y	N
LOC =	Y	Y	Y	Y **

** user beware - while constraint is available there are syntax differences.

Constraint	Merge 1.0			
	Design	NCF	UCF	PCF
LOWPWR=(ON OFF)	Y	N	N	N
MAP	N	N	N	N
MAXDELAY	Y	Y	Y	Y **
MAXSKEW	Y	Y	Y	Y **
MEDDELAY	Y	Y	Y	N
MINIM	Y	N	N	N
Net Flag Attributes				
F	Y	N	N	
H	Y	N	N	
P	N	N	N	Y
S	Y	N	N	N
X	Y	N	N	
NODELAY	Y	Y	Y	N
NOREDUCE	Y	Y	Y	N
OFFSET	N	Y	Y	Y **
OPT	Y	N	N	N
OPTIMIZE	Y	Y	Y	N

** user beware - while constraint is available there are syntax differences.

Constraint	Merge 1.0			
	Design	NCF	UCF	PCF
OPT_EFFORT	Y	Y	Y	N
PART	Y	Y	Y	N
PERIOD	Y	Y	Y	Y **
PROHIBIT	Y	Y	Y	Y **
PWR_MODE	Y	Y	Y	N
REG	Y	N	N	N
RLOC	Y	Y	Y	N
RLOC_ORIGIN	Y	Y	Y	N
RLOC_RANGE	Y	Y	Y	N
SLOW	Y	Y	Y	N
TIG	Y	Y	Y	Y **
Timegroup Attributes	Y	Y	Y	N
TNM	Y	Y	Y	Y
TPSYNC	Y	Y	Y	N
TPTHRU	Y	Y	Y	N
TSidentifier = IGNORE	Y	N	N	N
Other TSidentifier	Y	Y	Y	Y **
U_SET	Y	Y	Y	N
USE_RLOC	Y	Y	Y	N
WIREAND	Y	Y	Y	N

** user beware - while constraint is available there are syntax differences

Appendix B: Constraining LogiBLOX RAM/ROM with the Synopsys M1 Design Flows

In the M1 XSI HDL methodology, whenever large blocks of RAM/ROM are needed, LogiBLOX RAM/ROM modules are instantiated in the HDL code. With LogiBLOX RAM/ROM modules instantiated in the HDL code, timing and/or placement constraints on these RAM/ROM modules, and the RAM/ROM primitives that comprise these modules, can be specified in a .ucf file. To create timing and/or placement constraints for RAM/ROM LogiBLOX modules, knowledge of how many primitives will be used and how the primitives, and/or how the RAM/ROM LogiBLOX modules are named is needed.

- **How many primitives are inside a LogiBLOX RAM/ROM module?**

When a RAM/ROM is specified with LogiBLOX, the RAM/ROM depth and width are specified. If the RAM/ROM depth is divisible by 32, then 32x1 primitives are used. If the RAM/ROM depth is not divisible by 32, then 16x1 primitives are used instead. In the case of dual-port RAMs, 16x1 primitives are always used. Based on whether 32x1 or 16x1 primitives are used, the number of RAM/ROM can be calculated.

For example, if a RAM48x4 was required for a design, RAM16x1 primitives would be used. Based on the width, there would be four banks of RAM16x1s. Based on the depth, each bank would have three RAM16x1s.

- **How are the RAM primitives inside LogiBLOX RAM/ROM modules named?**

Using the example of a RAM48x4, the RAM primitives inside the LogiBLOX would be named as follows:

```
MEM0_0 MEM1_0 MEM2_0 MEM3_0
MEM0_1 MEM1_1 MEM2_1 MEM3_1
MEM0_2 MEM1_2 MEM2_2 MEM3_2
```

Each primitive in a LogiBLOX RAM/ROM module has a instance name of MEMx_y, where y represents the primitive position in the bank of memory, and where x represents the bit position of the RAM/ROM output.

For the next 4 items, refer to the Verilog/VHDL examples included at the end of this solution. The Verilog/VHDL example instantiates a RAM32x2S, which is in the bottom of the hierarchy. The RAM32x2S was made with LogiBLOX. The next 4 items are written within the context of the Verilog examples, but also apply to the VHDL examples as well. Note, the runscripts included were designed for FPGA Compiler. If you want to use Design Compiler, remove the replace_fpga step.

- **How to reference a LogiBLOX module/component in the FPGA/Design Compiler flow**

LogiBLOX RAM/ROM modules in the M1 FPGA/Design Compiler flow are constrained via a .ucf file. LogiBLOX RAM/ROM modules instantiated in the HDL code can be referenced by the full-hierarchical instance name. If a LogiBLOX RAM/ROM module is at the top-level of the HDL code, then the instance name of the LogiBLOX RAM/ROM module is just the instantiated instance name. In the case of a LogiBLOX RAM/ROM which is instantiated within the hierarchy of the design, the instance name of the LogiBLOX RAM/ROM module is the concatenation of all instance which contain the LogiBLOX RAM/ROM; The concatenated instance names are separated by a /. In the example, the RAM32X1S is named memory. memory is instantiated in Verilog module inside with an instance name U0. inside is instantiated in the top-level module test. Therefore, the RAM32X1S can be referenced in a .ucf file as U0/U0. For example, to attach a TNM to this block of RAM, the following line could be used in the .ucf file:

```
INST U0/U0 TNM=block1;
```

Since U0/U0 is composed of two primitives, a timegroup called block1 would be created; block1 TNM could be used throughout the .ucf file as a timespec end/start point, and/or U0/U0 could have a LOC area constraint applied to it. If the RAM32X1S has been instantiated in the top-level file, and the instance name used in the instantiation was U0, then this block of RAM could just be referenced by U0.

- **How to reference a LogiBLOX module/component in the FPGA Express flow**

LogiBLOX RAM/ROM modules in the M1 FPGA Express flow are constrained via a .ucf file. LogiBLOX RAM/ROM modules instantiated in the HDL code can be referenced by the full-hierarchical instance name. If a LogiBLOX RAM/ROM module is at the top-level of the HDL code, then the instance name of the LogiBLOX RAM/ROM module is just the instantiated instance name. In the case of a LogiBLOX RAM/ROM which is instantiated within the hierarchy of the design, the instance name of the LogiBLOX RAM/ROM module is the concatenation of all instance which contain the LogiBLOX RAM/ROM; The concatenated instance names are separated by a _. In the example, the RAM32X1S is named memory. memory is instantiated in Verilog module inside with an instance name U0. inside is instantiated in the top-level module test. Therefore, the RAM32X1S can be referenced in a .ucf file as U0_U0. For example, to attach a TNM to this block of RAM, the following line could be used in the .ucf file:

```
INST U0_U0 TNM=block1;
```


Since U0_U0 is composed of two primitives, a timegroup called block1 would be created; block1 TNM could be used throughout the .ucf file as a timespec end/start point, and/or U0_U0 could have a LOC area constraint applied to it. If the RAM32X1S has been instantiated in the top-level file, and the instance name used in the instantiation was U0, then this block of RAM could just be referenced by U0.

- **How to reference the primitives of a LogiBLOX module/component in the FPGA/Design Compiler flow**

Sometimes its necessary to apply constraints to the primitives that compose the LogiBLOX RAM/ROM module. For example, if you choose a floorplanning strategy to implement your design, it may be necessary to apply LOC constraints to one or more primitives inside a LogiBLOX RAM/ROM module. Consider the RAM32x2S example above, suppose that the each of the RAM primitives had to be constrained to a particular clb location. Based on the rules for determining the MEMx_y instance names, using the example from above, each of RAM primitives could be referenced by concatenating the full-hierarchical name to each of the MEMx_y names. The RAM32x2S created by LogiBLOX would have primitives named MEM0_0 and

MEM1_0. So, clb constraints in a .ucf file for each of these two items would be:

```
INST U0/U0/MEM0_0 LOC=CLB_R10C10;
INST U0/U0/MEM0_1 LOC=CLB_R11C11;
```

- **How to reference the primitives of a LogiBLOX module/component in the FPGA Express flow**

Sometimes its necessary to apply constraints to the primitives that compose the LogiBLOX RAM/ROM module. For example, if you choose a floorplanning strategy to implement your design, it may be necessary to apply LOC constraints to one or more primitives inside a LogiBLOX RAM/ROM module. Consider the RAM32x2S example above, suppose that the each of the RAM primitives had to be constrained to a particular clb location. Based on the rules for determining the MEMx_y instance names, using the example from above, each of RAM primitives could be referenced by concatenating the full-hierarchical name to each of the MEMx_y names. The RAM32x2S created by LogiBLOX would have primitives named MEM0_0 and MEM1_0. So, clb constraints in a .ucf file for each of these two items would be:

```
INST U0_U0/MEM0_0 LOC=CLB_R10C10;
INST U0_U0/MEM0_1 LOC=CLB_R11C11;
```

FPGA/Design Compiler Verilog Example

test.v:

```
module test(DATA,DATAOUT,ADDR,C,ENB);
input [1:0] DATA;
output [1:0] DATAOUT;
input [4:0] ADDR;
input C;
input ENB;
wire [1:0] dataoutreg;
reg [1:0] datareg;
reg [1:0] DATAOUT;
reg [4:0] addrreg;

inside U0 (.MDATA(datareg),.MDATAOUT(dataoutreg),.MADDR(addrreg),.C(C),.WE(ENB));

always@(posedge C) datareg = DATA;
always@(posedge C) DATAOUT = dataoutreg;
always@(posedge C) addrreg = ADDR; endmodule
```

inside.v:

```
module inside(MDATA,MDATAOUT,MADDR,C,WE);
input [1:0] MDATA;
output [1:0] MDATAOUT;
input [4:0] MADDR;
input C;
input WE;
```

```
memory U0 ( .A(MADDR), .DO(MDATAOUT), .DI(MDATA), .WR_EN(WE), .WR_CLK(C));  
endmodule
```

memory.v:

```
module memory(A, DO, DI, WR_EN, WR_CLK);  
input [4:0] A;  
output [1:0] DO;  
input [1:0] DI;  
input WR_EN;  
input WR_CLK;  
endmodule
```

runscript:

```
TOP=test part = "4028expg299-3"  
read -f verilog "guts.v"  
read -f verilog "inside.v"  
read -f verilog "test.v"  
current_design TOP  
remove_constraint -all  
set_port_is_pad ""  
insert_pads  
compile  
write -format db -hierarchy -output TOP + "_compiled.db"  
replace_fpga  
set_attribute TOP "part" -type string part  
write -format db -hierarchy -output TOP + ".db"  
ungroup -all -flatten  
write_script > TOP + ".dc" sh dc2ncf test.dc  
remove_design guts  
write -f xnf -h -o TOP + ".sxnf"
```

test.ucf:

```
INST "U0/U0" TNM = usermem;  
TIMESPEC TS_6= FROM : FFS :TO: usermem: 50;  
INST "U0/U0/mem0_0" LOC=CLB_R7C2;
```

FPGA Express Verilog Example

test.v:

```
module test(DATA,DATAOUT,ADDR,C,ENB);  
input [1:0] DATA;  
output [1:0] DATAOUT;  
input [4:0] ADDR;  
input C;  
input ENB;  
wire [1:0] dataoutreg;  
reg [1:0] datareg;  
reg [1:0] DATAOUT;  
reg [4:0] addrreg;  
  
inside U0 (.MDATA(datareg),.MDATAOUT(dataoutreg),.MADDR(addrreg),.C(C),.WE(ENB));  
  
always@(posedge C) datareg = DATA;  
always@(posedge C) DATAOUT = dataoutreg;
```

```
always@(posedge C) addrreg = ADDR; endmodule
```

inside.v:

```
module inside(MDATA,MDATAOUT,MADDR,C,WE);  
input [1:0] MDATA;  
output [1:0] MDATAOUT;  
input [4:0] MADDR;  
input C;  
input WE;  
  
memory U0 ( .A(MADDR), .DO(MDATAOUT), .DI(MDATA), .WR_EN(WE), .WR_CLK(C));  
endmodule
```

test.ucf:

```
INST "U0/U0" TNM = usermem;  
TIMESPEC TS_6= FROM : FFS :TO: usermem: 50;  
INST "U0/U0/mem0_0" LOC=CLB_R7C2;
```

FPGA/Design Compiler VHDL Example

test.vhd:

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
use IEEE.STD_LOGIC_UNSIGNED.all;  
  
entity test is  
port( DATA: in STD_LOGIC_VECTOR(1 downto 0);  
      DATAOUT: out STD_LOGIC_VECTOR(1 downto 0);  
      ADDR: in STD_LOGIC_VECTOR(4 downto 0);  
      C, ENB: in STD_LOGIC);  
end test;  
  
architecture details of test is  
signal dataoutreg,datareg: STD_LOGIC_VECTOR(1 downto 0);  
signal addrreg: STD_LOGIC_VECTOR(4 downto 0);  
  
component inside  
port(MDATA: in STD_LOGIC_VECTOR(1 downto 0);  
      MDATAOUT: out STD_LOGIC_VECTOR(1 downto 0);  
      MADDR: in STD_LOGIC_VECTOR(4 downto 0);  
      C,WE: in STD_LOGIC);  
end component;  
  
begin  
  U0: inside port map(MDATA=>datareg,MDATAOUT=>dataoutreg,MADDR=>addrreg,C=>C,WE=>ENB);  
  
  process( C )  
  begin  
    if(Cevent and C=1) then  
      datareg <= DATA;  
    end if;  
  end process;  
  
  process( C )  
  begin  
    if(Cevent and C=1) then  
      DATAOUT <= dataoutreg;  
    end if;  
  end process;  
end architecture details;
```

```
        end if;
    end process;

    process( C )
    begin
        if(Cevent and C=1) then
            addrreg <= ADDR;
        end if;
    end process;

end details;
```

inside.vhd:

```
entity inside is
    port(MDATA: in STD_LOGIC_VECTOR(1 downto 0);
          MDATAOUT: out STD_LOGIC_VECTOR(1 downto 0);
          MADDR: in STD_LOGIC_VECTOR(4 downto 0);
          C,WE: in STD_LOGIC); end inside;

architecture details of inside is
    component memory
        port(A: in STD_LOGIC_VECTOR(4 downto 0);
              DO: out STD_LOGIC_VECTOR(1 downto 0);
              DI: in STD_LOGIC_VECTOR(1 downto 0);
              WR_EN,WR_CLK: in STD_LOGIC);
    end component;

    begin
        U0: memory port map(A=>MADDR,DO=>MDATAOUT,DI=>MDATA,WR_EN=>WE,WR_CLK=>C);
    end details;
```

runscript:

```
TOP=test part = "4028expg299-3"
analyze -f vhdl "guts.vhd"
analyze -f vhdl "inside.vhd"
analyze -f vhdl "test.vhd"
elaborate TOP
current_design TOP
remove_constraint -all
set_port_is_pad "*"
insert_pads
compile
write -format db -hierarchy -output TOP + "_compiled.db"
replace_fpga
set_attribute TOP "part" -type string part
write -format db -hierarchy -output TOP + ".db"
ungroup -all -flatten
write_script > TOP + ".dc" sh dc2ncf test.dc
remove_design guts
write -f xnf -h -o TOP + ".sxnf"
```

test.ucf:

```
INST "U0/U0" TNM = usermem;
TIMESPEC TS_6= FROM : FFS :TO: usermem: 50;
INST "U0/U0/mem0_0" LOC=CLB_R7C2;
```

FPGA Express VHDL Example

test.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity test is
port(DATA: in STD_LOGIC_VECTOR(1 downto 0);
      DATAOUT: out STD_LOGIC_VECTOR(1 downto 0);
      ADDR: in STD_LOGIC_VECTOR(4 downto 0);
      C, ENB: in STD_LOGIC);
end test;

architecture details of test is
signal dataoutreg, datareg: STD_LOGIC_VECTOR(1 downto 0);
signal addrreg: STD_LOGIC_VECTOR(4 downto 0);

component inside
port(MDATA: in STD_LOGIC_VECTOR(1 downto 0);
      MDATAOUT: out STD_LOGIC_VECTOR(1 downto 0);
      MADDR: in STD_LOGIC_VECTOR(4 downto 0);
      C, WE: in STD_LOGIC);
end component;

begin
U0: inside port map(MDATA=>datareg, MDATAOUT=>dataoutreg, MADDR=>addrreg, C=>C, WE=>ENB);

process( C )
begin
    if(Cevent and C=1) then
        datareg <= DATA;
    end if;
end process;

process( C )
begin
    if(Cevent and C=1) then
        DATAOUT <= dataoutreg;
    end if;
end process;

process( C )
begin
    if(Cevent and C=1) then
        addrreg <= ADDR;
    end if;
end process;

end details;
```

inside.vhd:

```
entity inside is
port(MDATA: in STD_LOGIC_VECTOR(1 downto 0);
      MDATAOUT: out STD_LOGIC_VECTOR(1 downto 0);
      MADDR: in STD_LOGIC_VECTOR(4 downto 0);
      C, WE: in STD_LOGIC);
end inside;
```

architecture details of inside is

component memory

```
port(A: in STD_LOGIC_VECTOR(4 downto 0);
      DO: out STD_LOGIC_VECTOR(1 downto 0);
      DI: in STD_LOGIC_VECTOR(1 downto 0);
      WR_EN,WR_CLK: in STD_LOGIC);
```

end component;

begin

```
U0: memory port map(A=>MADDR,DO=>MDATAOUT,DI=>MDATA,WR_EN=>WE,WR_CLK=>C);
```

end details;

test.ucf:

INST "U0_U0" TNM = usermem;

TIMESPEC TS_6= FROM : FFS :TO: usermem: 50;

INST "U0_U0/mem0_0" LOC=CLB_R7C2;



The Programmable Logic CompanySM

Headquarters

Xilinx, Inc.
2100 Logic Drive
San Jose, CA 95124
U.S.A.

Tel: 1 (800) 255-7778
or 1 (408) 559-7778
Fax: 1 (800) 559-7114

Net: hotline@xilinx.com
Web: <http://www.xilinx.com>

North America

Irvine, California
(714) 727-0780

Englewood, Colorado
(303)220-7541

Sunnyvale, California
(408) 245-9850

Schaumburg, Illinois
(847) 605-1972

Nashua, New Hampshire
(603) 891-1098

Raleigh, North Carolina
(919) 846-3922

West Chester, Pennsylvania
(610) 430-3300

Dallas, Texas
(214) 960-1043

Europe

Xilinx Sarl
Jouy en Josas, France
Tel: (33) 1-34-63-01-01
Net: frhelp@xilinx.com

Xilinx GmbH
Aschheim, Germany
Tel: (49) 89-99-1549-01
Net: dlhelp@xilinx.com

Xilinx, Ltd.
Byfleet, United Kingdom
Tel: (44) 1-932-349401
Net: ukhelp@xilinx.com

Japan

Xilinx, K.K.
Tokyo, Japan
Tel: (03) 3297-9191

Asia Pacific

Xilinx Asia Pacific
Hong Kong
Tel: (852) 2424-5200
Net: hongkong@xilinx.com

© 1996 Xilinx, Inc. All rights reserved. The Xilinx name and the Xilinx logo are registered trademarks, all XC-designated products are trademarks, and the Programmable Logic Company is a service mark of Xilinx, Inc. All other trademarks and registered trademarks are the property of their respective owners.

Xilinx, Inc. does not assume any liability arising out of the application or use of any product described herein; nor does it convey any license under its patent, copyright or maskwork rights or any rights of others. Xilinx, Inc. reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx, Inc. cannot assume responsibility for the use of any circuitry described other than circuitry entirely embodied in its products. Products are manufactured under one or more of the following U.S. Patents: (4,847,612; 5,012,135; 4,967,107; 5,023,606; 4,940,909; 5,028,821; 4,870,302; 4,706,216; 4,758,985; 4,642,487; 4,695,740; 4,713,557; 4,750,155; 4,821,233; 4,746,822; 4,820,937; 4,783,607; 4,855,669; 5,047,710; 5,068,603; 4,855,619; 4,835,418; and 4,902,910. Xilinx, Inc. cannot assume responsibility for any circuits shown nor represent that they are free from patent infringement or of any other third party right. Xilinx, Inc. assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made.



The Programmable Logic CompanySM



0401643