SYNOPSYS®

Module Compiler™

User Manual

Version 1998.02

February 1998

Printed in the U.S.A.

Document Order Number: 31859-000 EA
Module Compiler User Manual, Version 1998.02

About this User Guide

Audience

This manual assumes that you are a logic designer or an electronics engineer with knowledge of CAE tools and ASIC design flow. Previous exposure to digital hardware structures is helpful.

Contents

The *Module Compiler User Guide* introduces the basic principles of logic design and describes how Modules Compiler facilitates the task of the logic designer. It discusses how to install the product, how to use the graphical user interface, the elements of the Module Compiler language, Module Compiler's support for various technology libraries, and describes the various output files and how to use them for analyzing results and planning future designs. There are also chapters on advanced usage of the MC language, building memories, layout support, and other topics that will guide more experienced designers.

Conventions

The following conventions are used in Synopsys documentation.

<table>
<tr><th>Convention</th><th>Description</th></tr>
<tr><td>sans-serif</td><td>Indicates command syntax.</td></tr>
<tr><td>sans-serif italic</td><td>Indicates a user specification, such as object_name.</td></tr>
<tr><td>monospace</td><td>In examples, shows system prompts, text from files, error messages, and reports printed by the system.</td></tr>
<tr><td>monospace bold</td><td>In examples, indicates user input (text the user types verbatim).</td></tr>
<tr><td>[ ]</td><td>Denotes optional parameters, such as pin1 [pin2, … pinN]. This example indicates that at least one pin name must be entered (pin1), but others are optional [pin2, … pinN].</td></tr>
<tr><td>< ></td><td>Denotes a required variable. The user must substitute an actual value of the designated type.</td></tr>
<tr><td>|</td><td>Indicates a choice among alternatives, such as low | medium | high. This example indicates that you can enter one of three possible values for an option: low, medium, or high.</td></tr>
<tr><td>_</td><td>Connects two terms that are read as a single term by the system. For example, design_space.</td></tr>
<tr><td>(Control-c)</td><td>Indicates the user holds down the Control key then presses c.</td></tr>
<tr><td>\</td><td>Indicates a continuation of a command line.</td></tr>
<tr><td>/</td><td>Indicates levels of directory structure.</td></tr>
<tr><td>name1 –> name2</td><td>Shows a menu selection. name1 is the menu name, and name2 is the item on the menu.</td></tr>
</table>

Table of Contents

List of Figures

List of Tables

Basic Concepts

1

This chapter provides a brief introduction to Module Compiler 1998.02 (MC) and associated technologies, along with some fundamental concepts and constraints affecting the use of MC.

Chapter 1 discusses the following topics:

- Module Compiler's functional design and process flow
- Basic concepts in datapath construction
- Hierarchy and Module Compiler
- Network objects and attributes
- Designer control features in Module Compiler

Computational Systems and Module Compiler

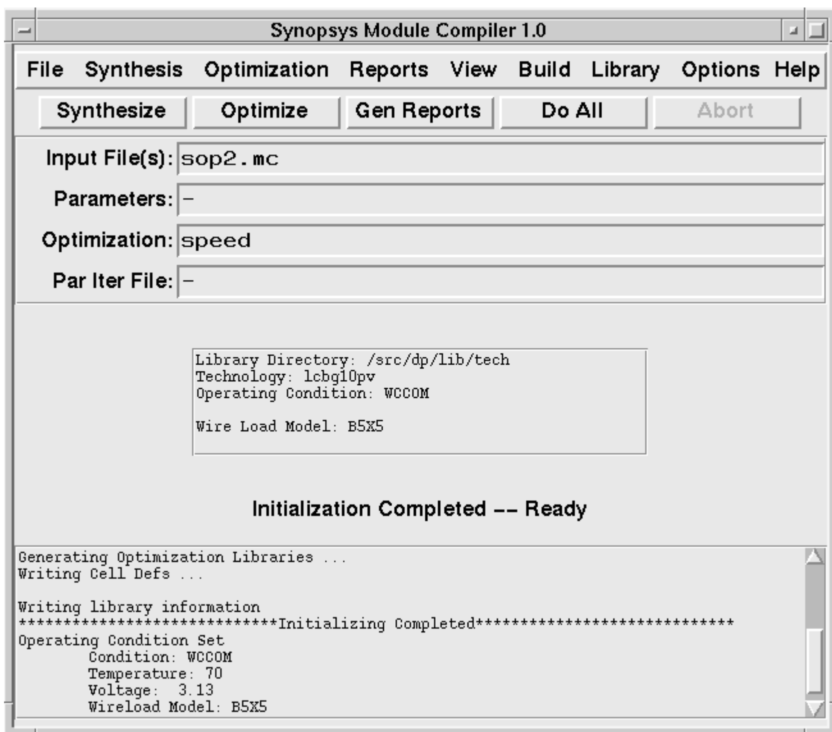
A computational system typically consists of three parts: the computation engine, the control logic, and some storage. The computation engine is built using elements which can be as simple as adders and multipliers or as complex as FIR filters. "Datapath" is the term used to describe these elements and their interconnections. In the context of ASIC technology, "datapath" usually refers to the part of an IC that implements this computation.

Module Compiler (MC) is a tool for designing datapaths for application specific ICs. It is used to build the complex, high performance datapaths that are required by technologies like multimedia, digital signal processing, and communications. Because many controllers are actually constructed from datapath elements, MC is also able to build the control logic in these cases.

Starting Module Compiler

See the “User Quickstart” in Chapter 2 for instructions on how to start using Module Compiler. By default, MC runs in GUI mode. The opening screen is shown below. Chapter 3 describes the graphical user interface (GUI) in detail.

Figure 1-1 The Module Compiler Startup Screen



After you have started MC, the next step is to choose any input files and parameters that are needed. The File menu provides a browser to help you locate files in the UNIX directory structure. You can also enter the file names manually or select them from the set of DPE functions. MC also supports creating a new MC language description. In all cases, the parameter list for synthesis can be set manually or extracted by MC from the input file.

The Command-Line Interface

MC also supports a command-line interface, which is more amenable to script-based automation. The command-line interface provides a noninteractive approach to running MC. Each time MC is started it executes the options provided and then exits. In contrast, the GUI provides an interactive, point-and-click model.

The command-line options for MC are documented in the *Module Compiler Reference Manual*. Most of the command-line options can be set through the graphical user interface. There are some options (such as the technology) that must be specified on the command line when MC is started.

Flow for Building Modules

The sequence of steps in using Module Compiler begins with the design input file and ends with the MC outputs. In between, you may need to complete several iterations to debug and optimize the design. The overall process of running MC consists of the steps listed below. Depending on the objective, you may be able to skip some of the steps in a given iteration. For example, you can skip optimization if the objective is behavioral simulation. If there has been no change since the last iteration, you do not need to set the options again.

1. Start MC. If you want to use a technology other than the one that was used when you last started MC in the current directory, specify the technology with the **-tech** switch at the command line.
2. Select or edit input files, operating condition, and parameters if needed.
3. Set Synthesis options if needed. Synthesize.
4. Set Optimization options if needed. Optimize.
5. Set Reports options if needed. Generate and view reports.
6. Modify MC input file(s) if needed. Iterate (go to step 4).

The synthesis stage consists of specifying the input file, MC libraries, and design processing parameters. MC parses the input file and the libraries. If the parsing is successful, MC synthesizes the design. If the parsing fails or if the synthesis results are not acceptable, you can edit the input file and resynthesize.

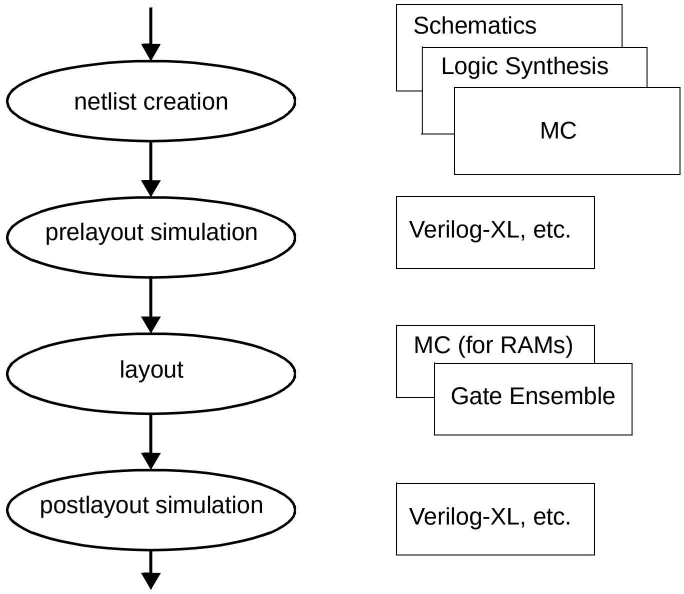
The optimization stage consists of improving synthesis results. If the results of this stage are not acceptable, you can edit the input files and/or specify different options. You must then repeat the synthesis and optimization steps.

The last stage is **results analysis**. During optimization, statistical data about the results is accumulated. During results analysis, you decide the particular data you want to generate and view. After evaluating the design data, you can return to the optimization stage to change optimization parameters or declare victory and integrate the results into your CAD environment.

Module Compiler: Function and Uses

Module Compiler builds high-performance datapaths. MC includes several programs and libraries and supports a GUI as well as a command-line interface. The input to MC consists of a high-level description of the datapath and some design constraints. The output of MC is the synthesized circuit represented by the various model views and report files. The input description is written in the MC language. This language has the look-and-feel of the Verilog hardware description language, but is better suited to the task of describing the synthesis and optimization of datapaths. The design constraints are described through the GUI or embedded in the input description.

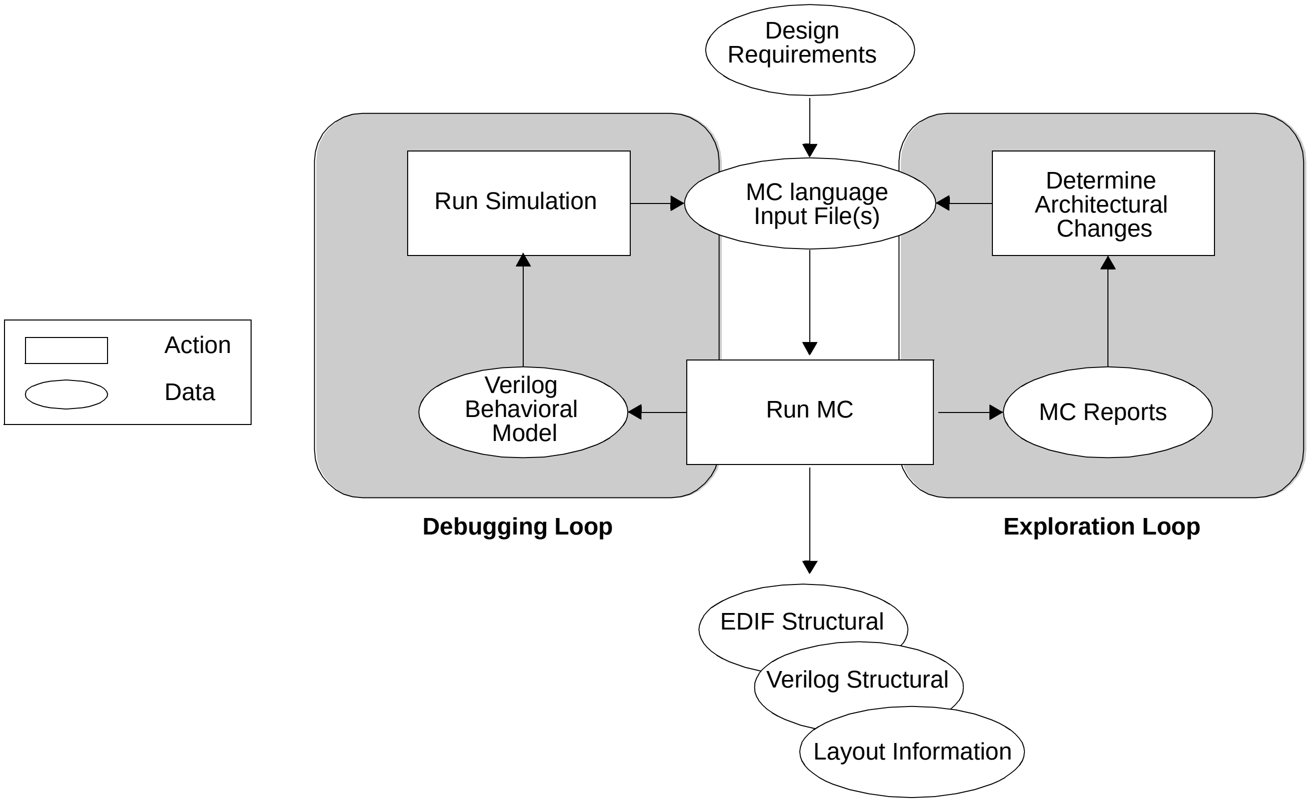
Figure 1-2 Design Flow and Module Compiler



MC supports the full spectrum of datapaths, from highly regular bit-sliced structures to very complex and irregular structures. Bit-slicing is a structured approach suitable for highly regular datapaths. It is not very versatile, however, and highly regular datapaths are becoming more uncommon. Furthermore, many well known, high-speed architectures, such as Wallace trees, are irregular. Irregular datapaths cannot be implemented efficiently using the conventional bit-slicing approach. Module Compiler uses a versatile approach that allows regular and irregular bit widths and is able to trade-off between speed and area.

The interaction of MC with other CAE/CAD tools in a typical design flow is shown below.

Figure 1-3 Process Flow in Module Compiler



This process flow is typical of similar synthesis tools. You usually start by writing the input description in the MC language, using any text editor. The MC language is described in detail in Chapters 4 and 5.

Next, the two primary loops (exploration and debugging) are exercised. These loops are typically interleaved, but you can follow your preferred style. The exploration loop consists of running MC, analyzing the output reports and then modifying the input files to optimize the macro-architecture. Since MC runs quite fast, it is usually possible to run many iterations to achieve the best trade-off of circuit performance, area and power. Performance considerations are discussed in Chapter 5 and Chapter 9. The details of using the Module Compiler GUI are in Chapter 3.

The debugging loop involves Verilog behavioral simulation and ensures that the network description you have provided is correct, and that latency introduced by automatic pipelining is acceptable. Details of the output models, design analysis and control, are provided in Chapter 10.

When the behavior is correct and the costs are acceptable, the design can be implemented. At this point, an EDIF and Verilog netlist is generated for the logic portion of the design. The complete design can now be verified and simulated. The EDIF netlist and the layout information are passed to the downstream tools for place and route and verification of the design.

MC also works with most other ASIC design flows that can accept a netlist in EDIF or Verilog formats.

Building Datapaths

In the context of Module Compiler, a datapath is a network of computational and sequential objects. This chapter provides an overview of the objects and attributes in this network as well as the basic concepts involved in constructing and evaluating the network.

The network objects are used in the input description that is supplied to MC. After synthesis and optimization, MC provides a summary of the network attributes to help in evaluating the original description. The network attributes include timing, area, and power. You can affect these attributes by controlling the synthesis and optimization processes. This can be done by setting constraints and optimization goal or by making architecture selections. MC also provides some control over test and layout.

Synthesis and Optimization

The two primary steps in generating the circuit are synthesis and optimization. Synthesis is the portion of the process in which the high-level input description in the MC language format is converted into a gate level network. After synthesis, optimization is employed to modify the gate level network to improve delay, area, and power.

Hierarchy Through Functions

Hierarchy is generally used to break a large task up into a set of smaller tasks. Most IC designs are very hierarchical because the synthesis tools—general purpose logic synthesizers or the human brain—tend to be very slow when processing large blocks of logic. A common difficulty in designs with extensive hierarchy is that timing problems tend to occur at the boundaries of the hierarchy and are, therefore, often not found until late in the design process. In addition, if you reuse a module, you can optimize it only once, not once for each of its instances. Traditional hierarchy is a trade-off between time and efficiency of implementation. Another factor to consider when reusing designs is that as technologies and cell libraries improve, the fixed design tends to become obsolete.

MC provides a different set of constraints that make the use of extensive hierarchical structure more feasible. For most circuits, the traditional form of hierarchy degrades the quality of the output without significantly reducing the design time. MC addresses this issue by providing a hierarchy in the *idea space* which does not become hierarchy in the final design. You can break your *ideas* into hierarchical segments—functions—in the network description. These are flattened before synthesis and optimization. MC synthesizes and optimizes each instance of the idea for its particular environment. For example, you might find that a counter is needed many times. Instead of creating a cell in the design that is a fixed counter and instantiating it many times, you can create a parameterized function. Each counter is synthesized and optimized independently of the other instances of the counter. The counters in more critical sections of the design are optimized differently than those in less critical sections.

Network Objects

To be able to create a network description for synthesis, you must first understand the objects used in the description. The objects are the *design*, *functions*, *timing groups*, *groups*, *operands* and *instances*.

At the root of the hierarchical tree of objects is the **design** that corresponds to a single synthesized cell or module. MC always creates a design with a single level hierarchy. The design is composed of timing groups, which are maintained by MC.

Timing groups are the set of all user-defined groups that have the same delay goal or desired delay. There is one timing group for each specified delay goal.

A **group** consists of one or more selected operands. All operands within a group must have the same delay goal. Statistics such as area, power, and critical path are maintained for each group and are provided in the design report. You can define as many groups as necessary to understand how the area, latency, and delay costs of the design are distributed. The “misc” group is predefined at the beginning of every design and contains all operands not included in a user-defined group.

Operands represent the signals in the network. Signals can have a signed or unsigned format and can be either constant or variable. *CLK* is a predefined operand for the global clock.

Functions and **operators** connect operands. The function or operator specifies the method used to compute the output operands from the input operands. MC provides an extensive set of basic functions and operators associated with datapath synthesis. These include integer addition, subtraction and multiplication, logical AND, OR and XOR, saturation, shifting, rotation, normalization, comparison, multiplexing, and cycle delays. MC also provides a function for each cell in the technology library. You can instantiate these cells by calling the function. Similarly, any other cell or netlist can be included in the design through an MC-created function.

MC creates **instances** of cells as the result of synthesizing a function or an operator.

Network Attributes

The network attributes provide you with information regarding the costs of implementing the circuit. MC considers timing to be the primary cost: that is, if the delay goal is not met, any amount of power and area can be “spent” to achieve the delay goal. Area and power are secondary costs; they are minimized only after the delay goal is met.

You control the synthesis and optimization processes either by making architecture selections or by specifying constraints. This is generally accomplished through the use of MC directives. MC directives are described in detail in the following chapters.

Timing

Achieving high performance requires that careful attention be paid to the timing during synthesis and optimization. There are two primary components of timing: **continuous time delay** and **discrete time delay** or **latency**. In nonsequential circuits, only the continuous time component is meaningful. In sequential circuits, and especially in DSP-oriented circuits, the latency component becomes a major design issue. In either case, MC provides several mechanisms for optimizing, reducing, and managing these delays.

For sequential circuits, one or more global single-phase rising-edge clocks, available throughout the MC language hierarchy, are supported. Although the clocks are independent, MC uses a simple timing model in which all clocks have no skew relative to each other. In the rare cases that require multiple clocks, you must ensure that the design is not sensitive to the clock skews. The clocks are not buffered by MC. MC assumes that the actual clock distribution is solved during place and route.

MC uses standard Synopsys wire load models for prelayout timing calculations during synthesis and optimization. Obviously, the exact wire lengths are unknown until place and route is completed, so MC must use an educated guess.

Continuous Time Delay

MC provides standard state-independent support for continuous time delays: separate rise and fall delays are maintained for each net and the unateness of the timing arcs is used to generate the most accurate delays possible. To ensure that the delays of all inputs to a function are known when the function is synthesized, all operands must be synthesized before being referenced as an input which has a timing arc to one or more of the outputs. MC sorts the network automatically to guarantee that operands are synthesized in the correct order regardless of the ordering of operands in the input description. MC issues an error when the network cannot be sorted because a continuous time loop is encountered.

One of the primary goals of the synthesizer is to minimize delay, thus maximizing performance. Wallace trees are used extensively to meet this goal, since the final circuit takes into account the arrival times of all inputs and the cells being used in the synthesis. While it is well known that Wallace trees provide the highest performance circuits for multiplication, they can also be used for adders and **AND**, **OR**, and **XOR** logic. In addition, two final adder architectures are provided, which adapt to both the input arrival times and the output delay goal to minimize the area for a given performance level. Other techniques are also employed, such as the optimization of the select inputs of multi-level multiplexor structures to accommodate skewed arrival times.

As noted above, MC's primary goal is minimizing delay. This behavior is sometimes undesirable when delay matching is employed (for example, to meet a hold time) or whenever the delay of the circuit has been purposely increased. MC provides directives to override the default behavior and turn logic optimization off for parts (or all) of the design.

Latency and Registers

MC provides extensive support for controlling and optimizing latency. You can employ state and pipeline registers, automatic and manual pipelining and automatic, manual, or no latency deskewing.

Complex designs require both state and pipeline registers. State registers provide delay that is required by the algorithm being implemented (for example, the accumulator of a MAC is required for the operation of the MAC), while pipeline registers introduce latency that is undesired (for example, to minimize the continuous time delay in a pipelined circuit). The output of a state register has the same latency as the input, while the output latency of a pipeline register is greater than the input latency. MC generates an error if latency is introduced into a loop.

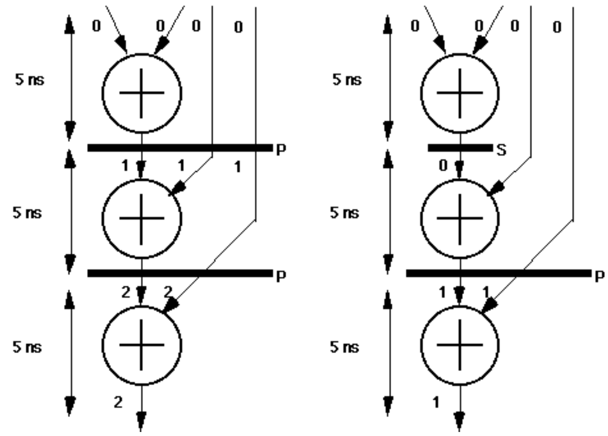
Pipeline registers can be inserted manually or automatically. Automatic mode is commonly used in DSP circuits; the synthesis routines insert pipeline registers whenever the delay exceeds the user-specified cycle time. Pipeline registers can be inserted at *any* point in the circuit, rather than at only a few convenient locations as in other compilers. For example, pipelines can be placed inside a function or operator at any instance boundary.

Pipelining can create latency differences between two or more operands that must be corrected. This process is referred to as *latency deskewing*. Latency deskewing occurs automatically whenever two or more signals with different latencies are connected to the same instance. Signals are delayed so that all latencies are equal to the largest latency. This process can have undesirable results, particularly when sequential loops are involved.

Latency deskewing can also be invoked manually. In general, this technique is used to force multiple outputs (or any other two operands) to have the same latency.

In the examples below, automatic pipelining is enabled with a delay goal of 5 ns and each adder has a delay of 5ns. For simplicity, the setup times and the register delays are assumed to be zero. For the case on the left, pipelines are inserted at the output of each adder to keep the critical path delay within the delay goal. At the input to the second adder, pipeline deskewing is used to delay the fast input to have the same latency as the slow input (latencies are shown next to each signal). At the third adder input, pipeline deskewing is used to delay the fast input by two cycles.

In the case on the right, a state register was inserted manually at the output of the first adder. Automatic pipelines and pipeline deskewing are only employed at the input to the third adder. Note that state register does not cause deskewing and that the final latency is one less for the circuit on the right. Although the registers are shown at discrete points between the adders, automatic pipelining can insert registers inside the adders.



If no special precautions are taken, introducing a signal with latency into a loop causes pipelining inside the loop. This is clearly unacceptable. To prevent this problem, you can suppress deskewing for these signals. Finally, if you need to force the latency of one operand to match that of another, you can use equalization before they interact at the instance level.

Area

MC uses a technology library in Synopsys db format as the basis for synthesis and optimization. In general, MC uses the db area unit for all area measures.

If you are using a CBA technology library, however, MC computes the area taking into account the two types of sections in the array. The CBA architecture is an array of compute and drive sections in a 3 to 1 ratio. The primary measure of area is the total of number of sections (number of drives plus number of computes) occupied. If two area calculations are equal, then the circuit that contains fewer of the scarce sections is considered to be superior. For instance, if the compute-to-drive ratio is less than 3 to 1, then the design with fewer drives is considered better. If a tie breaker is still needed, the number of instances and pins are used in order.

Power

MC uses a simple static power model. The power for each instance is computed in isolation using only the power model for a single cell, the clock frequency and the AC and DC switching factors. The inter-instance effects are ignored; for example, the effect of rise and fall delay of one instance on another is not considered.

Each cell has a power model that includes a DC component, P_{DC}, and an AC component, P_{AC}. The AC component includes the input pin capacitance and associated estimated wire load in addition to any internal AC power. The power for an instance is computed as

$$P_{total} = P_{AC} * F * S_{AC} + P_{DC} * S_{DC}$$

where F is global clock frequency, S_{AC} is the AC switching factor, and S_{DC} is the DC duty cycle. When a cell has no DC component, the power of a cell and that of an instance can be compared quickly, using only the AC component. If a cell has a DC component, the full power equation is used. Notice that in this model the driver is not “charged” with the power required to charge and discharge its load. Instead, each instance is charged only for its contribution to the load. During optimization, cells with lower input capacitance and lower internal power contributions are chosen when possible. You can control the calculation of power by using directives to adjust S_{AC} and S_{DC}.

Designer Control

MC automatically maintains delay, slack, area, and power for each instance, operand and group in the design. In other areas such as macro architecture optimization, MC relies heavily on user input.

Technology and Operating Condition

You can set technology parameters, which can be modified to quickly map a design from one process to another. MC allows you to specify the technology (as supplied by a vendor) as well as the operating condition. You can associate operating conditions with “fast”, “typical”, or “slow” use conditions.

Numeric Representation

You can control the format of an operand. Signed and unsigned formats up to 1024 bits are supported for operands. The operand format is used extensively in the synthesis process because the structures must be adjusted for each format. All signed operands are represented with a 2's-complement representation (the sign bit has a negative significance while all other bits have positive significance). Unsigned numbers are represented in standard binary. The value of these numbers is as follows.

Table 1-1 Signed and Unsigned Formats

Format	Value
signed	$-b_{n-1}\,2^{n-1} + \sum_{i=0}^{n-2} b_i\,2^i$
unsigned	$\sum_{i=0}^{n-1} b_i\,2^i$

The Architecture

You have full control over the macro architecture (the interconnection of user-specified functions) and minimal control over most of the low level details of the architecture (the gross structure used to implement a function) and micro architecture (the interconnection of instances). MC does not optimize the macro architecture; it is always synthesized exactly as described. You are often provided with several choices for the architecture of a given function.

Delay Goal

The design can be partitioned into multiple groups and each group can have a different delay goal. The delay goal can also be specified for the entire design. This delay is used as the current goal when the operands in the group or the design are being synthesized and optimized. If all paths have a delay less than the delay goal, the delay goal is met and the secondary goal (area or power) is pursued.

Note: You cannot set point-to-point path delay constraints.

Automatic Pipelining

You can enable or disable automatic pipelining to achieve the current delay goal. Again, the design can be divided into groups, with some groups pipelined and some not. Pipelining should be used when additional latency can be tolerated to achieve a lower cycle time.

There is no way to specify a latency goal and to determine the delay that results; rather than specifying a latency goal, you must manually iterate by changing the delay goal until the latency goal is met.

Chip Level Mode

There are two primary operating modes: *chip level* and *subchip level*. The mode is controlled by the **Top Level Mode** option. When this option is enabled, the current design is assumed to be a complete chip, containing I/Os. Each input, output, or inout in the module must connect to exactly one PAD connection and one I/O buffer of the correct type. When **Top Level Mode** is disabled, the design is assumed to be a sub-chip module. There should not be any I/O buffers in this type of design. MC generates warnings when any of these rules are violated.

Automatic Buffering

All synthesized functions utilize automatic buffering internally to prevent overloading. Overloaded nets have underestimated delays that can result in poor pipelining performance and can generally reduce the quality of timing-driven synthesis. You can manually assign a specific buffer depth to an operand.

Logic Optimizer

You can enable and disable logic optimization for specific portions of the design. Typically, you disable logic optimization when inserting a cell or netlist that utilizes complex or unusual timing into the design, or when you don’t want to minimize delays. For example, you can insert a delay element into a RAM address path to ensure that the hold time requirement is met. Disable logic optimization locally to prevent the removal of the delay element. Area and performance will suffer if logic optimization is disabled for large portions of the design.

Optimization

MC supports the following four optimization criteria:

Table 1-2 Optimization Criteria Categories

Criterion	Effect
timing, size	achieve delay goal at any cost, then minimize area
timing, power	achieve delay goal at any cost, then minimize power
size	ignore timing, try to minimize area
power	ignore timing, try to minimize power

The first two criteria are the most commonly used. The delay goal is used as the primary optimization criterion and either area or power are optimized secondarily. The last two criteria are the same as the first two with the delay goal set very large. These two cases are of limited value, since they are likely to generate circuits with very large delays.

Clocks

MC supports the use of one or more single-phase clocks that are active at the rising edge for all sequential circuits. In addition, these clock are assumed to be globally buffered, hence MC does not insert local clock buffers. This approach is consistent with ASIC design methodologies that cannot implement multiple clocks with very low skew.

A pure combinatorial design has no clocks, while sequential circuits typically have a single clock, named *CLK*. In some cases, the design is partitioned into groups with different clocks. The use of multiple clocks is highly restricted. Automatic latency skewing is not permitted between signals generated with different clocks. If signals from different clocks are pipelined before interacting, latency hiding must be employed. Also skews between clocks are ignored and you must understand that timing information provided by MC might not be accurate in all cases.

The default clock signal is *CLK*. A clock trunk or a clock buffer tree is inserted during place and route.

External Constraints

The influences of external circuits at both the input and output of the synthesized circuit are supported by MC through external constraints. At the input, the maximum allowed load can be specified for each input operand to accommodate loading constraints of the driver. Arrival times can also be specified to represent any delay incurred in the external circuit. At the output, a load can be specified to represent the input loading of the following circuit. A delay can be specified to represent delays expected by the following circuit.

Testing

MC supports scan test methodologies implemented in a third party tool. MC does not wire the scan chain or generate the test vectors; it only attempts to anticipate the changes that will be made when the scan chain is inserted. This approach allows the scan chain and test vectors to be generated more globally.

When operating in scan mode, all simple and enabled D-type flip-flops are converted to scan registers during synthesis. This ensures that the correct area, timing and power estimates are used during synthesis and optimization. After the design report is written, but before the netlist is written, an attempt is made to convert the scan flip-flops back to D-types. If any other flip-flop types were included in the design, a warning message is generated. Both synthesized and instantiated flip-flops are supported.

Naming

You can control the verbosity of instance and net names using the Use Groups Names item on the Synthesis menu and the Sim Debug Mode item on the Reports menu. These names are meaningful in both layout and simulation. See “Naming” in Chapter 10 for a discussion of naming issues in MC.

Layout

Module Compiler provides detailed placement information that can be used to control the placement of instances in the design in a variety of placement approaches. The entire datapath can be bit-sliced, bit-stacked, or floorplanned, or a combination of the these approaches can be used. You can choose any technique for each block of the design, based on high-level floorplanning constraints and the complexity and regularity of the design. See Chapter 8 for a detailed discussion of using the layout information provided by MC.

Degenerate Cases

To improve productivity, MC handles degenerate cases efficiently. Several types of degeneration are handled, including missing data and constants.

The missing data case is the most important because it is the most common. All Wallace tree-based functions tolerate any number of inputs, including zero, in any bit position. There is no need to worry that the sum of one input results in an adder with one signal input and another input tied to zero. In addition, the use of bit ranges and constant shifts can cause missing data. Again, the structures adapt to the missing data to create the smallest and highest performance structure.

Constants are also handled efficiently while being interchangeable with normal variable signals. Many synthesis functions optimize the constants as a special case, providing the greatest optimization. For example, multiplying two constants results in no instances, while multiplying a variable signal by a constant, results in a smaller, faster circuit than a 2-variable signal multiplier. Even partially constant signals (those with some bits which are variable and some which are constant) can be optimized.

Installation and Setup

2

This chapter describes how to install the Module Compiler software and how to set up the user and group environments.

Chapter 2 discusses the following topics:

- Platform requirements
- User Quick Start
- Installation instructions
- System administration information
- Instructions for building pseudo-cell libraries

Platform Requirements

The graphical user interface for MC requires the X Window system. The command-line interface for MC can be used in any terminal environment. Other specific platform requirements are outlined below.

- SUN-Sparc workstation; SunOS 4.1.3
- Main Memory: 64MB
- Swap Space: 250MB
- Disk Space: 20MB

User Quickstart

To get started as an end user of Module Compiler, follow the steps in this section. If you are functioning as the administrator and need to install and maintain Module Compiler, follow the steps given in the "Installing Module Compiler" and "System Administration" sections.

First Run

The following steps assume that Module Compiler has been properly installed. To run Module Compiler for the first time:

1. Create a clean directory. In the example below, the directory is `mcproj`:

   ```
   % mkdir mcproj
   ```

   ```
   % cd mcproj
   ```

2. Initialize your UNIX environment. In most cases, the administrator will have put the necessary path information into the `setup.csh` file, so you type the path to that file; for example:

   ```
   % source /mc1.0/localadm/setup.csh
   ```

   This sets the variables that point to the MC program and to the directories containing the technology libraries.

3. Start Module Compiler using the -tech switch to specify the technology library you want to use. MC will not run without a technology library.

   ```
   % mc -tech XYZ
   ```

   This loads the specified library and runs Module Compiler in GUI mode.

4. To test MC, click the Do All button. Module Compiler should build an 8-bit adder, showing its progress in the Status Area and in the Log Window. See Chapter 3 for information about using MC's GUI.

Subsequent Runs

When you run MC for the first time, it creates an `mc.env` file in the directory where you started it and stores all your settings in it, both those set from the command line and those set in the GUI. When you start MC again *in that directory*, MC reads the settings and starts up in the previous configuration. You can override these settings with command-line switches.

Installing Module Compiler

This section and the following sections of this chapter are intended for use only by system administrators who need to install and maintain Module Compiler. End users can ignore the remainder of this chapter.

1. During installation, set umask to

   ```
   % umask 22
   ```

   or

   ```
   % umask 2
   ```

2. Determine where you would like to install the software and create a new directory in that location. These instructions assume that the location is `/mc1.0`.To create the new directory, type:

   ```
   % mkdir /mc1.0
   ```

3. Change directory to the new location:

   ```
   % cd /mc1.0
   ```

4. Load the contents of the tape into this directory. The following is an example `tar` command for loading the tape:

   ```
   % tar xvf /dev/rst0
   ```

5. Execute the following commands:

   ```
   % set x = (‘\ls | grep -v localadm‘)
   % chmod -R a-w $x
   % unset x
   % mkdir tech
   ```

6. `localadm/setup.csh` is a source script that sets up the UNIX environment for users of the “csh” shell. Change the MCDIR variable in this file to reflect the actual location chosen in Step 2 above. Type in a complete path: do not use the “~” character in the path.

7. `localadm/setup.csh` assumes that the UNIX script `/bin/arch` correctly returns the platform name, such as sun4. If this is not the case, change the character sequence `‘/bin/arch‘` to the name of the platform (sun4, for example).

8. Execute the following command:

```
% \ls -lF
```

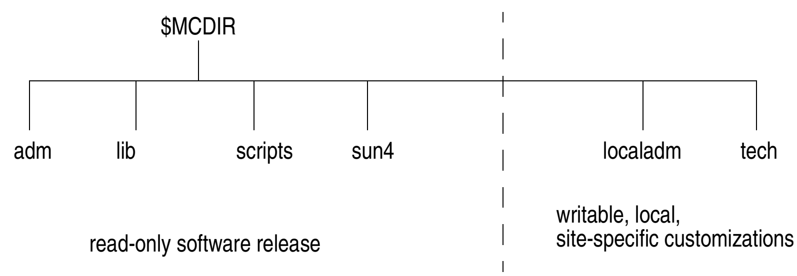
A successful installation should look something like this:

```
total 6
dr-xr-xr-x  2 root  512 Oct 10 20:14 adm/
dr-xr-xr-x  6 root  512 Oct 10 20:14 lib/
drwxrwxr-x  2 root  512 Nov  5 01:53 localadm/
dr-xr-xr-x  2 root  512 Oct 10 20:14 scripts/
dr-xr-xr-x  3 root  512 Oct 10 20:14 sun4/
drwxrwxr-x  2 root  512 Nov  5 14:39 tech/
```

System Administration

Once installation has been successfully completed, you should see a directory structure similar to the following one:

Figure 2-1 MC Directory Structure



The read-only portion of the directory tree should be preserved in its original state without any modifications.

- The `localadm` directory is the place for local administration and setup files.
- `localadm/setup.csh` is a source script that you can use to initialize your UNIX C shell environment.
- `localadm/mc.env` is a file that contains site-specific settings for MC environment variables.
- The `tech` directory is where technology-specific library files should be kept, including the Synopsys "db" libraries. This is not required, but is convenient. When MC creates pseudo-cell libraries, they are written into the `tech` directory.

UNIX Environment Variables

Module Compiler uses a few UNIX environment variables. Most of these variables can be initialized using the `localadm/setup.csh` source script. Table 2-1 identifies and defines the various UNIX environment variables used by Module Compiler.

Table 2-1 UNIX Environment Variables

Type	Variable	Description
MC	MCDIR	The path name of the software installation point. This is the Module Compiler root directory location. *Required.*
	MCLIBDIR	The path name of the technology library directory. This directory holds all the technology libraries for MC including any pseudo-cell libraries. *Required.*
	MCTECH	The name of the current technology. This variable is NOT initialized in `localadm/setup.csh`. *Optional. The dp_tech mc.env variable has priority if it is defined.*
	MCENVDIR	In addition to the Unix environment variables, there are a number of Module Compiler environment variables that are specified in `mc.env` files. These are not Unix variables. The MC environment variables are described in the *Module Compiler Reference Manual*. MCENVDIR is a list of path names to directories that contain `mc.env` files. The directory "./" is implied at the beginning of the list. The priority is decreasing from left to right so that the variables set in the working directory have the highest priority, followed by the other directories given in the list. *Optional. MC uses $MCDIR/adm if MCENVDIR is not defined.*
TCL	TCL_LIBRARY	The path name of the Module Compiler TCL library directory.
	TK_LIBRARY	The path name of the Module Compiler TK library directory.
License	SYNOPSYS	See Design Compiler installation.
	SYNOPSYS_KEY_FILE	See Design Compiler installation.

Technology-Specific Module Compiler Variables

Module Compiler has a number of environment variables that are specified in `mc.env` files. These MC environment variables are described in the *Module Compiler Reference Manual*. Some of these variables have technology-specific versions. A technology-specific MC environment variable has the technology name appended to the normal variable name. Table 2-2 identifies and defines the technology-specific MC variables. "XYZ" is used as a placeholder for the technology name. When a variable has both technology-independent and technology-specific versions, the technology-specific version has the highest priority.

Table 2-2 Technology-Specific MC Environment Variables

Variable	Description
dp_tech_lib dp_tech_lib_XYZ	A comma separated list of db files. The file names must include the full path name. These db files comprise the technology library.
dp_dc_wireload dp_dc_wireload_XYZ	This variable is the named wire load model from the technology library.
derate_slow_named_opcond derate_slow_named_opcond_XYZ	The named operating condition from the technology library that is used when the operating condition is **slow**.
derate_typ_named_opcond derate_typ_named_opcond_XYZ	The named operating condition from the technology library that is used when the operating condition is **typ**.
derate_fast_named_opcond derate_fast_named_opcond_XYZ	The named operating condition from the technology library that is used when the operating condition is **fast**.

Using the `mcenv` Program

You can use the `mcenv` program to set and query MC environment variables. When you set an MC variable, `mcenv` stores the value in the `./mc.env` file. When you query the value of a variable, `mcenv` first checks the `./mc.env` file for the MC variable. This ensures that the working directory has the highest priority. Next, it check directories in the `MCENVDIR` list from left to right until the variable is located. This is the same mechanism that Module Compiler uses when it checks for variable values.

To set an MC variable, specify the variable name and its new value as shown in the following example:

```
% mcenv dp_opcond slow
```

This sets the value of the *dp_opcond* variable to **slow**.

To query the value of an MC variable, specify the variable name as shown in the following example:

```
% mcenv dp_opcond
```

This returns the value of the *dp_opcond* variable.

To query the value of an MC variable that has a technology-specific version, use the **-tech** switch:

```
% mcenv -tech dp_tech_lib
```

This returns the value of the *dp_tech_lib* variable with the highest priority. Since all technology-specific variables have higher priority than technology-independent variables, `mcenv` returns the technology-specific version for the current technology if one exists.

Customizing Module Compiler Environment Variables for All Users

Module Compiler has a number of environment variables that are specified in `mc.env` files.

Note: In practice, you need to set only a few of these variables. In fact, you may never need to set any of these variables. The software installation stores default values for all of these variables in the **`$MCDIR/adm/mc.env`** ***file. The system administrator can override these default values for all users by setting MC environment variables in the*** **`$MCDIR/localadm/mc.env`** ***file. This is a convenient way to set preferences for the entire group.***

The technology-specific MC variables are the variables that the system administrator most commonly needs to manage in the `$MCDIR/localadm/mc.env` file. To initialize the technology-specific MC environment variables, follow the steps below. These instructions assume that the software installation point is `/mc1.0`, and that the technology name is “XYZ”.

1. Initialize your UNIX environment. For example:

   ```
   % source /mc1.0/localadm/setup.csh
   ```

2. Change directory to the `localadm` directory:

   ```
   % cd $MCDIR/localadm
   ```

3. Execute the following commands (where “XYZ” is the technology):

   ```
   % mcenv dp_dc_wireload_XYZ <my_wireload>
   % mcenv derate_slow_named_opcond_XYZ <my_worst>
   % mcenv derate_typ_named_opcond_XYZ <my_typical>
   % mcenv derate_fast_named_opcond_XYZ <my_best>
   ```

4. Set the *dp_tech_lib_XYZ* variable based on the location of the library files for the XYZ technology. Assume that the XYZ technology has two db library files. If these files are located in the `$MCDIR/tech` directory, execute this command:

```
% mcenv dp_tech_lib_XYZ ’(MCLIBDIR)/XYZ.db,(MCLIBDIR)/XYZ_wires.db’
```

If the db files for the XYZ technology are located in the `/my/dbs/go/here` directory, execute this command:

```
% mcenv dp_tech_lib_XYZ /my/dbs/go/here/XYZ.db,/my/dbs/go/here/XYZ_wires.db
```

Licensing

This program requires an MC-Pro version 1.0 license key. Module Compiler uses the standard Synopsys floating license manager.

Building Pseudo-Cell Libraries

To build a pseudo-cell library for a given technology, the system administrator should follow these steps. These instructions assume that the software installation point is `/mc1.0`, and that the technology name is "XYZ."

1. Initialize your UNIX environment. For example:

   ```
   % source /mc1.0/localadm/setup.csh
   ```

2. Execute the following commands:

   ```
   % cd $MCLIBDIR
   % makeMcLib
   ```

   The `makeMcLib` program displays its usage as follows:

   ```
   usage: makeMcLib <tech> [<wireload>]
   by default 2.5 load per fanout is used
   ```

3. Determine the most commonly used wire load model. The pseudo-cell library will be built and characterized using this wire load model.

4. Execute `makeMcLib` for your technology and wire load model. This example builds the pseudo-cell library for the XYZ technology, using the "wires_15K_used" wire load model.

   ```
   % makeMcLib XYZ wires_15K_used
   ```

3

Using the Module Compiler Graphical User Interface

This chapter describes how to use the MC graphical user interface (GUI) to build datapaths. The focus is on how to use the GUI effectively and how the many parts of the program interact.

Chapter 3 discusses the following topics:

- A description of object types in the GUI
- An overview of the MC GUI
- File manipulation and sessions
- A brief discussion of each menu item
- Report generation and viewing

See the “User Quickstart” in Chapter 2 for instructions on how to start and run Module Compiler. Chapter 2 also contains instructions for installing MC and configuring the environment.

GUI Objects

Windows in the MC GUI interface contain the following types of objects:

Action Buttons

These buttons have an action associated with them. To execute the action, position the mouse pointer over the button and click the left mouse button, sometimes referred to as MB-1 in this document.

Edit Fields

Edit fields allow you to enter some text in a form. Table 3-1 lists the keyboard commands you can use to move the cursor and edit text in fields. You can also use the right and left arrow keys to move the cursor within fields.

Table 3-1 Keyboard Shortcuts for Editing

Key Sequence	Action
Control-b or ←	move the cursor left one character
Control-f or →	move the cursor right one character
Control-a	move the cursor to the beginning of the line
Control-e	move the cursor to the end of the line
Control-d	delete one character to the right of the cursor
Control-h	delete one character to the left of the cursor
Control-i	insert a tab
Control-w	delete the selected text
Control-k	delete text from the cursor to the end of the line
Control-u	delete entire line
left click	change insertion point
press and drag with the middle button	scroll the text

Toggle Buttons

These buttons store binary (on /off) values. To change the state, position the mouse pointer over the button and left click. The color of the button changes to show its state.

Text Windows

These are scrollable windows that display uneditable text. When the window is independent—not part of the main window—the Find Top and Done buttons are present. Select Done to remove the window and Find Top to bring the main window to the top of the window stack.

Dialog Boxes

These are transient windows that allow you enter some requested information. The windows pop up over the current display and are removed when you click OK or Cancel. Select OK to accept the current changes and Cancel to discard any changes. You must dismiss a dialog box before you can continue working in MC.

Error Windows

These are transient windows which indicate an error has occurred and an action is required. The windows pop-up over the current display. The windows are removed when you click the OK button. Click the Find Top button to force the lost main window to the top of the window stack. When the location in one of the input files can be determined as the source of the error, a Edit button is provided. Clicking the button invokes the editor at the location of the error. These windows can be left open when performing other operations with MC, but are removed automatically when the circuit is resynthesized.

Menus

These objects present a drop-down list of items when clicked with MB-1. To choose an item from the menu, click it with MB-1. Once you choose a menu item, the original menu is dismissed.

Tearing off a menu. You can keep a menu displayed by pressing and dragging the menu title with the middle mouse button. This “tears off” the menu. It becomes a separate item, and you can now drag it anywhere on the screen. You can tear off a cascaded menu by dragging its parent item with the middle button. To dismiss a torn off menu, left click on the title or menu choice that displayed it. This is a convenient way to turn menus into a pseudo dialog box when many options need to be changed.

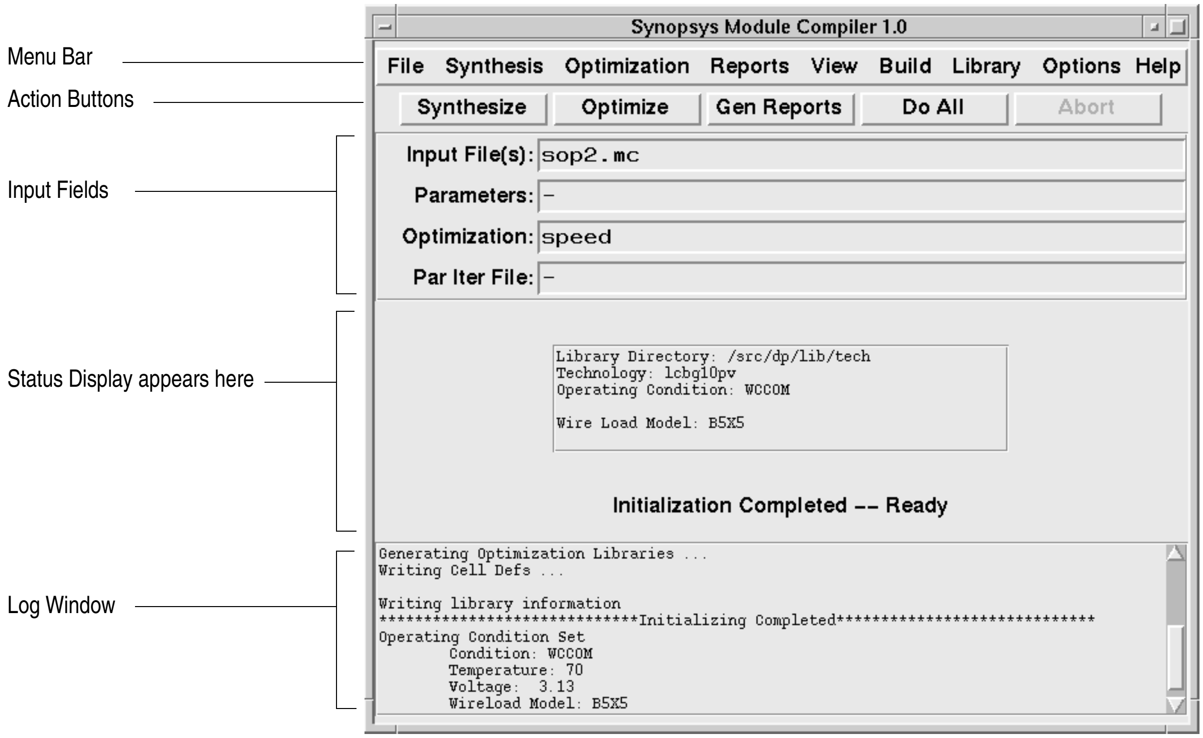
Cascading Menus

A right-pointing arrow in a menu identifies a cascading or submenu. The cascading menu appears when the cursor passes over the parent menu item.

Graphical User Interface Overview

The GUI consists of a permanent main window, transient dialog boxes, and text and error windows. The main window, shown in Figure 3-1, displays when MC is started in graphical mode. It consists of the menu bar, the action buttons, the input fields, the status display, and the log window. It is designed to make the most important information and options easily available, so that numerous pop-up windows are not needed.

Figure 3-1 The Module Compiler Main Window



Menu Bar

You use the menu bar primarily to select files to synthesize, to initiate an action, to get online help, to control the GUI environment, and to set options for synthesis, optimization, and report generation. The menu bar dims when the menus are not available.

Action Buttons

The action buttons used to start a step in MC are **Synthesize**, **Optimize**, **Gen Reports**, and **Do All**. The **Abort** button aborts a step in progress. Action buttons dim when the function is not available.

Input Fields

Use the input fields to specify which files to compile, the current parameters to use during synthesis, and the optimization criterion to use during synthesis and optimization.

The Status Window

The status window is used to indicate the progress of the current step and to display library and operating condition information.

The Log Window

The log window is a text window embedded within the main window. It contains a running log of all operations. The scroll bars can be used to review messages that have scrolled out of view. This window can be resized with the mouse and is cleared automatically with each synthesis operation.

Getting Help

MC provides a simple help mechanism. A list of topics can be found in the Help menu. Selecting any help item activates a text viewing window for that topic.

Action Buttons

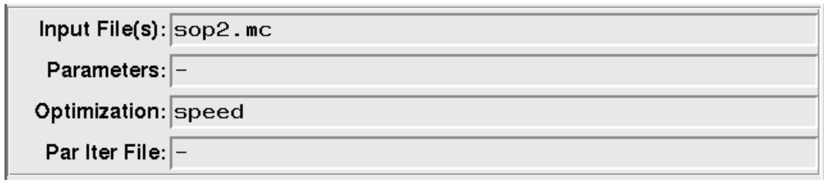
The **Synthesize**, **Optimize**, **Gen Reports**, and **Do All** buttons are used to initiate an action in MC. **Synthesize** and **Optimize** cause the circuit to be synthesized or optimized. **Gen Reports** generates the reports that you have selected in the Reports menu. To see a report, select it from the View menu. **Do All** causes all three operations to be performed in order and is a convenient way to generate the reports after making an input file or parameter change. When Module Compiler is busy, the action buttons dim and the **Abort** button activates to allow you to interrupt some processes.

These actions can also be accessed from the Build menu.

Choosing an Input File, Parameters, and Optimization Criterion

Use the input area of the main window to set which files to synthesize, the parameters for synthesis, and the synthesis optimization criterion.

Figure 3-2 The Input Area of the Module Compiler Window



The following items are available.

Input File

Enter the name of the design description files that describe the module to be synthesized. You can enter the file name or use the Find Input File option in the File menu to open a browser to select the files. Specify multiple files as a comma-separated list without any space between entries.

Parameters

Use this field to specify input parameters. These are parameters that are expected by the module in your input. For instance, if the module has an integer parameter called width and you would like to pass in 8 as the value, then this edit field should contain the following string:

```
width=8
```

If there are no parameters, then the value of this field is “-”. To specify more than one parameter, separate the parameters with commas:

```
width=8,name=test
```

Character strings are allowed as parameters, but must contain only nonnumeric characters. The parameter list must not contain any spaces

To retrieve the parameters and any defaults from the current input file, use the Get Parameters option in the File menu.

Optimization

This field is used to specify the optimization criterion. It should contain one of the values in Table 3-2. In these values, **delay** is an integer representing the delay goal in picoseconds. You can override this value by using the **delay** directive in your design description.

Table 3-2 Optimization Criterion Values

Value	Effect
speed	try to generate the fastest circuit possible
size	ignore timing, minimize area
power	ignore timing, minimize power
speed, size	same as speed, but consider size when breaking ties
speed, power	same as speed, but consider power when breaking ties
<delay in ps>	try to achieve the specified delay in picoseconds
<delay in ps>, size	same as delay, minimize size when there is slack
<delay in ps>, power	same as delay, minimize power when there is slack

Par Iter File

Enter the name of the parameter iteration file that contains the sets of parameter values to be used for synthesis. See the *Module Compiler Reference Manual* for a detailed description of the *param*() function and the parameter iteration file.

File Manipulation and Sessions

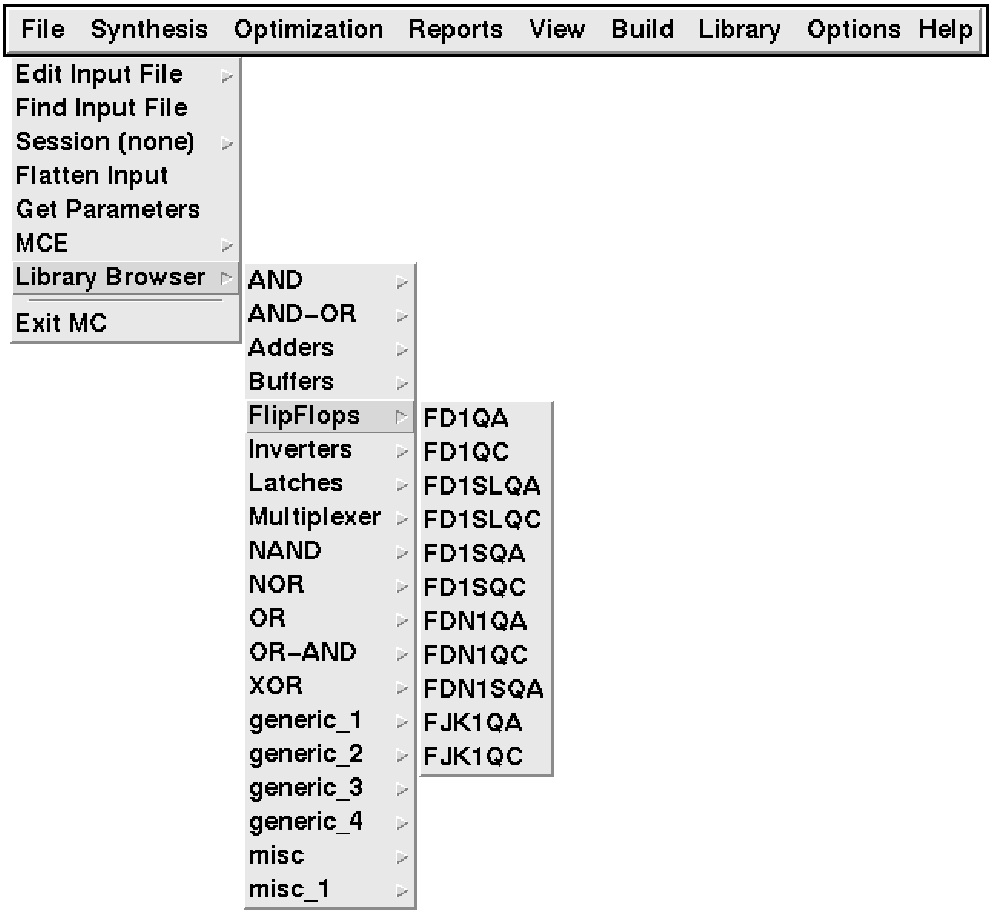
The File menu provides several shortcuts for selecting, editing, and retrieving the parameters from the input files and for manipulating MC sessions.

A session is the set of all settings in the GUI, including synthesis, optimization, and report options, in addition to preferences for the GUI. You can define as many sessions as you like. For example, you can define a session for each block of the design and one for the top level. You can use one session for each parameterization of a block, or you can define a session for a quick estimate and another for a full optimization of the same block. You can even choose to totally ignore sessions altogether.

MC warns you if you attempt to discard any changed settings by exiting MC or loading a new session. When MC is restarted, it automatically reloads the last active session.

You can start MC with a particular session by using the **-ses** command line option followed by the session name. If “**-**” is used for the file name, MC is started without a session. You may want to do this if you like to set options on the command line or if you have no use for sessions. The startup session normally supersedes most of the command line options.

Figure 3-3 The File Menu



The File menu contains the following items for manipulating the input files and options settings.

Edit Input File

To edit an input file, choose Edit Input File from the File menu, and choose the input file to edit.

Note: The default editor is vi. You can change this default by setting the `dp_editor` ***MC environment variable. For instance, you can set your default editor to emacs by executing the following command:***

```
mcenv    dp_editor emacs
```

Find Input File

Opens the file browser to locate a file to edit. When a file is selected, it is appended to the list of files in the Input File(s) entry area.

Session (<current session>)

Selects a session operation from the cascading menu. You can choose to save the current settings under the current name with Save, or a new name with Save As. A dialog-box is opened when selecting Save As to enter the session name. Enter either the session name or use a file name with a `.dps` extension. Use Load to select one of the already saved sessions listed in the cascading menu.

Flatten Input

Removes all macros, function calls (except library functions), replicates, conditions, and integers. You can also see how temporary variables are created and declared for complex expressions. The flattened output is displayed in the log window. This mode can be used to better understand how the description was broken into a set of synthesizable expressions and functions.

Get Parameters

Retrieves the parameters with any default values from the current input file. This option is useful if the design contains many parameters which are difficult to remember. Virtually all errors which occur during parsing are ignored.

MCE

Selects an MCE (Module Compiler Express) function for synthesis. Use this option for quickly generating blocks already available in MCE without having to write any MC language code. Parameters entered for MCE functions are saved automatically on a function-by-function basis and are recalled automatically when the function is selected later. Help is available for all MCE blocks.

Library Browser

Shows all cells available in the currently loaded technology library, plus all foreign cells and netlists loaded from the MC command line. Cells are grouped by categories, with the user netlists and foreign cells located in the “misc” category. When you select a cell in the library browser, the status area of the main window displays the interface, are, and function (if available) for the cell. Each cell or netlist is available within MC as a function with the interface shown.

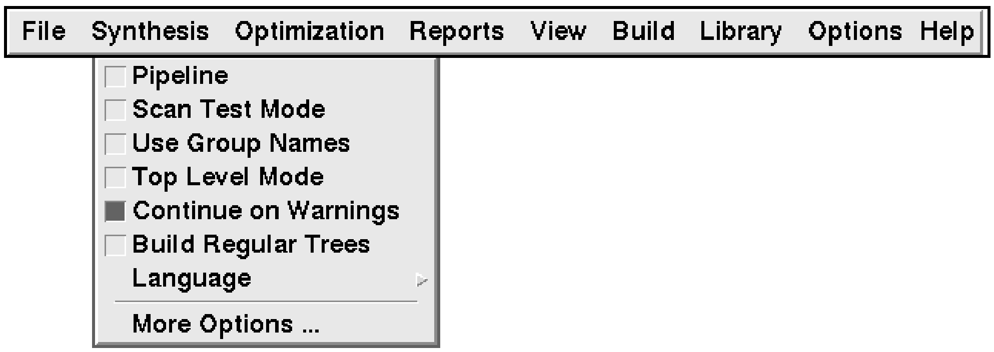
Exit MC

Exits MC. If some session settings or the network has changed with out being saved, you are warned before MC exits.

The Synthesis Menu

After the input files have been selected or edited and the parameters have been set, the synthesis options can be set before the circuit is synthesized.

Figure 3-4 The Synthesis Menu



You can set the following options from the Synthesis menu:

Pipeline
Enables/disables the automatic pipelining default. You can override this value using the **pipeline** directive in the input description.

Scan Test Mode
Enables/disables scan-test mode. For more information on scan test mode, read “Stalling and Scan Test” in Chapter 5.

Use Group Names
Toggles whether to prefix instance names with the name of the group to which they belong. The longer names make it easier to debug and floor plan the results.

Top Level Mode
Indicates whether the design is a full chip that contains I/Os. I/O connection rules are checked in both modes.

Continue on Warnings
Toggles whether MC interrupts the synthesis process when warning conditions are encountered. In general, it is a good idea to stop when a warning condition is detected.

Build Regular Trees
Toggles whether MC should try to maximize the regularity of structures used to build various operators during synthesis.

Language

Displays the Strict Parsing submenu for controlling options to the language parser.

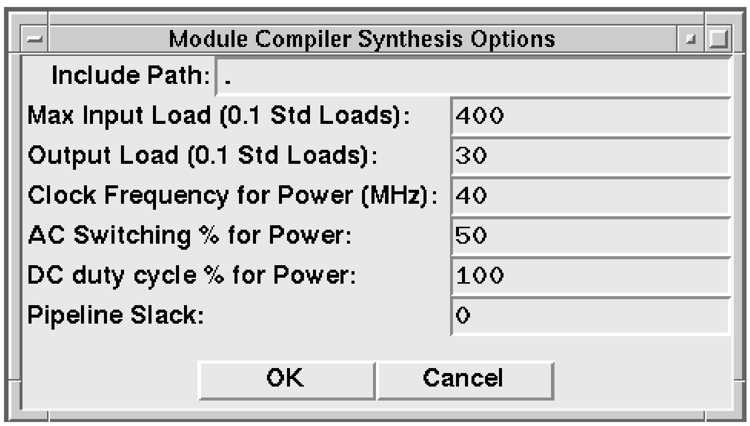
Strict Parsing

Toggles whether to display warnings when obsolete constructs are encountered in the MC input file, and when size or format mismatches occur in function calls. It is a good idea to leave this option enabled.

Synthesis Options

The More Options item on the Synthesis menu displays a dialog box (Figure 3-5) in which you set defaults for a number of synthesis options. You can override these values using a directive statement in an input file.

Figure 3-5 The Synthesis Options Window.



The Synthesis Options window contains the following items for controlling the synthesis process.

Include Path

Sets the search path for any files included by the input file. Normally you set this field to dot (.) to indicate current directory. You can specify an alternate list of directories. Each item in the list should be separated by a colon (:). If your design includes any files, MC searches these directories in sequential order.

Max Input Load

Sets the default value for maximum input loading. You can override this value by using the **inload** directive, described in “I/O Constraints” in Chapter 5. Units are 0.1 standard loads.

Output Load

Sets the default value for the external loading on the output. You can override this value using the **outload** directive described previously. Units are 0.1 standard loads.

Clock Frequency for Power

Sets the clock frequency in megahertz. The value is related to the **acswitch** and the **dcduty** directives for power computation only, and does not affect the delay goal. The maximum allowed frequency is determined by the library being used.

AC Switching Percent for Power

Sets the estimated percentage of nodes switching in each cycle as the initial value of **acswitch**. (The clock cycle time is set using **Clock Frequency.**) Positive integer values are allowed, with 100% representing the maximum toggle rate. This value affects only power calculations.

Design Compiler Duty Cycle Percent for Power

Sets the estimated percentage time any DC power-consuming blocks are active as the initial value of **dcduty**. Positive integer values are allowed, with 100% representing the always active. This value affects only power calculations.

Pipeline Slack

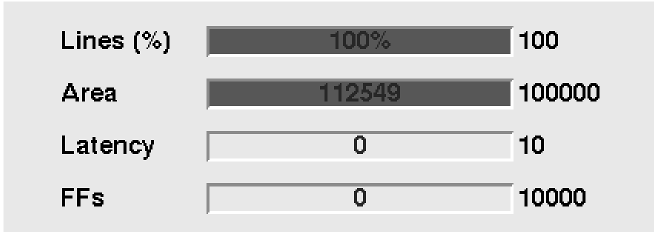
Sets the quantity of slack available for automatic pipelining. It is useful in those cases where the delay goal cannot be met by forcing pipelines to be inserted closer together (for positive values). The pipeline slack is specified in picoseconds. A positive value forces the pipelines closer together whereas a negative value forces the pipelines further apart.

Synthesis Status Display

The progress during synthesis is displayed on a series of thermometers, shown in Figure 3-6. The **Lines (%)** thermometer indicates how much of the flattened input file has been processed. The **Area**, **Latency**, and **FFs** thermometers indicate the number of sections, the maximum latency, and the number of flip-flops currently in the design. To find indications that an error exists in the parameters of the input files, look for values that are far from expectations.

To set the maximum limits for the thermometers, choose Options in the menu bar.

Figure 3-6 The Synthesis Status Display

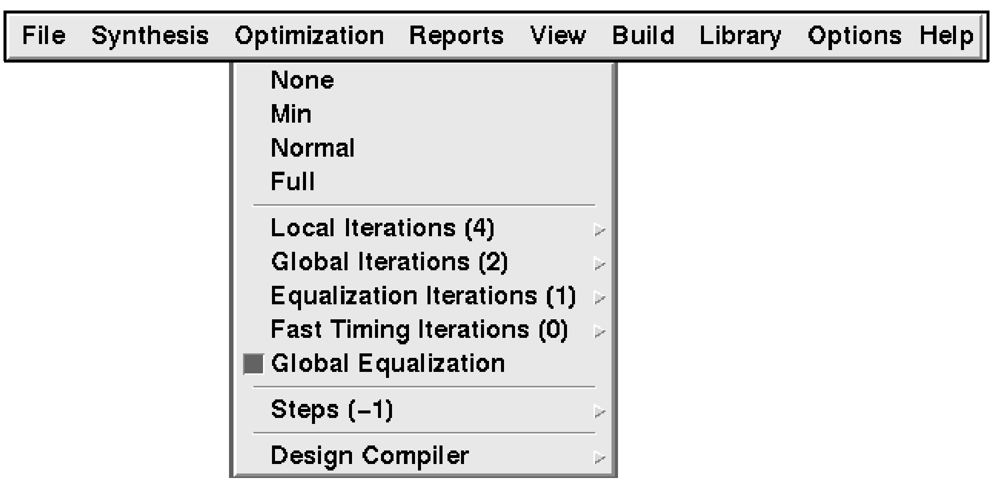


You can click the **Abort** button to stop synthesis. Obviously, the network is not complete when synthesis is interrupted; no optimization or report generation is initiated until the circuit has been resynthesized. MC ignores abort commands that are issued during input file conversion.

The Optimization Menu

Once synthesis is complete, you can optimize the circuit. Use the Optimization menu to control how hard MC tries when optimizing the circuit. There are several ways to specify the optimization level. Choosing one of the four quick choices (**None**, **Min**, **Normal,** or **Full**) sets values for Local Iterations, Global Iterations, Fast Timing Iterations, and Equalization Iterations. You can also set the values individually, or you can use the quick choice to set them and then modify individual settings. The current value for each iteration type is displayed in parentheses next to the menu item.

Figure 3-7 The Optimization Menu



The Optimization menu contains the following items for manipulating the optimization process.

None, Min, Normal, or Full

Choosing one of these items sets values for Local Iterations, Global Iterations, Fast Timing Iterations, and Equalization Iterations. The values assigned appear in parentheses next to the menu items. For most purposes, choosing one of these four items (None, Min, Max, or Full) is sufficient. Selecting None bypasses optimization by setting all four values to 0, and selecting Full optimizes a great deal. Normal is a good choice for most circuits. Modify the preset values by selecting the appropriate menu item from the second section of the Optimization menu.

Local Iterations

Sets the maximum number of local iterations allowed for each step. This is the maximum number of times a step is tried before going to the next step. An optimization step is terminated if no progress is made.

Global Iterations

Sets the number of global iterations performed. All selected optimization steps are performed as a group the number of times indicated. Global iterations always continue until all global iterations have been performed regardless of whether or not progress is made.

Equalization Iterations

Sets the number of global iterations (at the end of the process) that employ equalization. Equalization allows the use of a relaxed delay goal (the current critical path) rather than the original goal when the original goal cannot be met.

Fast Timing Iterations

Sets the number of global iterations that use a simple timing model without transition time effects. Fast Timing iterations are always the first iterations performed. All subsequent iterations use the full timing model. The Fast Timing model is fast, but somewhat less accurate than the other models. By default, Fast Timing Iterations is set to 0 when you set the optimization values by choosing None, Min, Normal, or Full from the top of the Optimization menu.

Global Equalization

Sets the delay goal equal to the largest delay within a timing group when the original delay goal cannot be met. When Global Equalization is disabled, local equalization is employed and the delay goal is set equal to the largest delay within a group.

Steps

Selects the optimization steps to be performed. The text label is the integer code representing all the currently selected optimization steps. You can use this integer value in the command-line mode (option -opt) to turn on the same optimization steps.

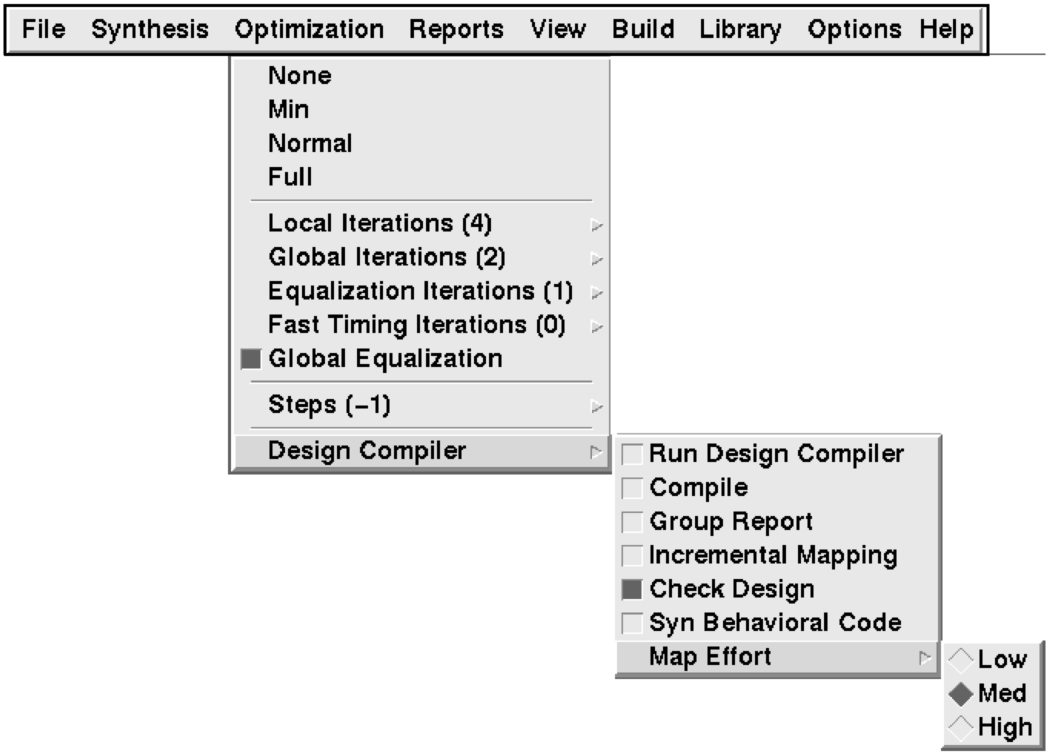
The following table lists the optimization options that can be toggled from the Steps submenu on the Optimization menu. It is usually a good idea to enable all options, even though this can cause optimization to take a long time for large designs.

You can specify the optimization steps to be executed but not the order. In general, strict improvement optimizations are performed first, followed by rule optimizations, then reversible and finally irreversible optimizations.

Table 3-3 Optimization Steps

Optimization Step	Explanation
Synthesis	Allow logic reduction during synthesis
Gate Eater	Remove all instances which have no connected outputs
Rules	Correct nets whose load exceeds the maximum allowed
Reorder	Improve the circuit by reordering equivalent input pins (potentially time consuming but occasionally results in large performance improvements).
LogicMin 1	More sophisticated logic minimization than the one used during synthesis.
Logic Min 2	Find instances that can be removed.
LogicMin 3	Merge parallel inverters, buffers, and flip-flops. Usually fast, but not reversible.
LogicMin 4	Push “bubbles” from instances into inverters or flip-flops
LogicMin 5	Break or reduce an instance into a number of inverters and/or buffers.
Timing	Increase slack in the circuit when the delay goal is not met. May increase area and/or power.
Area/Power	Use a set of smaller or lower-power equivalent cells as candidates for swaps
Min Slack	An enhancement to the Wallace tree building algorithm which provides some performance improvement.
Comp/Drive	Try to balance the usage of compute and drive sections in the design to match that available in the array.
FF	Optimize FFs during optimization rather than synthesis to prevent bad swaps from being made early.

Figure 3-8 The Design Compiler Submenu



Run Design Compiler

Enables/disables whether Design Compiler runs during report generation. Constraint files are generated only if Design Compiler is running.

Compile

Toggles whether the compile is performed within Design Compiler. Set this option if you want to optimize the circuit with Design Compiler.

Group Report

Normally only the critical path for the design is generated by Design Compiler. Enabling this menu choice causes a critical path to be reported for each group in the design. The critical paths are analyzed before compiling the circuit, because Design Compiler changes the instance names during optimization. This option can be useful when you use Design Compiler to analyze an MC design after placement and routing.

Incremental Mapping

Enables/disables incremental mapping by Design Compiler. Generally, when incremental mapping is enabled, runtime is reduced and the circuit structure is changed less severely.

Check Design

Enables/disables the **check_design** command within Design Compiler.

Syn Behavioral Code

Selects which type of MC output code to use as input for Design Compiler. Enabling this option selects MC behavioral-level code; disabling the option selects MC gate-level code.

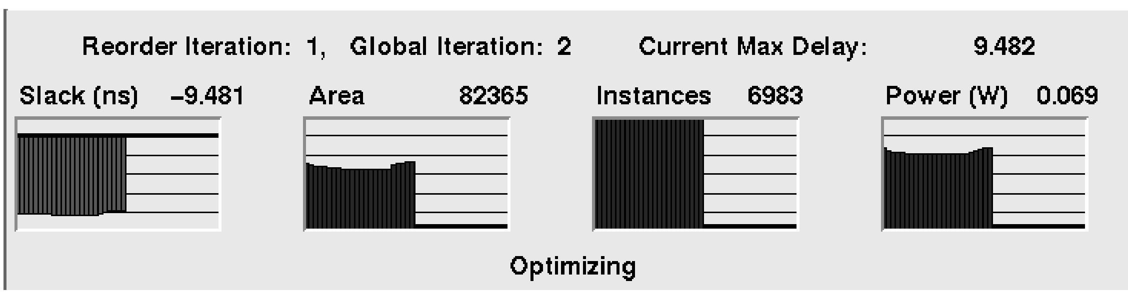
Map Effort

Displays a cascade menu in which you to choose Low, Medium, or High for the mapping effort. In general, the higher the mapping effort, the greater the runtime and quality of results.

Optimization Status Display

During optimization, the progress is displayed as a series of bar graphs. This type of display lets you quickly gauge the effectiveness of the optimization processes and to determine the distance to the design goals. Negative slack values (delay goal not met) are displayed in red, while positive values are displayed in blue. There are bar graphs for Slack, Sections, Instances, and Power. The current optimization step and delay are displayed above the bar graphs.

Figure 3-9 The Optimization Status Display



Click the **Abort** button to stop the optimization process. The current optimization step is always completed before aborting. This ensures that the network is complete so that any reports generated after the abort are valid. Of course, the network may be suboptimal if optimization is aborted.

During optimization, details are sent to the **log window**, indicating the timing, area, power, and critical group for each optimization step.

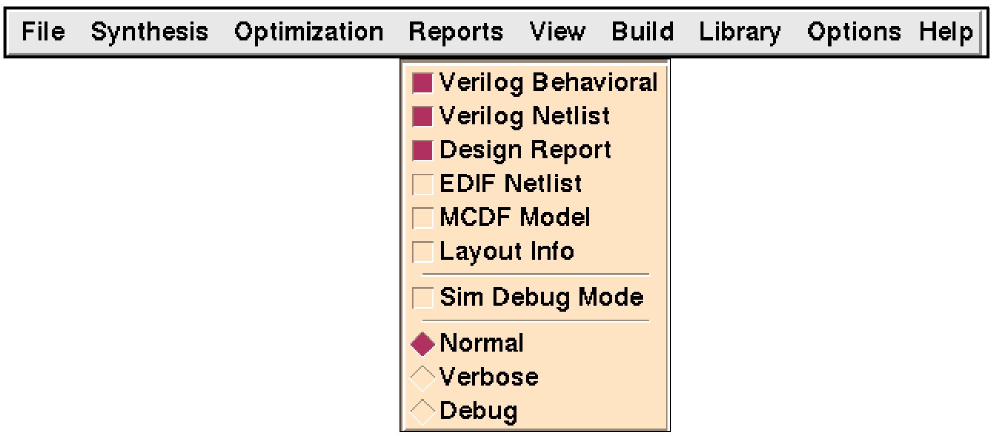
Report Generation (The Reports Menu)

MC can generate a number of different reports after any successful synthesis or optimization operation. Use the Reports menu to choose which report files to generate. Once you have generated a report file, you can look at it by selecting it from the View menu.

See Chapter 10 for a full description of MC's various output files and suggestions for using these files to interpret your results and plan further design refinements.

Note: Scan test mode and the use of clock I/O buffers result in network changes during report generation, so the circuit must be resynthesized after generating reports when either of these features is enabled.

Figure 3-10 The Reports Menu



The following items are available for controlling which files are generated.

Verilog Behavioral
Enables/disables generation of the Verilog behavioral model.

Verilog Netlist
Enables/disables generation of the Verilog gat e-level model.

Design Report
Enables/disables generation of the detailed Design Report.

EDIF Netlist
Enables/disables generation of the EDIF gate level model.

Layout Info

Enables/disables generation of MC's layout information file. See Chapter 8 for an extensive description of this file and its uses.

Sim Debug Mode

Enables/disables the use of debugging names in the netlist models. When this is enabled, long instance names that include the operand, bit position, and cell name are used. When this option is disabled, all instance names start with I and end with a unique number. This mode should be disabled before going to place and route, because the long names may cause problems in verification. The Use Group Names option is orthogonal to this option and controls the insertion of the group name at the beginning of the instance name.

Normal/Verbose

This selects either Normal or Verbose output. Verbose mode provides more information about errors, warnings, and status information in the MC log file. Normal mode is recommended except when debugging. Messages generated with the **info** function appear only in Verbose mode. Contextual information from the HDL code is available only in Verbose mode.

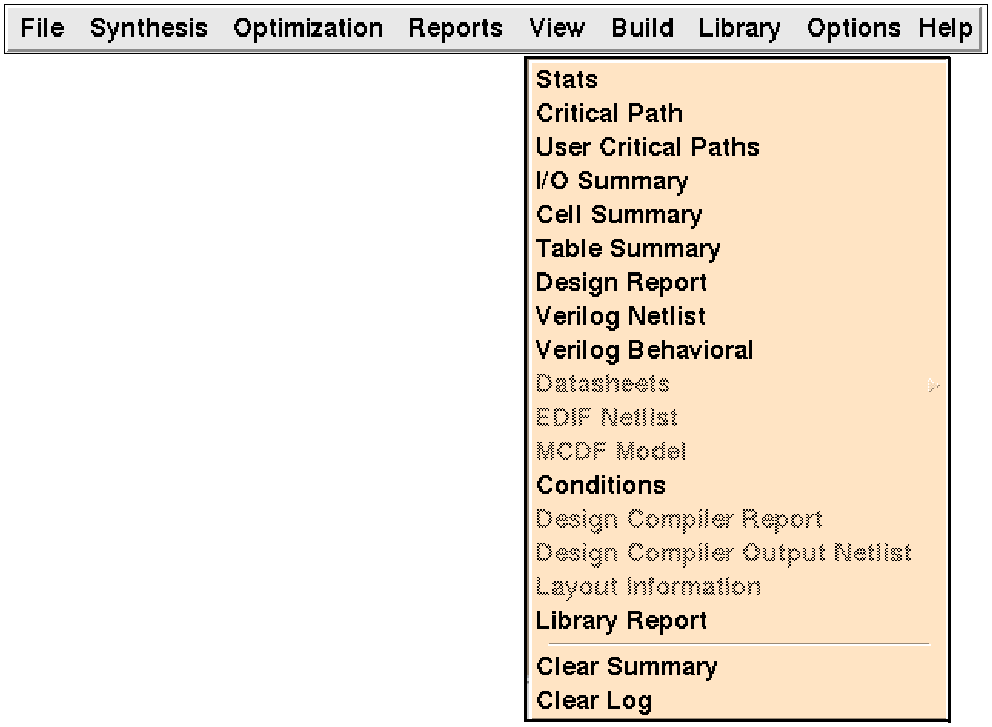
Viewing MC Output (The View Menu)

Use the View menu to view generated reports. Reports that are not available are dimmed in the menu. When any of the following items are selected (except for Conditions), a text window opens with the requested information. Text windows are updated automatically whenever an MC operation is performed that changes the contents while it is open.

Only one viewer of each type can be open at a time. Selecting the item for a viewer that is already open brings the existing viewer to the top of the display stack.

See Chapter 10 for a full description of MC's various output files and suggestions for using these files to interpret your results and plan further design refinements.

Figure 3-11 The View Menu



The View menu contains the following items. Except as noted, all files are generated when you click the Gen Reports button. Each file is displayed in a text window that can be resized and scrolled. In addition, each window has a Find Top button that brings the main window back to the top of the display.

Stats

Displays the group and design statistics. This is available whenever a valid network exists.

Critical Path

Displays the most critical path in the design. This is available whenever a valid network exists.

User Critical Paths

Displays any user-defined critical paths in the design.

I/O Summary

Displays a summary of loading and timing for each bit of every input and output operand.

Cell Summary

Displays a summary of cells used in the design. Cell usage is reported by type (RAM, Combinatorial, I/O, and flip-flop). The listings are sorted both by name and by percentage of sections used by each cell type.

Table Summary

Displays the running summary, showing brief results for previous runs of MC. The number of sections, delay, latency, and the parameters for each run of MC are shown with the design last generated at the top.

Design Report

Displays the detailed Design Report. This report contains group and design summaries, critical path information, the I/O summary, the cell summary, and an operand summary.

Verilog Netlist

Displays the gate-level Verilog model.

Verilog Behavioral

Displays the behavioral-level Verilog model. The behavioral model is generated at the end of synthesis.

Datasheets

Displays the datasheets for any cells, such as RAMs, created during synthesis. A cascading menu displays the list of all newly-created cells.

EDIF Netlist

Displays the gate-level EDIF netlist.

Conditions

Displays the vendor, technology, wire load model, and current operating conditions in the status window.

Design Compiler Report

Displays the report produced by Design Compiler.

Design Compiler Output Netlist

Displays the netlist produced by Design Compiler.

Layout Information

Displays the placement information generated by MC as described in “Information Provided” in Chapter 8.

Library Report

Displays information about the currently loaded technology library and maps the generic MC cells to specific vendor-provided cells.

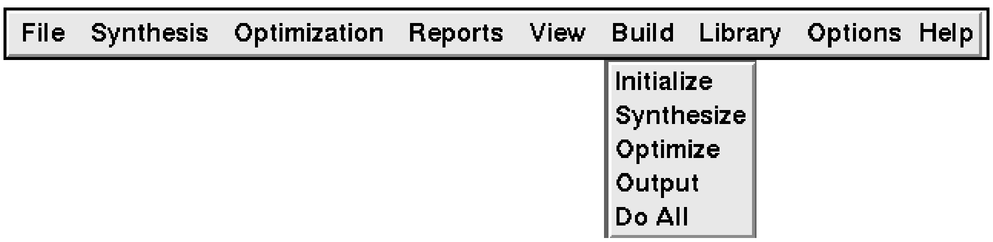
Clear Summary
Clears the table summary.

Clear Log
Clears the log window.

The Build Menu

The Build menu provides the operations that you need to build your design. These items—except for Initialize—are also available on the action buttons found just below the menu bar.

Figure 3-12 The Build Menu



Initialize
Reads in the technology library only if it has not already been read. Module Compiler initializes automatically when it is invoked, so you do not normally need this option. When MC is already initialized, selecting this option has no effect.

Synthesize
Causes the circuit to be synthesized. Selecting this menu item is the same as clicking the Synthesize action button.

Optimize
Causes the circuit to be optimized. Selecting this menu item is the same as clicking the Optimize action button.

Output
Generates the reports that you have selected in the Reports menu. To see a report, select it from the View menu. Selecting this menu item is the same as clicking the Gen Reports action button.

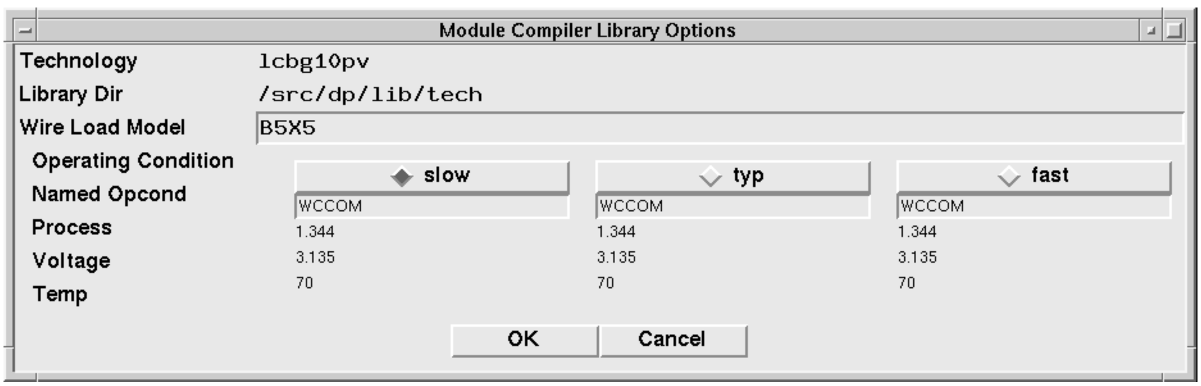
Do All

Performs synthesis, optimization, and report generation in order and is a convenient way to generate the reports after making an input file or parameter change.

Library Options

The Library menu displays the Module Compiler Library Options dialog box, which displays information about the technology library that is currently loaded, and allows you set the wire load model and operating condition.

Figure 3-13 The Library Options Dialog Box



Technology

Displays the name of the currently loaded technology library.

Library Dir

Displays the name of the directory that contains the technology library files.

Wire Load Model

Shows the name of the current wire load model. To change the model, type in a new name. If you type in a model name that is not in the current vendor library, MC pops up a list of available models. You can find detailed information about what's available in the current vendor's technology library by choosing Library Report from the View Menu.

Operating Condition

Selects the conditions under which the chip is likely to be used. These are radio buttons, so only one can be selected.

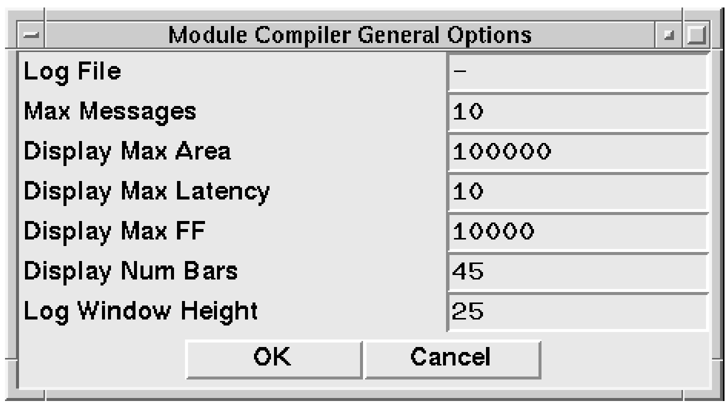
Named Opcond
Specifies which model in the technology library is associated with each of the Operating Condition radio buttons.

Process, Voltage, and Temp
These three items display the values assigned to these items by the associated Named Opcond.

General Options (The Options Menu)

Some general setup and GUI options are set in the general options window, shown below. Choose Options in the main menu bar to display this window.

Figure 3-14 The General Options Dialog Box



The following items are available in the general options window:

Log File
Enter the name of the file to record log messages. All messages sent to the log window are also copied to this file, unless "-" (dash) is provided as the file name.

Max Messages
Enter the limit on the number of similar messages to print before giving up. Use this option to keep large numbers of similar messages from filling the log window, but be aware that important messages may also be masked.

Display Max Area
Enter the maximum number of area units for the synthesis status display.

Display Max Latency

Enter the maximum latency for the synthesis status display.

Display Max FF

Enter the maximum number of flip-flops for the synthesis status display.

Display Num Bars

Enter the maximum number of bars to be displayed in each optimization status bar graph.

Log Window Height

Enter the height of the log window in characters. To remove the window, enter zero. This value takes effect the next time data is sent to the log window. Note that manually resizing the window overrides this value. For large or complex designs, large log windows in conjunction with Verbose mode may significantly slow down synthesis.

MC language Guide

4

This chapter describes the general makeup and components of the MC language. Chapter 5 describes the practical application of the language.

Chapter 4 discusses the following topics:

- The layout of MC input and input flow control
- Modules
- Variables, operators, expressions, and directives
- Functions
- The macro preprocessor
- Error handling

The MC language

The MC language is the primary means for providing a high level description of your design to Module Compiler. The MC language is a Verilog-like, hardware description language. It has the look-and-feel of Verilog, but it differs in details. It also introduces some constructs and operators not found in Verilog. Nevertheless, Verilog users will quickly become comfortable with the MC language.

The MC language borrows heavily from the C programming language as well. Prior experience with C will prove helpful, but is not required.

General Layout of the Input

In its most general form, input to MC consists of one or more files containing MC language code. These files contain a high level description of the design to be synthesized. Logically, these files appear as one input stream. If you were to concatenate the files together you would get a monolithic description. The file might have one or more sections as shown below.

```
#define MAX 128                                →1
#define MIN MAX - 64
..
/* this is a module definition                 →2
*/
module test (arg1, arg2, ...);                 →3
     ...
endmodule  // end of module                    →2

/* function f1 is defined here                 →2
*/
function f1(arg1, arg2, ...);                  →4
     ...
endfunction

function f2(arg1, arg2, ...);                  →4
     ...
endfunction
```

1. **Macro definitions** These definitions implement some preprocessing constructs. Though they are grouped together in one place in the example, they can appear anywhere in the input.

2. **Comments** Everything enclosed by **/* */** and everything to the right of **//** in a line is considered a comment. Comments can appear anywhere in the input.

3. **Module definition** This is *the* design description. It is a description of the design to be synthesized. Consequently, an input that does not contain a module is an empty input: there is nothing to synthesize.

   A module is the MC language analog of *main*() in a C-program. A module can appear anywhere in the input, but it must not contain or otherwise overlap a function.

4. **Function definition(s)** These are pieces of encapsulated code that can be called from the module or from inside other functions. Functions are a means of grouping a set of operations into an *abstract* object that can later be referred to by its name. Since they are abstract, functions do not appear as groups or other hierarchical entities in the output.

   A function is the MC language analog of a procedure in a software language. A function can appear anywhere in the input, but it must not overlap a module or any other function.

The following sections describe these constructs in more detail.

Modules

A module definition is the description of the design to be synthesized. The description begins with **module** and ends with **endmodule**. The module statement describes the interface to the cell, where as the description specifies the contents of the cell. As mentioned above, **module** is the MC language equivalent of the C-language *main*(). A simple adder is shown below.

Example 4-1 Sum of Inputs

```
module test (Z, X, Y);            module interface
    output [7:0] Z;               declare the output
    input [7:0] X, Y;             declare the inputs
    Z = X + Y;                    sum!
endmodule
```

This example generates an adder cell that takes two 8-bit inputs (*X* and *Y*) and provides one 8-bit output (*Z*). The list of arguments (*Z*, *X*, *Y*) is the interface specification for the module. The arguments can appear in any order, though it is customary to place the outputs first. There is no limit on the number of arguments.

As in many other structured programming languages, the MC language requires that a variable be declared before it is accessed. Similarly, the arguments for a module need to be declared before they are referenced. The input statement declares a signal input for a module, where as the output statement declares the output of a module. The inout statement declares a bi-direction port of the module (inouts can only be connected to a pad of a bi-directional I/O driver). These signals must have a width, and can optionally be assigned a numeric format.

Table 4-1 Examples of Module Argument Declarations

`output unsigned [7:0] A, B;`	unsigned 8-bit wide outputs A, B
`output [7:0] a, b;`	8-bit wide outputs a and b, unsigned by default
`input signed [15:0] X, Y;`	signed 16-bit inputs X and Y
`input [0:0] xxx, yyy;`	unsigned 1-bit inputs xxx and yyy

The wire statement also declares a signal. This signal is not an input or an output: it is internal to the module.

The module definition itself consists of one or more statements. The definition ends with the **endmodule** keyword. Note that as in Verilog, there is no semicolon after the **endmodule** whereas most other statements end in a semicolon. The statements which make up a module are one of the following types.

- Declare a variable
- Compute something and assign it to a variable
- Set a directive
- Print a message

These groups of statements are described in the following sections.

It is possible to write arbitrarily complex input descriptions using a module alone, without any hierarchical abstraction of groups of operations. However, code written in this way is not amenable to reuse. A more effective approach is to build a set of functions which can then be called to build this or another module which happens to require the same functions. Functions are described in a separate section below.

Variables, Operators, and Expressions

The section above included examples of the addition based operators. Most operations in the MC language involve some variables and operators. All variables must obey the following rules:

- Variables must be declared before use
- Variable names must begin with a letter and can contain only alphanumeric characters and "_"
- Variable names must not be the same as other keywords in the language or the same as other symbol names (names of functions, cells, etc.)

Variables are combined into expressions using operators. The MC language supports variables of several types.

Signal Variables

The section above introduced module input, output, and wire declaration statements. These statements are used to declare signal variables. In addition to the general rules above, signal variables must obey the following rules.

1. A continuous time path from a signal variable to itself must pass through a sequential element or a feedback input of a function. With this restriction, the network can be sorted for synthesis. If a path from a variable to itself does not pass through a sequential element or a feedback input, the circuit cannot be synthesized and an error is generated.

Incorrect: Creates a Loop from Z to Z	OK, Because X Is an Input

`wire [7:0] X, Z;`
`Z = Z + 10;`
	`input [7:0] X;`
	`wire [7:0] Z;`
	`Z = X + 10;`

2. Signal variables must be assigned-to only once. Thus, a variable can appear on the left-hand side of an equality only once, and it is not possible to assign a bit range of a variable.

3. Signal variable must have a width before they are used in an expression. Thus, while the following is syntactically correct, it is semantically meaningless. Signal declarations without a width are a very useful construct which is further explained in the section on functions.

```
wire [7:0] Z;
wire X;
Z = X + 10;                    ERROR: X must have a width
```

Note that *X* in the example above does not have a 1-bit width; rather it has no defined width.

The + operator is only one of the signal operators. There are several other operators which can be used to combine signal variables into expressions. A "datapath" is actually no more than a series of these signal expressions. Most of these operators should be familiar to users of common programming or behavioral modeling languages. There are some new operators, such as >>>.

Where applicable, these operators follow precedence rules of the C language (which are the same as the precedence rules in Verilog). The operators are listed below in order of decreasing precedence. Operators on the same line have the same precedence. When operators of the same precedence are encountered in the MC language file, they are processed from left to right.

Table 4-2 Signal Operators

Signal Operator	Name
(width)	casting
[]	bit range
()	expression grouping
- ~	(unary) arithmetic negate, bitwise invert
*	multiply
+ -	add, binary minus
<<< >>>	left, right rotate
<< >>	left, right shift
< > <= >=	magnitude comparison
== !=	equality, inequality compare
&	bitwise AND
^	bitwise XOR
|	bitwise OR
? :	multiplex

The precedence rules govern the order in which the operators are applied to the variables. Parentheses can be used to override this order or to make the code more readable. In the example below, while the first two expressions are identical, the second expression is easier to understand because the order of evaluation is immediately clear. Also, note that the first two expressions are quite different from the third which computes the OR of B and C first and then multiplies it with A.

```
Z0 = A * B | C;      compute the product, then OR
Z1 = (A * B) | C;    same as above
Z2 = A * (B | C);    compute the OR and then the product
```

Normally, a variable name such as “A” denotes the entire signal. It is sometimes necessary to selectively access a certain range of bits in a given signal. This can be done by using the [ ] operator. Bit ranges are bounded by the width of the signal variable, meaning that bit ranges must be in the interval from 0 to width–1. Bit ranges can be used any place a signal can be used except that bit ranges must not be used on the left hand side of an expression; that is, you must not selectively assign a value to a range of bits.

```
Z1[3:0] = A * (B | C);
```

Not OK! Cannot assign to a bit range

```
Z2 = A[3:0] * (B | C);
```

OK, if A is 4 or more bits wide

```
Z = Z1 = Z2 = ~((A[3:0] ^ B) + (C == D));
```

A complex expression that computes XOR of A and B and adds 1 if C equals D, else adds 0. Finally, it complements the result and assigns it to Z, Z1, and Z2

While the degenerate case is a single variable, an expression can arbitrarily contain many operators and variables. An expression does not need to contain an assignment, but in most cases, an expression without an assignment is not very useful.

You can now rewrite the example presented in the Module section so it accepts one more input and computes a product of a sum instead of just a sum. The output is now declared to be larger to hold the product.

Example 4-2 Sum of Products

```
module test (Z, X, Y1, Y2);       module interface
    output [15:0] Z;              declare the output
    input [7:0] X, Y1, Y2;        declare the inputs
    Z = X *(Y1 + Y2);             compute!
endmodule
```

Temporary Signal Variables

Operator-based notation allows for compact description of signal operations. These operations however are not always synthesized in a single step. Often when an expression contains operators of different types, it is broken into multiple expressions and then synthesized. The intermediate steps lead to the creation of temporary variables that are otherwise invisible and inaccessible, but appear in the output. The creation of temporary variables can be controlled with the **autotemp** attribute.

Example 4-3 Creating Temporary Signal Variables

```
Z = X + Y + (U | V);
```

Assuming that Z, X, Y, U, and V are properly declared variables, this expression is legal but it cannot be synthesized in one step. It is broken into two expressions as shown below.

```
temporary_variable = U | V;
Z1 = X + Y + temporary_variable;
```

These two expressions yield the same result as the above

If the MC language parser creates a temporary variable, it typically names it after the signal on the left-hand side. In the example shown above, the temporary variable is named *Z_<value>_*, where *value* is an integer value greater than 1 (for example, *Z_5_ or Z_7_*). The complete details for the naming scheme are described in a separate section below.

A construct that can be synthesized in one step generally results in higher performance. This is discussed in more detail in Chapter 5. Where this is not possible, you have the option of describing the design in discrete synthesis steps. The second code fragment in the example above shows this option. Alternatively, you can use a more natural, algebraic notation and let MC create the intermediate steps. This is the case in the first expression in the example.

If the parser breaks up expressions and creates temporary variables, it needs to compute the width (and format) of the temporary variable. It does so by using the width and format of the variables on the right-hand side. In most cases, this does not present a problem because the attributes of the temporary variable can be determined unambiguously. There are some cases where there is no single correct answer, and inefficiencies (or unexpected results) can occur. In these cases, you can specify the width and format of the temporary variable by employing the width operator. The width operator allows you to prefix the familiar **signed [x : 0]** or **unsigned [x : 0]** to an expression to indicate the width and format of the temporary variable used to hold the result. This is further illustrated below.

The rules for generating temporary variables are quite simple: *, –, + can be grouped together while other operators cannot be grouped. In some cases ~ can also be grouped. The detailed rules are as follows.

1. If a width operator is present, then a temporary is always created. The width operator (unsigned [7:0]) in the following example causes the first expression to be broken up in to two expressions. The result is the same as the two subsequent statements.

```
Z0 = A * (unsigned [7:0]) (B + C);
```

force a temporary variable

```
wire unsigned [7:0] Z1 = B + C;
Z2 = A * Z1;
```

identical to the above

2. Left shift and right shift by integer values always generates a temporary variable with enough bits to prevent data loss. In the following example, the first statement will be broken into two. Note that the width of *Z1* is wide enough to prevent data loss.

```
Z0 = A * (B << 5);                    integer shift creates a temporary

input unsigned [7:0] B;
wire unsigned [12:0] Z1 = B << 5;     identical to the above
Z2 = A * Z1;
```

3. Sums or differences of variables, or sums or differences of products of variables do not generate a temporary variable. Magnitude operators (>, >=, <, <=) can also be included without generating a temporary. The following examples illustrate this rule:

```
Z0 = (A * B) + (C * D ) - (E * F);
```

temporary variable not required

```
Z2 = A * (B + C);
```

temporary variable needed for B + C

```
Z3 = (A * B) | C;
```

temporary variable needed for A * B

4. If the expression consists entirely of one logical operator (AND, OR, XOR), then no temporary variable is required. Each operand can be inverted.

```
Z0= A | B | C;              requires no temporary variable
Z1 = A & ~B & C;            ditto

Z00 = A | B & C;            requires temporary variable for B & C
Z01 = A + B | C;            requires temporary variable for A + B
Z02 = ~(A | B | C);         requires temporary variable for A | B | C
```

5. If the expression consists of a single shift or rotate then no temporary variable is required.

```
Z0= A << B;              requires no temporary variable
Z1 = A >> B;             requires no temporary variable

Z3 = A >> (B >> C);      temporary variable required for (B >> C)
Z4 = A << (B - C);       temporary variable required for (B - C)
```

6. If the expression consists of a MUX only, then no temporary variable is required. Also, the output can be inverted.

```
Z0 = ~(A ? B : C : D);   requires no temporary variable
Z1 = A ? B : (C | D);    temporary variable required for C | D
```

7. Any other mixing of signal operators requires the use of temporary variables.

8. If a parameter passed to a function has a width or format that does not match that of the corresponding input or output declaration inside the function, a temporary is generated to perform a conversion between the mismatching widths and formats.

```
function foo (Z,A,B);
input A;
input [3:0] B;
output [7:0] Z=A+B;
endfunction

module test (Z,A,B);
input [4:0] A,B;
                              no temp for A
                              need a temp for B to convert 5 to 4 bits
output Z=foo(A,B);            need a temp for Z, it is an output
endmodule
```

9. All module outputs result in temporary variables.

Once it has been determined that a temporary variable is needed, its width and format must be determined. In most cases this is quite obvious and error free. The detailed rules are:

- If the temporary variable is due to a width operator, then the supplied width and format are used.
- If the temporary variable is due to a left or right integer shift, then the width is the width of the input plus the left-shift or minus the right-shift. The format is always the same as the signal input. This rule does not apply if a signal is shifted by another signal.
- If the temporary variable is due to a comparison operator (==, !=, >=, etc.), then the width is always 1-bit and the format is always unsigned.

- For all other expressions, the format of the temporary variable is signed if any of the data inputs are signed; otherwise, the format is unsigned.(‘data inputs’ excludes variables such as the select input of a MUX, shift input of a shifter, etc.).

  The width of the temporary variable equals the width of the largest input. If the largest input is unsigned and there is another signed input, then the width is incremented by 1.

The example for computing sum of products relied on the MC language parser to create the intermediate steps. You could rewrite it as shown in Example 4 by using explicit steps. For the results from the two approaches to be identical, the width and format of the manually created intermediate variable has to match the automatically created one (8-bits wide, unsigned). That is, if you wanted to allow for overflow from the sum by using a bigger intermediate operand, then you should use the explicit approach and declare the intermediate operand to be 9 bits wide.

Example 4-4 Sum of Products

```
module test (Z, X, Y1, Y2);

    output [15:0] Z;
    input [7:0] X, Y1, Y2;
    wire [7:0] temp;            declare a local variable

    temp = Y1 + Y2;             same as Z = X * (Y1 + Y2)
    Z = X *temp;

endmodule
```

Integer Variables

Integer variables are the MC language equivalent of the C language **int** or the Verilog **integer**. They are normally 32-bit quantities representing whole numbers. The bit-width (and the resulting range of values) can be changed as described in the section about constants.

Integers follow the generic variable rules such as name convention, declaration requirements. Example 4-5 lists some examples of integer declaration and use:

Example 4-5 Examples of Integer Declaration and Use

`integer x;`	declare an integer variable named x
`integer y = 20;`	declare an integer and initialize it to 20
`integer a, b, c;`	declare three variables
`a = x + y / 2;`	assign a value to **a**
`integer a = x + y /2;`	same as above

Integers support most of C language operators and can be used to construct expressions in the usual way. These expressions can be used wherever an integer is expected. The obvious advantage over using a constant number is that the value of an integer variable can change while the MC language input is being synthesized. Thus allowing a high degree of parameterization. The integer operators supported by MC are listed in Table 4-3 in order of decreasing precedence.

Table 4-3 Integer Operators

Integer Operator	Name
( )	expression grouping
- !	(unary) arithmetic negate, relational not
~	bitwise invert
* / %	multiply, divide, mod
+ -	add, binary minus
>> <<	right shift, right shift
< > <= >=	magnitude comparison
== !=	equality, inequality compare
&	bitwise and
^	bitwise xor
|	bitwise or
&&	logical and
||	logical or

Integers can be used in the interface definition of a module. These integers can be used to further parameterize the input. For instance, you can rewrite the sum-of-products example to allow inputs and outputs of varying sizes. The widths are passed into the module using a construct such as “in=8.” See “Choosing an Input File, Parameters, and Optimization Criterion” in Chapter 3 for further discussion of passing in parameters. Note that in all cases, the output is a cell called test that has three inputs and one output; the integers in the module interface have been resolved away. Integers can be given a default value as shown in the example below. If no value is provided for **out**, it has the value 6. A value must be provided for **in**.

Example 4-6 Module Parameters of Varying Widths

```
module test (Z, X, Y1, Y2, in, out);

     integer in, out=6;
     output [out - 1:0] Z;              declare the output
     input [in - 1:0] X, Y1, Y2;        declare the inputs

     Z = X *(Y1 + Y2);                  compute !

endmodule
```

Another interesting use of integers is in signal expressions. When used in this way, these integers denote signals that have fixed values. Since they are represented using integer *variables*, these values are fixed at the time of synthesis, but are variable when the MC input is being processed. The following sequence illustrates this use.

```
integer step = 16;
```

If **Z** and **A** are signals, then adder inputs are **A** and the current value of **step** (which is 16)

```
Z = A + step;
```

Adder inputs are now **A** and the new value of **step** (which is 32)

```
step = step << 1;
Z1 = A + step;
```

The precedence of operators is unaffected by the type of operand (signal, integer, or mixed) and the parser is able to separate out signals and integers. When possible, enclose integer expressions in parenthesis when they are used inside signal expressions. Note that the synthesis result does not change, but readability and MC runtime are slightly improved.

```
Z1 = X + (XX + YY + 5) + Y;
```

XX, YY are integers; X, Y are signals

String Variables

String variables are the character-string equivalent of the integer variables. Unlike integers, strings allow only a limited number of operators.

Table 4-4 String Operators

String Operator	Name	Expression Prototype	Result
()	expression grouping	(a Op b)	
[]	substring	a[n:m], a[n]	string
+	concatenate	a + b	string
==	equality compare	a == b	integer
!=	inequality compare	a != b	integer

Like integers, strings can also be passed as arguments to a module. This is useful in passing some names into the module. Strings can also be given default values in the same manners as integers.

Strings are declared using the **string** keyword. String constants are differentiated from others by enclosing them in double quotes. Some representative uses of **string** are shown in the example below. A **string** function is also provided. This function is used for concatenating different names, constants, and values into one string. The *strlen* function is available to return the length of a string. The following conventions are also available:

To Enter This	Type This
newline	\n
embedded tab	\t
embedded double quotes	\"

Example 4-7 Using Strings

```
string x;                                   declare a string called x
x = "I am a string.";                       initialize it
x = "hi! " + x;

string y = "hi! I am a string.";            another string

integer eq = x == y;                        eq equals one because x and y are equal

integer eq1=x[2:0]=="hi!";                  eq1 equals one

integer len=strlen(x[4]);                   len equals one

          Create a string with the text “name of X is X and width is 16”.
wire [15:0] X;
string x = string("name of X is ", X, "and width is ", width(X), "\n");
```

As might be expected, it is an error to add or compare a string and an integer or a signal.

Constants

The sections above contain numerous examples of constants. Just as there are different types of variables, there are different types of constants. A number like 5 or 25 is an integer constant, whereas a character string like "abc" or "25" is a string constant. Constants can be used anywhere a quantity of its type is required.

MC supports large integers with decimal, binary, hex, and octal formats and widths up to 1024 bits. By default, integer values are represented with 32 bits. When a numeric constant appears in a signal statement, its width is the minimum possible; for example, the width is 4 bits if the constant is 15. This behavior can be modified by attaching a format as well as a width specification to a constant. Operations on large integers (> 32 bits) are more restrictive than normal integers, but most common operators like +, -, *, etc. are supported.

Every type of constant can begin with a minus sign to indicate that the value is negative. Hexadecimal constants are identified by **'h** followed by the characters in the set {0123456789abcdef}. The alphabetic characters can be replaced by the uppercase equivalents. Octal constants are identified by **'o** followed by characters in the set {01234567}. Binary constants are identified by **'b** followed by characters in the set {01}. Decimal constants can be identified by **'d**.

Example 4-8 Examples of Constants

```
Z = A + 15;                  15 is a 4-bit input to the adder
Z1 = A + 32 'h f             15 is a 32-bit input here

integer x1 = 101;            assign decimal 101 to integer variable
x1
integer x2 = 'h 101;         assign hex 101 to x2; x2 = 25
integer x3 = 'o 101;         assign octal 101 to x3; x3 = 65
integer x4 = 'b 101;         assign binary 101 to x4; x4 = 5

integer x5 = 64'h 101;       assign hex 101 to x5; x5 is 64-bit wide
integer x6 = 'h a12b5678c;   x6 is a large integer
integer x7 = x6 * 15;        x7 is also a large integer

integer y = 'hx;             make y a "don't-care" value
```

Global Variables

As described above, MC requires that all variables be declared before they are referenced: there are no implicitly created variables that can be accessed in an MC language input. A notable exception to this rule are global variables, which are always available and visible.

MC currently has one predefined global signal variable, called *CLK*, which is used to represent the default clock signal. *CLK* is like a module input: it is considered preassigned and can therefore be used to compute other signals. Other clock signals can be created by setting the **clock** attribute**.**

You can create other global signals by placing the **global** keyword after the **wire** keyword in a signal declaration. The global wire defined in this way can be accessed in any code executed after the declaration. Module inputs and outputs cannot be declared as global.

Global integer and string variables are created in a similar manner by placing **global** after the **integer** or **string** keywords in the variable declaration.

It is recommend that global variables be used only when absolutely necessary. The overuse of global variable can make your code more difficult to reuse and maintain.

A locally declared variable with the same name as a global variable takes precedence over the global variable within the function that the local variable was declared.

In the example, below a global reset signal, RESET, is defined and used in a function.

Example 4-9 Examples of Global Variables

```
function cont (Z,A);
     input A;
     output Z;
     Z=count (A,RESET,start,0);
endfunction

module test (Z,A,R);
     input [7:0] A;
     input [0:0] R;
     integer global start=3;
     wire global [0:0] RESET=R;
     output [7:0] Z=cont(A);
endmodule
```

Directives and Attributes

Directives provide a mechanism for providing operating hints to MC by setting the value of attributes. Generally, the attributes influence the way a design description is compiled and synthesized rather than changing the functionality of the design. However, some attributes—**pipeline**, for example—can affect the latency of the design.

There are three types of directives: global, local and default. What distinguishes these types is the scope of influence. Global directives affect all statements following the directive, both higher and lower in the function hierarchy. Therefore, a global directive issued in a function can affect subsequent statements in the caller. By default, directives only affect statements following the directive that are at the same or a lower level in the hierarchy. A default directive issued in a function can affect statements in the same function and in functions called from the function containing the directive. But a default directive in a function cannot affect statements in the function or module that called the function containing the directive. Local directives affect only the next statement. If the next statement is a function call, the entire function call and all functions called from that function are affected. These directives are used to make temporary changes in the directive values.

Attributes are actually typed variables that accept a range of integers or certain strings. These variables are accessed using the **directive** keyword.

Example 4-10 Directive Scope

```
directive (pipeline = "on");              default scope
directive global (pipeline = "on");       global scope
directive local (pipeline = "on");        local scope
```

These directive statements set the **pipeline** attribute to **on**. This attribute accepts only two values: “on” and “off”.

The value of an attribute can be queried any time. For example, the following statement sets the string **x** to “current value of the pipeline attribute is on.”

```
string x = "current value of the pipeline directive is " + directive(pipeline);
```

You can set several attributes in a single statement, but access to the attributes must be one at a time, as shown in this example.

```
directive(pipeline = "on", delay = 1000);

string currentPipe = directive(pipeline);
integer currentDelay = directive(delay);
```

Further details on attributes are provided in Chapter 5 and in the *Module Compiler Reference Manual*. As a final illustration of **directive** use, the following example modifies the sum-of-products example so it outputs a cell with a given name. It allows inputs and outputs to be of varying widths and uses the **modname** attribute to set the name of the output cell to whatever value was passed in.

Example 4-11 Using **directive** to Create a Named Output Cell

```
module test (Z, X, Y1, Y2, in, out, name);
     string name="foo";                                module name is foo, by default
     directive(modname = name);

     integer in, out;
     output [out - 1:0] Z;                             declare the output
     input [in - 1:0] X, Y1, Y2;                       declare the inputs

     Z = X *(Y1 + Y2);                                 compute !

endmodule
```

Messages

The MC language provides several functions for printing messages during the input compilation stage. These functions are useful in catching and reporting errors and as a general debugging aid. The following types of messages are provided.

- Information Message

  The syntax for the info message is very similar to the string function. info concatenates all its inputs and prints them on the standard output. As such, it is a general debugging aid. Some examples of info messages are shown below.

Example 4-12 Using the **info** Keyword

```
integer n = 16;
...
wire [n-1: 0] X;
info ("name of X is ", X, " and width is: ", width(X), "\n");
                                     prints out name of X is X and width is 16

info("n exceeds magic value? ", n > magic, "\n");
                                if n > magic, prints out n exceeds magic value: 1
                                else prints out n exceeds magic value: 0
```

There is usually a leading identifier in the output to indicate that this message was generated as a result of an **info** statement. The name of file and function containing the statement are also printed. Such messages when combined with macros and conditional (if/else) constructs provide a useful tool in debugging complex MC inputs. Macros and flow control (if/else) are described later in this chapter.

- Warning Message

  This message is another variation on the **info** message. Here the occurrence is counted as a warning. The parser as well as the synthesizer try to continue. The keyword for warning messages is **warning**.

- Error Message

  This message is virtually identical to the info message except that its occurrence is counted as an error. If this message is encountered, the MC language parser prints it and tries to continue, but no synthesis takes place. If the parser encounters many of these messages, then the parser quits as well. The keyword for error messages is **error**.

```
error("n exceeds magic value! n = ", n, "magic = ", magic, "\n");
```

prints `n exceeds magic value! n = 10, magic = 5` if n and magic are 10 and 5, respectively

- Fatal Error Message

  This message is a stricter form of error. When the MC language parser encounters this statement, it prints the message and immediately quits all processing.

```
fatal ("integer divide by zero! m - n equal zero in (x / ( m - n))\n");
```

prints `integer divide by zero! m - n equal zero in (x / ( m - n))` and then quits

Note: The MC language parser processes the input in two passes. All user-created messages are processed in the first pass while the final checks for consistently declared and defined signals take place in the second pass. Consequently, it is possible to get the illusion that the MC language parser is generating error messages that are not properly synchronized with the user-created messages.

Macro Preprocessor

The MC language supports use of the C-language preprocessor, `cpp`. This preprocessor can usually be found in `/lib/cpp` or `/usr/lib/cpp` on UNIX systems. While complete description of this preprocessor is beyond the scope of this document, it is appropriate to provide some introductory material.

To begin with, this preprocessor is truly a *pre*processor: it runs before any of the other processing in MC takes place. For instance, this processor is used to strip out comments (**/* */** or **//**) in MC language code. As a result, by the time MC gets around to parsing the input, the comments have been removed. This applies to other preprocessor constructs described in this section as well.

#define

The most popular use of this preprocessor is to define macros. A macro is basically a string of text which can be given a name. Thereafter, whenever this name is encountered, the string is inserted in the place of the name. To make things more interesting, the substitution can be parameterized so that all occurrences of some keyword are replaced by another keyword in the string. The macro is defined using the **#define** construct (the “#” needs to be at the beginning of a new line).

```
#define MAX125
.....
info("n exceeds max value", MAX, "\n");

#define myinfo(x) info("===> width of ", x, " is",
width(x), "\n");
....
wire [n:0] dataIn;
myinfo(dataIn);
```

the preprocessor replaces the line above with **info “===> width of ”, dataIn, “ is”, width(dataIn), “\n”);**

The example above defines **MAX** to be 125. Hereafter, whenever **MAX** is encountered, 125 is inserted. This is a powerful technique, because if the value of **MAX** changes, you need to modify it in only one place and the change ripples through the rest of the code. This example also defines a macro called “myinfo” which accepts one argument. The result of “myinfo” is to expand its call into a call to **info** using the supplied argument.

#include

Another use of the preprocessor is to use it to include one MC language file inside another. This is done via the **#include** construct. When the preprocessor encounters a **#include**, it substitutes the contents of the named file in the place of the line containing **#include**. This technique is useful in distributing your design over several files, but combining it all into one logical stream before presenting it to MC. For instance, if a file called `test.mc` contains the following line,

```
#include "test1.mc"
```

then the contents of `test1.mc` are merged into the contents of `test.mc` at this line. The merging is done on the fly; the original contents of `test.mc` and `test1.mc` are left unchanged.

#ifdef

Both the **#define** and the **#include** constructs can be combined with the **#ifdef** construct to conditionally invoke the preprocessor.

You can use these constructs to build a “debug mode” into the sum-of-products example. If the input is used as shown below, then the MC language parser will print the two sets of info messages. When debugging is no longer needed, you can disable it by defining the “DEBUG” macro to zero.

The following example computes a sum of products, allow inputs and outputs to be of varying widths, and calls the resulting cell whatever name was passed in.

Example 4-13 Using **ifdef**

```
#define DEBUG1

module test (Z, X, Y1, Y2, in, out);

     string name = "testCell";
     directive(modname = name);

#ifdef DEBUG
     info("name of the output cell is: ", name,
"\n");
#endif

     integer in, out;
     output [out - 1:0] Z;
     input [in - 1:0] X, Y1, Y2;

#ifdef DEBUG
     info ("input width is: ", in, "\toutput width
is: ", out, "\n");
#endif

     Z = X *(Y1 + Y2);

endmodule
```

Caution ***Macros and includes can be difficult to debug. These constructs should be used only when the increased efficiency warrants the added complexity.***

Input Flow Control

While it is possible to write many interesting design descriptions using the constructs described above, it is at best cumbersome without the use of some flow control. The reason is that when writing a program or a synthesis description, it is very natural to want to say, “if some condition is true then do the following.” Or, one often wants to say, “repeat the following set of actions so many times.”

One of the strengths of the MC language is that it has general flow control mechanisms which allow the input to be conditionally processed in different ways. These mechanisms consist of conditional blocks (if/else), loops (replicate), and substitution ({}). In each case, the flow of the input stream to MC is altered to fit the mechanism in use. For example, if an *n*-stage loop is used, then the code inside the loop is replicated n times and then processed by MC.

Unlike most programming languages, the flow control constructs in the MC language can appear anywhere, including inside other statements and constructs, and can therefore alter the input or create new tokens. There are some exceptions to this rule which are listed below.

Substitution ({})

This construct allows computed substitutions into the input stream. It evaluates the expression enclosed in {}, converts the results into a string and substitutes it into the input stream. The inserted text gets concatenated with the surrounding text in the input stream if there are no white-space separators. For example the following code fragment creates a new token which is used to name a wire.

```
integer n = 10;
...                  the value of n might be modified in here

wire [7:0] X{n};     creates a wire named X10 if the value of n is 10.
```

The expression in {} can contain any integer or string variables and constants, but it cannot contain any flow control constructs.

Conditional Block (if/else)

This construct allows the input to be conditionally processed. The condition can be any expression which evaluates to a zero or nonzero result. If the condition is true then the text following the **if** is inserted into the input stream and the input is reprocessed. If the condition is not true then either nothing is inserted into the input stream or the text following the else is inserted into the stream. Some examples are shown below.

```
if (n == m) {
    fatal ("integer divide by zero! m-n is zero in (x / (m - n)) \n");
}
```

If n — m is zero, then print an error and stop further processing

```
if (w) {
    wire [w - 1 : 0] X;
}
else {
    wire [7:0] X;
}
```

If a bit width is given, then use that; otherwise use 8 bits

```
wire [if (w) {w - 1} else {7} : 0] X1;
```

Create another wire, identical to **X**

Note that the **if/else** can appear inside other statements. This style is more compact but not as easy to understand as the alternate. More importantly though, this style allows **if/else** constructs to be used in contexts such as module interface definitions where the other style is not allowed. We can use this to modify our recurring example to allow an additional output. The following example uses the values passed in or uses hard-wired values.

Example 4-14 Conditional Blocks

```
module test (Z, X, Y1, Y2, param if (param) {,Z1});

     integer param,in=8,out=16;
     output [out - 1:0] Z;
     input [in - 1:0] X, Y1, Y2;

     Z = X *(Y1 + Y2);
     if (param) {
output [out-1:0] Z1=Z+1;
     }

endmodule
```

Conditional blocks can nest indefinitely. Conditional blocks must be completely contained within a module or function. The condition expression for all conditional blocks must be free of flow-control and substitution constructs. A side-effect of this restriction is that expressions such as

if (width(X{i}) == 8)

which are quite natural, lead to parse errors. This limitation is easily overcome by simply computing the expression with the substitution outside of the conditional.

Loops (replicate, repl)

While the **if/else** construct conditionally inserts a block into the input once, the **replicate** construct conditionally inserts a block into the input stream zero or more times. **replicate** can appear anywhere an **if/else** can appear. It simply replicates the associated text block back-to-back while the loop is executed.

A terse form of replicate is provided by the **repl** construct which is described below. The syntax for **replicate** is very similar to the **for-loop** in the C language. The replication is controlled by three statements and an optional separator. The first statement is executed only at the beginning. Then the second statement (the condition) is evaluated, and if it is true, the text block is replicated and third statement is executed. This is repeated for as long as the condition evaluates true. After that, control is passed to the input following the replicate block. Here is an example. If the optional separator is provided, it is used to separate adjacent segments of the replicated text. Do not include the separator after the final iteration. The following example generates eight wires named X0 through X7.

```
wire [7:0] replicate(integer i = 0; i < 8; i = i + 1) {X{i}, };
```

The following example generates eight wires named Y0 through Y7.

```
replicate(i = 0; i < 8; i = i + 1) {
      wire Y{i};
}
```

These cases are essentially equivalent. The first case expands as shown below with an extra comma at the end of the list. Extra commas at the end of lists are ignored.

```
wire X0, X1, X2, X3, X4, X5, X6, X7, ;
```

Note that in the second case, the **start** statement is different: variable *i* is not declared because it was already declared in the context of the first case. Actually, the **start** and **update** statements can be any statement and the conditional expression can be any expression. This can sometimes lead to trouble as in the following example which creates an infinite loop. In this case, the parser quits after executing the loop some fixed but large number of times.

This example generates eight wires named Y0 through Y7.

```
replicate(i = 0; i < 8; i = i - 1) { wire Y{i}; }
```

oops! infinite loop ^^

The **loop** variable—or any other variable—can be modified or otherwise accessed in a completely unrestricted manner inside the **replicate** block.

Caution

In the case of replicates embedded in a statement, the entire statement is collected before it can be executed. This can lead to some subtle but potentially dangerous side effects.

In the following example, the second code fragment generates the correct result while the first code fragment left-shifts everything by 4. Note that enclosing the integer variable inside {} causes it to be evaluated immediately and the resulting string placed into the input stream.

Wrong! generates `Z = (X0 << i) + (X1 << i) + ...`:

```
Z = replicate(i = 0; i < 4; i = i+1)
     { (X{i} << i) + } 0;
```

Correct: Implementation of shift-and-add. generates `Z = (X0 << 0) + (X1 << 1) + ...`:

```
Z = replicate(i = 0; i < 4; i = i+1)
     (X{i} << {i}) + } 0;
```

A better form for the above is the following:

```
Z = replicate(i = 0; i < 4; i = i+1; "+")
     { (X{i} << {i}) };
```

Since **replicate** simply replicates the text block back-to-back, a problem occurs when the text blocks are separated by a "," or an operator like "+" as in the case above. When the loop terminates, there is a dangling "." or "+" at the end. MC language accepts lists of the type "A, B, C," as well as lists of the type "A, B, C.". So, the dangling "," is harmless. The dangling "+" however has to be properly terminated by padding the replicate with a 0 as shown above. Alternatively, the replicate construct can specify a separate string which is appended to all but the last replication. Note that in the example above, the use of the separator in the **replicate** statement removes the need for the final 0.

Finally, replicates have the same restriction as **if/else** blocks. The start, update and condition expression cannot contain any flow-control constructs, and the replicate must not span or straddle a module. A replicate can appear in all other contexts, including interface definitions. The following shows the now-familiar example, modified so that it generates a cell with variable number of inputs. Note that this example computes (X * Y0) + (X * Y1) +.... It is certainly possible to compute X * (Y0 + Y1 + ...) by rearranging the replicate. Note that integer parameters, *n* and *param*, can be used in the parameter list before they are formally declared in the body of the module.

The following example of multi-input sum of products computes $\sum X*Y_i$ using bit widths that are either hard-wired values or values passed in. The bit widths need to be called either as *n*=<*int value*>, *param*=0' or as *n* = <*int value*>, *param* = 1, *in*=<*int value*>, *out* =<*int value*>

Example 4-15 Conditional Blocks and Replicates

```
module test (Z, X,
     n, replicate(integer i = 0; i < n; i = i + 1) { Y{i}, }
     param, if (param) { in, out});
     /*declare X and Z as before; in addition declare Y0, Y1, etc.
     */
     integer n, param;
     if (param) {
          integer in, out;
     }
     else {
          integer in = 8, out = 16;
     }
     output [out - 1 : 0] Z;
     input [in - 1 : 0] X, replicate(i = 0; i < n; i = i + 1) { Y{i},};
     /*generate X * Y0 + X * Y1 + X * Y2 ...
     */
     Z = replicate(i = 0; i < n; i = i + 1; "+") { X * Y{i} } ;

endmodule
```

A terse form of **replicate** is provided by the **repl** construct. This construct assumes that the **start** statement is always of the form *integer* $x = 0$, the condition is always of the form $x < n$, and the update statement always of the form $x = x + 1$. You must specify the name of the iterator (x) and the upper limit (n). You can optionally specify a separator string.

Note that the arguments are separated by commas, and that the scope of the iterator variable is strictly local to the replicate. The following example is the same as the one above, but uses the short form of **replicate**.

Example 4-16 Using **repl**, the Short Form of **replicate**

```
module test (Z, X, n, repl(i, n) { Y{i}, } param, if (param) { in, out});

     /* declare X and Z as before; in addition declare Y0, Y1, etc.
     */
     integer n, param;
     if (param) {
          integer in, out;
     }
     else {
          integer in = 8, out = 16;
     }
     output [out - 1 : 0] Z;
     input [in - 1 : 0] X, repl(i, n) { Y{i},};

     /* generate X * Y0 + X * Y1 + X * Y2 ...
     */
     Z = repl(i, n, "+") { X * Y{i} } ;

endmodule
```

Functions

As described above, it is possible to write many interesting and nontrivial descriptions without the use of functions. But this becomes increasingly difficult as the complexity of the problem rises. This is where hierarchical partitioning of the input becomes invaluable.

The idea behind hierarchical partitioning is simple: break a big design into many smaller designs and then construct the large design by making references to the smaller pieces. This approach has the obvious advantage of decreasing complexity. In addition, it promotes code reuse: pieces of code written for one design can later be used in another design without any rework. This technique is all too familiar to software programmers today. In fact, it would be difficult to find a nontrivial piece of software which does not employ such hierarchy through the use of procedures.

In the MC language, a function is the equivalent of a software procedure. It is a chunk of MC language code which has been abstracted away into a named entity. It is then possible to instantiate copies of this code by referring to its name. The code that *calls* this entity can itself be a similar entity. Thus, it is possible to have hierarchies of function, where each higher-level function is built using calls to lower-level functions or building blocks. These functions are *abstract* entities: they are pieces of code which are have no meaning outside the processing in MC. When this processing is complete, all function calls have been resolved and the result is a flat description.

A function has two aspects to it: the function definition (the code) and a function call (a reference to the code). In the MC language, the function definition is very similar to a *module***.** A module is actually a special function which always appears at the top and cannot be called like a function. The following example converts the earlier example of the sum-of-products into a function definition with some minor editing. It computes $\sum X*Y_i$ using bit widths that are passed in.

Example 4-17 Function Definition

```
function productSum (Z, X, n, repl(i , n, ",") { Y{i}});

     /* declare inputs and outputs
     */
     integer n;
     output Z;
     input X, replicate(i = 0; i < n; i = i + 1) { Y{i},};

     /* generate X * Y0 + X * Y1 + X * Y2 ...
     */
     Z = replicate(i = 0; i < n; i = i + 1; "+") { X * Y{i} };

endfunction
module test (OUT, A, B, C);

     output [15 : 0] OUT;
     input [7 : 0] A, B, C;
     productSum(OUT, A, 2, B, C);

endmodule
```

This example first replaces **module** and **endmodule** with “function” and “endfunction” and then gives this function a more meaningful name, “productSum”. Finally, it modifies the interface definition to exclude *param*, *in* and *out.* This is because the inputs to a function are a given: a function cannot create its own input. The function still declares its inputs, but without any attributes (which are determined by the caller.) The attributes for the output can be determined by the function, but in this case, we leave that up to the caller as well, so the output declaration is without attributes. This function can now be called as shown below. The following example computes the value of *out* using a function call that maps *out* to *z* in *productSum*. It maps 2 to *n*, A to *x*, B to Y0, and so on.

User-Defined String and Integer Functions

By default, user-defined functions do not have return values, although MC supports a syntax that gives the appearance of returning a signal value. You can create functions which return either a string or an integer value by inserting the keyword **string** or **integer**, respectively, before **function**. The **return** keyword is used to specify the return value of the function. An example of an integer function and a string function is shown below.

Example 4-18 Using Integer Functions and String Functions

```
string function adds (X,Y);
integer X,Y;
return (string(X,"+",Y));
endfunction

integer function sum (X,Y);
integer X,Y;
return (X+Y);
endfunction

module adder(a,b,X,Y,Z);

integer a,b;
input [7:0] X,Y;
output [7:0] Z;

if (sum(a,b)>8) {
     warning ("sum of a and b is greater than 8,
got:",adds(a,b),"\n");
}
Z=sum(a,b);
endmodule
```

Function Argument Lists

MC supports both complete and incomplete argument lists for functions. A function with a complete argument list must be called with the same number of arguments as declared in the function. In some cases, it is useful to be able to call a function without specifying values for some optional declared arguments. These incomplete argument list functions are identified by placing **VAR** after the function name. A function can never be called with more arguments than were declared.

Example 4-17 shows a complete argument list function, *productSum*(). The arguments to this function are matched with the interface definition for the function by position, starting from the left. The number of arguments as well as the type of arguments, must match. For instance, in the example above, it would be an error to use *productSum(Z, A)* or to attempt to pass a non-integer value for *n*. One notable exception to this rule is that an integer or an integer constant can be passed in where a signal input is expected.

By default, functions must be called with complete argument lists. MC then performs extensive error checking on the number and type of arguments passed to the function. Functions with incomplete arguments lists must be more carefully coded to deal with missing arguments. Therefore, we recommend using complete argument lists whenever possible.

Note that **if, repl** and **replicate** can be used in either complete or incomplete argument lists to create variable length lists. A special function, *fnArgs*(), is available within a function to facilitate the construction of variable length lists. It returns the total number of arguments supplied to the function. In the example below, the function *sum*() accepts any number of arguments by using *fnArgs*() and **repl**.

Example 4-19 Using *fnArgs*() with **repl** in Functions

```
function sum (Z, repl(i,fnArgs()-1,","){X{i}});
    input repl(i,fnArgs()-1,","){ X{i}};
    output Z;
    Z=repl(i,fnArgs()-1,"+"){X{i}};
endfunction

module adder (Z,A,B,C);
    input [7:0] A,B,C;
    output [7:0] Z;
    Z=sum (A,B,C);
endmodule
```

The example below shows a function *foo* with an incomplete argument list. Note the insertion of the keyword **VAR** between the function name and the argument list. This useless function can be called with 2, 3 or 4 arguments. By default, **num** has the value 5 and *B* is the inverse of *A*. If the caller supplies a value for either of these, the supplied values override the defaults.

Example 4-20 Using the Keyword **VAR** in Function Argument Lists

```
function foo VAR (Z, A, B, num);
     integer num=5;
     input A;
     if (fnArgs()>2) {
input B;
     } else {
wire B=~A;
     }
     output Z=A+B+num;
endfunction

module test (Z1, Z2, Z3, A, B);
     output [7:0] Z1, Z2, Z3;
     input [7:0] A,B;
     Z1=foo(A);// Z1=A+~A+5;
     Z2=foo(A,B);// Z2=A+B+5;
     Z3=foo(A,B,3);// Z3=A+B+3;
endmodule
```

Function arguments fall into one of the following classes:

Constant Arguments

These arguments are declared as **integer** or **string** inside the function. The caller must pass in a matching value. This value can be modified by the function, but this has no effect in the caller.

Signal Inputs

These arguments are declared as **input** inside the function. The caller can pass in a signal or an integer value. The signal must have a width. The function must not assign to the inputs again. If the declaration contains a width and/or format, the width and/or format of the signal passed in is expected to match that in the declaration. If the **Strict Parsing** option is enabled, a warning is generated if any mismatch occurs. In any case, when a mismatch occurs, a temporary operand is generated to convert the width of the signal passed to the function to that declared in the function as follows:

temporary = input from caller;

value used in function = temporary

Feedback Inputs

These arguments are declared as **input fb** inside the function. Feedback inputs behave the same as normal inputs except that these inputs are points that MC can use to break a loop. You should not need to use this feature except to allow the creation of continuous time loops.

Suppose we really want to create a circuit with a continuous time loop as shown in the example below. MC synthesizes the loop starting with the inverter input. Of course, timing estimates for this circuit will not be meaningful (MC will report the delay through one pass of the loop starting at the feedback input).

Example 4-21 Using Feedback Inputs

```
function delay (Z,A);
    input fb A;                can break loops at this input
    output Z=~A;
endfunction

module loop (Z,A);
    input [0:0] A;
    output [0:0] Z;
    Z = delay(Z&A);            loop created here
endmodule
```

Signal Outputs

These arguments are declared as **output** inside the function. The caller must pass in a signal. There are two types of functions and outputs:

- There are functions which have a good idea of what their output must be. For example, a register function knows that the output should be as wide as the input. You can call these functions with an output which does not have a width and the function assigns a width to the output. If you call the function using an output that has a width different from the expected width, a temporary is created as follows:

  temporary = result of the function;

  output from caller= temporary

- There are functions which require the caller to specify the output width. For example, an up-counter function which requires the caller to specify the upper limit on the counter. It is an error to call such functions with an output that does not have a width.

In all cases when a variable is passed into a function, it is substituted in the place of the argument that it was matched to. Then whenever the argument is referenced, the name of the variable which was passed in is used. So, in the example below, the statement *info("name of Z is: ", Z, "\n");* prints "name of Z is OUT".

These rules are illustrated in the following examples. In the first example note that both the function and the caller have integer variables with the same name. These variables are quite distinct. Second, the integer variable which is passed to the function is modified by the function, but its value does not change in the caller. Third, the caller declares *SIG* without a width and passes it to the function as an output; the function then creates the output by assigning it the appropriate attributes. Fourth, the function assigns to *Z*, thereby assigning to *SIG*. The caller can now use *SIG* to compute something else.

Example 4-22 Deferred Declarations

```
    module test (OUT, A, B, C)
        integer dummy = 1;
        integer w = 16;
        output [7:0] OUT;
        input [7:0] A, B, C;
        wire SIG;
→       product (SIG, A, B, C, w);
        ...
        info ("w is: ", w, "\n");
        OUT = SIG << 2;
    endmodule
→   function product (Z, X, Y0, Y1, param);
        integer dummy = 100;
        integer param;
        output [param-1: 0] Z;
        input X, Y0, Y1;
        Z = X * Y0 + X * Y1;
        param = 0;  // change local param
        info("param is: ", param, "\n");
        info ("name of Z is: ", Z, "\n");
    endfunction
```

Example 4-23 is a modified version of Example 4-22. Here, note that a constant, 16, is passed in the place of X. This is quite legal. Second, the caller defines SIG as an 8-bit quantity and the function defines Z as a 16-bit quantity. A temporary signal is created to convert the 16-bit Z from the function to the 8-bit SIG in the module.

Example 4-23 Overriding Function Declarations

```
         module test (OUT, A, B, C)
             integer w = 16;
             output [7:0] OUT;
             input [7:0] A, B, C;
             wire [7:0] SIG;
→            product (SIG, 16, B, C, w);
             ...
         endmodule
→        function product (Z, X, Y0, Y1, param);
             integer param;
             output [param-1: 0] Z;
             input X, Y0, Y1;
             Z = X * Y0 + X * Y1;
         endfunction
```

Local Variables

Since the code representing a function is copied into the caller, the naming scheme for locally created variables—variables created using **wire**—has to be such as to allow unique names only. Usually this is done by prefixing the local variable name with the output name or the left-hand-side of the expression which called the function. Sometimes it is also necessary to append a unique integer to the name. This name is reflected as such in the synthesis results as well.

The naming scheme is described in detail in Chapter 10.

Calling Conventions

The example above shows one style of calling a function. An alternate style of function call is in the form, *X = name(...)*. This style assumes that the output of the function is *X*. So, in the examples above, the call to the product function could be replaced with *OUT = product (A, B, C)* without any change in results. To use this style, the function must have one or more outputs and the first parameter must be an output.

Yet another style of a function call is to attach an “instance name” to the function. This style takes the form *X = name instance_name(...)* or *name instance_name(...)*. In either case, the instance name is used as the prefix string in naming the local variables of a function. This is useful for identify and collect all the signals which were generated by a particular call to a particular function.

Functions can be embedded in expressions. In this case, the output of the function is implicitly created as a temporary variable. In order for this to work correctly, the width of the output must be declared inside the function.

Example 4-24 Function Calling Conventions

```
     module test (OUT, A, B, C)
          integer w = 8;
          output [7:0] OUT;
          input [7:0] A, B, C;
          wire U, V, W;
→         product(U, A, B, C, w);
          // virtually identical to the call above
          V = product (A, B, C, w);
          // use an instance name. 'temp' in the
          // function call will be named
          // call1_temp
          product call1 (W, A, B, C, w);
     endmodule
→    function product (Z, X, Y0, Y1, param);
          integer param;
          output [param - 1: 0] Z;
          input X, Y0, Y1;
          wire [7:0] temp;
          temp = Y0 + Y1;
          Z = X * temp;
          info ("name of Z is: ", Z, "\n") ;
          info ("name of temp is: ", temp, "\n");
     endfunction
```

Built-in and Library Functions

Module Compiler comes with a set of integer and string functions that are built-in functions. They are hard-coded into MC and cannot be redefined by the user. The *width*() function is an example of a built-in function. It accepts a signal and returns its width. Other such functions are *formatStr*() which returns *signed* or *unsigned* depending on the format of the signal passed in, *log2*() which return log-base-2 of an integer. The comprehensive set of built-in functions is provided in the Reference Manual. You can also define your own integer and string functions.

MC also includes a library of signal functions, which are defined in a library file. These functions implement primitives that are not representable using operators. You can redefine a library function, although the practice is not recommended.

Table 4-5 lists some examples of library functions.

Table 4-5 Examples of Library Functions

Function	Action
, *cat*()	Concatenates bits of different signals to create a new signal
sat()	Clips the outputs to given values.
sreg()	Creates a state register, etc.

The comprehensive set of these functions is also provided in the *Module Compiler Reference Manual*.

Each technology library cell and all cells and netlists loaded by the user at startup also have a corresponding function in the MC generic cell library.

Errors

The compiler is designed to keep parsing the input even if there are errors. However, after a certain point, error messages begin to lose their meaning. The compiler uses the following rules to prune error printing:

- If more than 5 errors, then quit
- If error in a function, then do not explode function calls originating in this function
- If there are any errors, then do not produce any output

Error pruning can be disabled by using the Verbose mode. See “Report Generation (The Reports Menu)” in Chapter 3.

MC prints out three types of errors. These types are:

- System Errors

  Examples are failure to open a file, running out of memory, etc. These errors are usually fatal.

- Syntax Errors

  These pertain to grammatical errors, such as missing “,” or missing “;” or wrong number of arguments for a statement like “wire ...,” etc. MC usually keeps going when it encounters such an error, but it skips the rest of the line.

- Semantics Errors

  These pertain to invalid variable contents, such as multiple definitions, multiple assignments, etc. Most semantic errors like multiple assignments, etc. are printed only once for a given variable.

MC also produces system warnings and supports user-produced errors, warnings and info messages.

Error messages try to convey the location of the error by printing two consecutive lines from the input. *The error is typically in the second line or after it; the first line is provided only as context.* Error messages also print out the name of the offending function and the file containing this function. A line number is also provided for easy reference. If the **verbose** option is set, then errors are accompanied by a “stack-trace” showing all the function calls leading up to the point of failure.

There are some semantic errors which cannot be trapped by the MC language parser. These errors are trapped by the synthesizer and are reported using similar contextual information.

Using the Module Compiler Language

5

Chapter 5 is a guide for using the MC language to build datapaths. The constructs of the MC language are described in detail in Chapter 4. This chapter focuses on the semantics: how the synthesized result is impacted when you use a particular construct in a particular manner.

Chapter 5 discusses the following topics:

- Guidelines for appropriate usage
- Built-in operators and functions
- Directives, attributes, sequential circuits, and registers
- Sequential circuits and registers
- The generic cell library
- Using groups
- Inserting cells
- Controlling reports

Module Compiler Language Details

Modules

The **module** construct specifies the interface and the contents of the design to be generated by MC.

Naming

The module name is used by default as the root name of all output files that are unique to the design. The Verilog simulation models have signal declarations in the same order as the signals in the module parameter list.

The module name can be changed using the **modname** attribute. This is particularly useful for preventing name collisions when you are constructing many modules from the same description. You can pass the module name in as a parameter, or the name can be generated internally to the module using the existing set of parameters.

Example 5-1 Using **modname** to Change the Module Name

```
module dummy (X,Y,Z,a);
input [7:0] X,Y;
output [7:0] Z;
integer a;

directive (modname=string("goober_",a));          change the module name

...

endmodule
```

I/O Constraints

Directives are used to specify the external loading and timing constraints for inputs and outputs of a module. The maximum loading allowed at an input is indicated by **inload**. MC does not put more than this load value on the inputs. The arrival time of the input is specified with **indelay.** If **indelay** is positive, it indicates an input arriving later than the default (0), making paths from that input more critical. Any negative values are treated as minus infinity, making paths from that point noncritical. The load associated with the output is indicated by the **outload** attribute. This load is placed on the driver of the output. Any external path delays are specified with **outdelay**. These delays are in the circuit following the MC synthesized circuit and are added to the MC path delay. Greater output delays result in greater net criticality. Negative output delays are not allowed. All load values have units of 0.1 standard loads, while delays have units of picoseconds.

Example 5-2 Input Arrival and Default Loading

```
module test(X,Y,Y1,Z,Z1);                 test has no parameters, only signals

    input [0:0] X;                        X is one-bit, default format, 0 arrival, default load

    directive (indelay=9000,inload=400);
    input signed[9:0] Y,Y1;               Y,Y1 can only have 40.0 stdloads, arrive at 9ns

    output [9:0] Z;
    directive (outdelay=10000,outload=400);
    output [5:0] Z1;                      Z1 has load of 40.0 stdloads, additional delay of 10ns
    ...
endmodule
```

Module Parameters

There are two related ways of passing in integer and string parameters that are specified in the interface definition of a module. At the command line, use the **-par** option to specify all declared parameters. Module parameters that have default values need not be included in the list. The general form uses comma-separated parameter name and value pairs, with no space separators as shown below:

```
-par <par>=<val>[,<par>=<val>*]
```

In the GUI, place the information in the Parameters entry field of the main window. Because both interfaces use the same syntax, parameters set in either interface carry over to the other.

Note: No spaces are allowed anywhere in the parameter list.

MC automatically determines the type of each parameter. Any parameter that does not appear to be a number is passed as a string, and any parameter that is a number is passed as an integer. This means that you cannot use numbers as a value of a string parameter.

Constants

Four types of constants are supported, which can be used in both integer and signal expressions. Negative constants are always signed, while positive constants are always unsigned. The *don't care* constant is provided for use in multiplexors.

The following types of constants are defined.

Table 5-1 Types of Constants

Constant	Example	Restriction
Decimal	–32, 879	limited to 32 bits
Binary	’b110011	limited to 1024 bits
Hexadecimal	’hffff, -’h100a	limited to 1024 bits
Octal	’o3777, -’o1066	limited to 1024 bits
Dont care	’hx	only used in MUX

Integer Variables

Integer expressions, described in Chapter 4, are resolved at MC runtime and do not cause any hardware to be built. They are used to parameterize and control the replication of objects that result in hardware. Mixed integer and signal expressions do result in the construction of hardware. The value of the integer portion of the expression is a constant in the hardware.

Operands and Constants

An operand is a variable that participates in an operator expression. In the context of MC, an operand is a signal variable or a signal constant. All operands have signed or unsigned formats and you can usually choose any range of bits as the input to a function. The format of signals can be declared explicitly; if not, then the format is unsigned. For all operands, the MSB is always the highest numbered bit; bit numbers always start from 0. For both constants and variable operands, any bit range including the MSB of a signed operand is also signed, otherwise it is unsigned.

Suppose X is a 10-bit signed number, then the following bit ranges are signed:

```
X
X[9:0]
X[9]
X[9:5]
```

The following bit ranges are unsigned:

```
X[8:0]
X[4:3]
X[0]
```

The format of input and wires should be chosen carefully, since many functions and operators use the formats of the operands to determine the synthesized structure. Some functions, such as *sat*(), do nothing more than convert formats and bit widths. For example, a multiplier is synthesized differently if the inputs are signed rather than unsigned. This should be clear because, for 4-bit numbers, 1111*1111 = 11100001 (225) and 00000001 (1) if the inputs are unsigned and signed, respectively.

Note that the use of **[ ]** to indicate a bit range is quite different when an operand is being used, compared to when an operand is being declared. If no range is specified when an operand is used, the entire operand is used. If no range is specified when an operand is declared, the operand is created with an undefined width and the width is determined later. Similarly, X[0] means the 0 bit of X. To declare an operand with only one bit, use X[0:0].

Bit ranges are not allowed on the left hand side of an expression.

Temporary Operands

Operands are combined into expressions using functions and operators. Expressions are considered native when they map into a single functional unit in the synthesizer. These expressions generally have very efficient implementations. When nonnative expressions are encountered, the expression is broken down into a sequence of native expressions. The intermediate or temporary operands are created automatically in all simple cases. The rules for creating these operands have been described in a previous chapter.

Because the width rules for the temporary operands do not provide for significant bit width increase, those operators that naturally result in a bit width increase may perform in an unexpected manner when a temporary operand is created. The operators that result in bit width increases include: *****, **+** and **-**. You can prevent temporaries from being created for these operators by setting the **autotemp** synthesis attribute to **safe** (default). To prevent any temporaries from being generated, set **autotemp** to **off**. To allow the generation of temporary operands for all operators, set **autotemp** to **all**.

Note that the right and left shift of a signal by an integer results in bit-width changes, but these changes can be computed unambiguously in all cases. The resulting temporary variable is always the same format as the signal and the width is the width of the signal plus or minus the left or the right shift, respectively.

There are cases where the width and or format of the temporary operands cannot be determined unambiguously. Then the size operator can be used to specify the width and format of the temporary operands. For example., (unsigned [7:0]) (*expression*) causes the result of *expression* to be assigned to a temporary variable of **unsigned** format and width 8.

Library Functions

The MC language supports a rich set of signal operators, as described in Chapter 4. However, the signal operators alone are insufficient to describe many interesting designs. For instance, there is no operator notation to describe a register. MC provides a library of functions for this purpose. Some of these functions are synthesis primitives while others are built using these primitives. Some direction regarding the interpretation and use of a selected set of these functions is provided in the following sections. The complete and definitive source for the usage is the *Module Compiler Reference Manual*.

The library of MC functions is expected to grow over time. Some of the signal library functions—those that represent synthesized hardware—are summarized in Table 5-2.

Table 5-2 Signal Library Functions

Function	Description
accum(output Z, input X, input R, input S)	accumulator
AccPM(output Z, input C, input X, input Y, input ADD, input XS, input YS);	Z=C +/- X*Y
alup(output Z, input A, input B, input DI, output DO, input CI, input INST, output FLAGS, input FirstCyc, integer inst Mask);	Programmable 16 instruction ALU
asyncRF(integer p, integer words ....)	multiple read 1-write port asynchronous netlist RAM (similar to ram1p)
bitrev(output out, input in);	reverse bits (MSB <-> LSB)
buffer(output out, integer depth);	set buffer depth for operand
cat VAR(output Z, input D0, ... , input Dn)	concatenate
convert(output Z1, output Z2, input X);	convert carrysave X to binary (Z1,Z2)
count(output Z, input X, input R, input S, integer detectOVF, output OVF)	counter
crc(output Z, output ERR, input X, input R, input GEN, integer Degree);	CRC encoder/decoder
decode(output out, input in)	decode in to out
demux VAR(input in, input select, outputlist out);	demultiplex in by factor width
ensreg VAR(output out,input in,input en, integer len, output tap0, ...);	shift-hold state register
eqreg(output out,input in,integer len, inputlist ref);	increase latency, set to maximum of ref list
eqreg1(output out, input in, integer deslat);	increase latency, set to deslat
eqreg2(output out, input in,integer len, inputlist ref);	increase latency, set to sum of the latencies of the reference operands
fir(output Z, integer len, input X, inputlist Y)	fir filter with len taps
hidelat VAR(output out, input in, integer numref, inputlist ref);	hide latency, set to minimum of ref list
isolate(output out, input in);	isolate output load from input
join VAR(output Z, input D1, ... input Dn);	bit-wise join all inputs
latch(output Q, input D, input G);	positive gate latch
nlatch(output Q, input D, input G);	negative gate latch
mac(output Z, input X, input Y, input R, input S)	multiplier-accumulator
maccs(output Z, input X, input Y, input R, input S)	multiplier-accumulator (carrysave)
mag(output Z, input X)	z=abs(x)
max2(output Z, output XGEY, input X, input Y)	z= max(x,y), XGEY=(x>=y)
maxmin(output Max, output Min, output XGEY, input X, input Y)	Min=min(x,y), Max=max(x,y), XGEY=(x>=y)
min2(output z, output XGEY, input x, input y)	z= min(x,y), XGEY=(x>=y)
multp(output Z, input X, input Y, input W);	Z=X*(Y+W)
norm(output mant, output exp, input in);	normalize leading zeros
norm1(output mant, output exp, input in);	normalize leading ones
preg VAR(output out,input in,integer len, outputlist taps);	pipeline register
sat(output out,input in);	saturate
sati(output out,input in);	saturate (inverted output)
sgnmult(output z, input x, input y)	sign multiplier, x or y must be 1-bit signed
shiftlr(output z, input x, input shift, input left, input log)	shift left/right logical/arithmetic
sreg VAR(output out, input in, integer len, outputlist taps);	state register
syncRF(integer wp, integer rp, integer words ...)	multiple read/write port synchronous netlist RAM

Directives and Attributes

The MC language provides a **directive** construct which sets the value of synthesis attributes that influence the synthesis and optimization processes but do not affect the function performed by the circuit. Use a comma separated list to change multiple attributes in one directive statement.

The current set of MC attributes is summarized below. Defaults listed as “cmd line” are determined by the GUI, the command line, and by MC environment variables. When an attribute is set to **auto**, MC makes a context-sensitive choice.

Table 5-3 MC Directives

Attribute	Description and Values	Default
acswitch	ac switching percentage for power calculations	50
autotemp	automatic temporary operand generation, **on**, **off**, **safe**	**safe**
carrysave	enable or disable carrysave generation: **on**, **off**, **convert**, or **optimize**	**off**
clock	name of the current clock	**CLK**
dcduty	dc duty cycle percentage for power calculations	100
dcopt	enable/disable optimization by Design Compiler: **on**, **off**	**off**
delay	current delay goal in ps	cmd line
delstate	control pipeline loaning	0
dirext	direct sign extension mode for sum operation: **on**, **off**	**off**
fadelay	final adder desired delay: for **csa**, **clsa** only	current delay goal
fatype	final adder architecture: **csa**, **cla**, **fastcla**, **clsa**, **ripple**, **auto**	**auto**
group	current group name	misc
indelay	input operand arrival time in ps	0
inload	input operand maximum load in 0.1 stdloads	cmd line
intround	internal rounding at position; 0 for no rounding	0
logopt	logic optimization mode: **on**, **off**	**on**
maxtreedepth	Wallace tree maximum depth	infinite
modname	set module name to string value provided	from module decl
multtype	multiplier architecture: **booth**, **nonbooth** or **auto**	**auto**
muxtype	MUX architecture: **mux**, **andor**, **tristate**	**mux**
outdelay	output operand delay in ps	0
outload	output operand load in 0.1 stdloads	cmd line
pipeline	pipelining mode: **on**, **off**	cmd line
pipeslack	pipeline slack in ps	0
pipestall	name of stall signal	none
round	simple biased round sum at position; 0 for no rounding	0
scan	scan test mode: **on**, **off**	**off**
selectop	optimization mode for MUXs and shifters: **msb**, **lsb**, **auto**	**auto**

Synthesis attribute changes can be global, local, or default in scope as described in Chapter 4. That is, once a synthesis attribute is set, it affects all statements executed after the change within the appropriate scope of the input description hierarchy, until a contrary directive is executed.

All attributes affect operands or operators. Operators are affected by the values of synthesis attributes in effect when the operator is used. Input and output operands are affected when they are declared rather than assigned.

Example 5-3 Examples of Directives

```
input [7:0] X, Y;
wire [7:0] Z, ZM;
wire [3:0] ZS;

// Z is in group SUM and can be pipelined to 10 ns
directive (group="SUM");
directive (delay=10000, pipeline="on");
Z=X+Y;

directive (logopt="off");
// ZS cannot be optimized, pipelining still on
ZS=sat(Z);

// new group, no pipelining, not critical
directive (logopt="on");
directive (group="MULT", pipeline="off", delay=99999, multtype="nonbooth");
ZM=ZS*ZS;

directive(acswitch=40,dcduty=25);
```

Assignment Operator

Assignments (=) are used in two ways: either the result of some operation is assigned to an operand, or one operand is assigned to another operand. The first form of assignment is handled by the specific operation involved. The second form of assignment is used to simply copy bits from the source operand to the destination. All bits from the source that fall within the bit range of the destination are copied to the bit with the same value. Bits from the source that fall outside the bit range of the destination are discarded. Signed sources are sign-extended when assigned to a wider destination. Unsigned sources are zero extended under the same condition. The format of the destination does not affect the operation in any way. In fact, assignment is a convenient way to perform format conversion.

Assignment does not check for overflow and truncation, potentially causing large errors. You should use the *sat()* function in circumstances where you want to map the source into the nearest legal value of the destination.

Pure assignment always converts a carrysave signal to binary. To copy a carrysave operand, use the + operator rather than = alone.

Operators and Functions Based on Addition

The addition operators and functions are the most complex and versatile of all the MC functions. They are used to implement any function requiring general addition, including subtraction, multiplication, incrementing, magnitude comparison or any combination of these operations. The sum is implemented in three distinct steps: the generation of addends, the Wallace tree reduction of the addends to a carrysave value (two signals per bit position) and the final or carry-propagate addition that reduces the carrysave value to a true binary (one signal per bit position) result.

Due to its complexity, most of the synthesis details of sum are discussed in “Arithmetic Computation” in Chapter 9.

Synthesis Attributes Affecting Addition Operators

Table 5-4 shows several attributes that affect the synthesis of these functions.

Table 5-4 Synthesis Attributes Affecting Addition Operators

Synthesis Attribute	Description	Values
fatype	final adder type	auto, csa, fastcla, cla, clsa, ripple
fadelay	final adder delay goal in ps	only for csa and clsa types
multtype	multiplier type	auto, booth, nonbooth
maxtreedepth	maximum Wallace tree depth	3=>serial, large value =>parallel
dirext	force direct sign extension	on, off
carrysave	carrysave mode	on, off, convert, optimize
round	round result to given position	integer values
intround	internally round arithmetic operations	integer values

The **maxtreedepth** synthesis attribute is used to limit the depth of the Wallace tree used to implement these functions. As the value of **maxtreedepth** decreases, the implementation becomes more serial and slower. As expected, the serial structures are slower than the parallel structures and the areas of the two often appear to be very similar. However, after place and route, the serial structures would be expected to have a higher utilization than the parallel ones. For most structures, this attribute should not need to be changed. If poor utilization is observed, try reducing **maxtreedepth**. The minimum Wallace tree depth is 3.

It is also possible to bypass the final addition to achieve area and performance improvements. This is accomplished by setting the **carry/save** synthesis attribute. This is further described in the Carry/save section below and in Chapter 9.

To use direct sign extension, set the **dirext** synthesis attribute to **on**. To round the result to the *n*th bit, where bit *n* is the new LSB, use the **round** synthesis attribute. The **round** attribute should not be used with accumulator (recursive) structures.

The multiplier architecture is specified using the **multtype** synthesis attribute. When **multtype** is set to **auto**, the Booth architecture is employed if the X and Y inputs have at least 16 bits combined; otherwise non-Booth architecture is used. The relative advantages of Booth and non-Booth architectures are discussed in “Multiplication” in Chapter 9 and in the *Module Compiler Reference Manual*.

The **fatype** synthesis attribute can be used to specify the final adder type. When **fatype** is set to **auto**, its value is set as shown below. Use the **fadelay** synthesis attribute to specify the delay goal of the final adder.

Table 5-5 Final Adder Type when **fatype** Is Set to **auto**

Condition	fatype
pipeline=on	cla
pipeline=off, optimization for speed	fastcla
pipeline=off, not optimizing for speed	clsa

Of the final adder types, the **clsa** is a good general choice, particularly with large delay skews, but it does not pipeline well. It is by far the most flexible architecture and can automatically create structures ranging from a **ripple** adder to a **fastcla** adder, depending on the desired delay. **csa** is not a particularly high performance adder, ideally achieving only O($\sqrt{n}$) delay. However, the **csa** often works well in pipelined circuits that have large delay skews, for example, a pipelined multiplier or FIR filter. In reality, the growing loading on the carry select lines degrades performance below the expected level. The **fastcla** is usually the fastest architecture, but is also the largest. The **cla** uses a sparse carry tree that roughly doubles (actually, $2(\log_2(n)-1)$) the delay in the carry tree relative to the **fastcla**, but provides significant area savings. The **ripple** adder is the smallest and slowest adder structure and is useful in noncritical portions of the design.

Example 5-4 Examples of Expressions that Use Addition

```
X=A+B;

X=A-(B[6:1]<<3);

directive(multtype="booth");
X=A*B;                                    uses a booth multiplier

directive(multtype="nonbooth");
X2=A[7:4]*(B[3:0]<<2);                    uses a nonbooth multiplier
X3=-A[7:4]*(B[3:0]<<2);                   ditto

wire [31:0] X;
directive(round=4);
X=A+B;

wire signed [31:0] Z;                     use default adder if Z is sum output
Z=X+Y;

directive(fatype="clsa", fadelay=4000);
wire signed [31:0] Z;                     4.0 ns clsa adder
Z=X+Y;

directive (fatype="csa");
wire signed [31:0] Z;                     csa, default delay goal
Z=X+Y;

directive (fatype="fastcla");
wire signed[31:0] Z;                      use fastcla
Z=X+Y;

directive (fatype="fastcla");
wire [31:0] X;
wire F;
X=A*B+C*(D<<2)+E*F-(G[8:0]<<1)+H*I[7]+K+L;
```

Functions Based on Addition

Functions based on addition consist of *sgnmult*(), *multp*(), and *mag*(). The *sgnmult*() function is used to multiply a signal by plus or minus one that is represented by a second single-bit signal. This function can be used to generate a carrysave output if the carrysave attribute is set appropriately. The *mag*() function is used to compute the absolute value of an operand.

Example 5-5 Examples of Expressions that Use *sgnmult*() and *mag*()

```
Z0 = mag(X);                         Z = -X if X < 0, Z = X if X > 0
Z1=sgnmult(X,S);                     Z1=+X if S=0, Z1=-X if S=1
directive (carrysave = "on");
Z2 = sgnmult(X,S);                   Z2 is a carrysave
```

Carrysave

The final stage in all addition-based operations consists of reducing the carrysave value (two signals per bit position) into true binary result by employing a final adder. It is sometimes desirable to skip the final reduction and leave the result in carrysave format in cases where a final adder will be employed further downstream.

A carrysave signal may be generated whenever **+**, **–** and ***** operators are used. The directive attribute **carrysave** is used to control carrysave generation. If the attribute is set **on**, then normal carrysave operands are created. Values of **convert** and **optimize** are used when connecting the carrysave operand to the convert function and to minimize the computational burden of the following addition, respectively. In general, these options should not be needed. Setting the attribute to **off** causes the carrysave generation to be disabled. Carry propagate adders are used to form true binary results.

See “Carrysave Operands” in Chapter 9 for a further discussion of Carry/save operands.

Logical, Reduction, Shift, and MUX Operators

Logical Operators: AND, OR, and XOR

These operators compute bitwise logical functions over all inputs. As with the addition operators, any number of inputs can be accommodated and degenerate cases are handled efficiently.

These operations are implemented with the &, | and ^ operators, respectively. Each of these operators generates a single Wallace tree regardless of the number of operands, even if some terms are inverted (~). Multiple operations produce one Wallace tree for each function—one for each temporary operand generated.

Example 5-6 Logical Operators and Wallace Tree Generation

```
wire signed [7:0] X;
X=~A&B&C&~(D[6:0]<<2);                          single Wallace tree

wire X;
X=A[0]^A[1]^A[2]^A[3];                          single Wallace tree

wire X;
X=~A[0]&B[1]&C[0]&(A[0]^A[1]^A[2]^A[3]);        two Wallace trees
```

Suppose we have the following example with values for A, B and C shown.

Example 5-7 More Logical Operators

```
wire [7:0] A;
wire signed [3:0] B;
wire signed [7:0] C;
wire [8:0] Z1,Z2,Z3;

Z1=(~(A<<2))&B&C;
Z2=(~(A<<2))|B|C;
Z3=(~(A<<2))^B^C;
```

Input	Value
A	11100010
B	1000
C	10101010

After shifting, sign extending, and inverting, the Wallace tree inputs are as follows:

Wallace Tree Inputs	Value
~(A<<2)	001110111
B	111111000
C	110101010

Output	Value
Z1	000100000
Z2	111111111
Z3	000100101

Reduction Operators

MC provides three unary reduction operators: reduction **AND** (&), reduction **OR** (|), and reduction **XOR** (^). These operators are used to reduce multibit operands to single-bit objects. The result is derived by applying the corresponding binary operator to each bit of the multibit operand in a pairwise fashion. In the following example, the two statements are exactly equivalent for an 8-bit operand, X:

```
wire [0:0] Z = ^X;
```

and

```
wire [0:0] Z = X[0]^X[1]^X[2]^X[3]^X[4]^X[5]^X[6]^X[7];
```

Comparison Operators

The complete set of comparison operators (==, != , >, <, >=, and <=) is supported.

The Equality Test

The equality test is implemented by the binary operator, ==. It requires two inputs, which can have any combination of signed and unsigned formats. The output is always a single-bit unsigned value that is 1 if the two inputs are equal and is 0 otherwise. The two inputs can have different widths.

```
wire [0:0] Z;
Z=A==B;
wire [0:0] Z1;
Z1=(A[7:0]<<1)==B;
```

This operation always treats the two inputs as integers, strictly observing the data formats. For example, if signed and unsigned inputs are compared, the signed input must be positive for the two to be considered equal.

The Not-Equal-To Test

The not-equal-to test is implemented by the != operator. This operator is identical to an equality test (==) followed by an invert (~).

Other Comparison Operators

The remainder of the comparison operators utilize subtraction; the decision is the sign bit of the result of the subtraction. These operators can perform comparisons such as (A + B) >= (A*C - B*D) using a single adder. The width of the operand used to hold the result of the subtraction is computed as the maximum width:

$$\log_2\left(\sum 2^{w_i} + \sum 2^{(w_j + w_k)}\right) + 1$$

where w_i are the width of the **+** and **–** operands and the w_j and w_k are the widths of the ***** operands.

Equality Comparison

The equality comparison is performed by performing a bit-wise XOR between the two inputs and then a NOR of all XOR outputs using a Wallace tree.

Selectop

You can use the **selectop** attribute to control the ordering of select signals for shifters, rotators, and MUX-based multiplexors. When **selectop** is set to **msb**, the select inputs are ordered from MSB to LSB: the delay from the LSB is the least and the MSB is the greatest. When **selectop** is set to **lsb**, the ordering is reversed: the delay from the LSB is the greatest. When **selectop** is set to **auto**, MC orders the select inputs to minimize the delay from the select inputs to the output, based on the select input arrival times.

Rotate and Shift

The rotate and shift operators provide left and right shifters and rotators that work with both signed and unsigned data. The result is correct even when the input and output bit ranges do not match. The input data is always directly sign extended if the output is wider than the input. For shift, if the output is narrower than the input, the full precision output is truncated after shifting. For rotate, if the output is narrower than the input, the input is truncated to the width of the output and then rotated.

The shift operation uses the familiar >> and << operators for right and left shift, respectively. These operators always perform an arithmetic shift: an approximation to division for right shift and to multiplication for left shift (one value is a power of 2). If you require a logical shift of a signed operand, you must first convert it to unsigned.

The rotate operation uses the >>> and <<< operators for right and left rotate, respectively. These operators perform a cyclical rotation of the bits in either a left or right direction. Unlike shifters, in which bits shifted out the ends are lost, the rotators shift bits out of one end and wrap them around to the other end.

When the shift value is a constant, the shift or rotate output is computed in advance and no hardware is generated. Negative constant shift values cause a shift in the opposite direction. If the data input is constant, the logic optimizer is relied upon to reduce the area.

```
wire [31:0] X;
X=A<<<S;                          rotate left

wire signed [31:0] X;
X=~(A>>S);                        shift right and invert
```

Example 5-8 Examples of Shift and Rotate

The table below shows several functional examples in which the input, X, is shifted by a signal, S, with a value of 2. For this case, there is no fixed shift or bit ranging, and the output has the same width as the input.

```
X[5:0] = (b5,b4,b3,b2,b1,b0)
```

Function	Format	Output
Z=X>>S	unsigned	0 0 b5 b4 b3 b2
Z=X>>S	signed	b5 b5 b5 b4 b3 b2
Z=X<<S	either	b3 b2 b1 b0 0 0
Z=X>>>S	either	b1 b0 b5 b4 b3 b2
Z=X<<<S	either	b3 b2 b1 b0 b5 b4

Example 5-9 Another Example of Shift

If X is shifted by a constant 2, the shift left results are the same as above while the shift right results are as shown below. The result is the same for both signed and unsigned inputs.

Func	Output
Z=X>>2	b5 b4 b3 b2
Z=X<<2	b3 b2 b1 b0 0 0

The shifter and rotator are built with a sequence of 2-input multiplexor stages ($\log_2(n)$ stages, where n is the number of bits in the shift operand). To maximize speed and minimize area, inverting multiplexors are used in all stages, except the last if there is an odd number of stages. You can optionally specify that the output should be inverted.

Note: Inversion provides improvement of area and delay when there are an odd number of stages and degradation when there are an even number of stages.

Multiplexing

The multiplexing operation uses the **?:** operator. The signal used for selection is specified to the left of **?** and the list of signals to be selected is specified to the right of **?**. The signals to be selected are separated by colons (**:**). These signals are selected from **right to left** as the select input value increases from 0 to n–1. Thus, if the select signal is 0, the rightmost signal is selected. If the select signal is 1, the second rightmost signal is selected, and so on.

An n-bit-wide select signal can be used to multiplex 2^n signals. It is not necessary to specify the entire range of inputs: if only m inputs are specified, then the top m+1 to 2^n inputs are not connected. It is also possible to create holes by specifying *don't-cares* (using the constant **'hx**) for the corresponding input. The don't cares should be used when possible to decrease the area required to implement the multiplexor.

Multiplexor Architectures

You can select the architecture of the multiplexor using the **muxtype** attribute. If you set **muxtype** to **mux**, you get a MUX-based architecture. Other possible values are **andor** and **tristate**, which produce ANDOR-based and TRISTATE-based multiplexors. Each architecture accepts any number of inputs, each of any width and format. Signed inputs are directly sign extended as necessary.

MUX-Based. The MUX architecture is the best default choice and the most straightforward, because it provides generally good speed and area. It is constructed from standard MUX cells and is most likely to be similar to what you would produce manually. This structure cannot take advantage of skewed data input arrival times but does optimize the structure when the select inputs have skewed arrival times, as determined by **selectop**. If the select space is not full, meaning that fewer than 2^n data inputs are provided for an n-bit select input, then the unused select values are assumed to be don't cares and are used to minimize the area. The behavioral model outputs X if unspecified select values are used, while the gate-level model outputs one of the data inputs. In general, this structure produces efficient results under degenerate conditions.

ANDOR-Based. The ANDOR architecture decodes the select input and uses the decoder outputs to gate (AND) each data input. The gated data inputs are then ORed together using a Wallace tree. This structure has the advantage of being fully timing-driven and so should provide good performance for highly skewed input arrival times. However, it is generally larger than the MUX-based implementations and is also typically slower for inputs with no arrival time skew. If the select space is not full, the output is 0 if an undefined select value is used. In general, this structure produces efficient results under degenerate conditions and is commonly used with only one select value defined. In that case, the select value is essentially a reset control: when it is 1, the data input is selected, otherwise the result is 0.

TRISTATE-Based. The final multiplexor architecture is that based on tristate buffers. The decoded select inputs are used to enable a tristate driver for the selected input onto the output bus. Generally, you would expect very small data delays, particularly for large numbers of data inputs. However, the logic optimizer cannot optimize the tristate buffers and the increasing load at the output tends to limit the usefulness of this structure.

Example 5-10 Specifying Multiplexor Architecture

```
directive (muxtype="mux");
wire signed [7:0] X;
X=select[1:0] ? B : C[15:8] : D : A<<3;

wire [15:0] X;
X= ~(select ? B : A);

directive (muxtype="andor");
wire [15:0] X;
X=select ? B : A;

directive (muxtype="tristate");
wire [15:0] X;
X=select ? B : A;

X=select ? B : A : ’h x;  don’t care what is output when select=0
```

Decoding

MC provides a general decoder function that is used by two of the MUX architectures. You can also call the *decoder*() function directly. It uses single stage **AND** logic to generate each output. This approach is not particularly area efficient for wide decoders, but is reasonable in the range from 4 to 16 outputs. As the **AND** Wallace trees are used, this structure automatically adjusts to incoming delay skews.

Note: In a partial decoder where not all 2^n outputs are used, the remaining logic is not optimized to take advantage of this constraint, so you should not make the output range any wider than necessary.

Format Conversion Circuits

Format conversion circuits are used to convert wires from one datatype to another or from one format to another.

Saturation

The saturation function is used to convert an operand with one range of legal values into another operand with a smaller range of values. Operand bit-range selection is the simplest form of this conversion; all bits of the input that are outside the selected bit range are discarded. This approach produces potentially large errors and can result in instabilities in many recursive algorithms. The saturation function provides the minimum error conversion; the closest value in the output space to the original input is selected as the output. That is, if the input exceeds the maximum or minimum value representable at the output, then the output is set to the maximum or minimum value, respectively.

Two functions are provided for the saturation operation: *sat*() and *sati*(). Each function requires an input and an output. *sati*() inverts the final result whereas *sat*() returns the true result. Inverting the output generally improves both area and delay by allowing the use of an inverting rather than a noninverting MUX.

This function works with any combination of signed and unsigned operands at the input and output. The formats and bit ranges chosen for the input and output are very important, as *sat*() is nothing more than a conversion from the input bit range and format to the output.

Example 5-11 Using the Saturation Function

```
input signed [7:0] X;
wire signed [3:0] Z1,Z3;
wire [3:0] Z2;
wire [7:0] Z4;
Z1=sat(X[6:0]);          unsigned -> signed
Z2=sati(X);              same as Z2=~sat(X);
Z3=sat(X);               signed -> signed
Z4=sat(X[7:4]);          signed -> unsigned
```

Normalize

The normalization operation can be thought of as conversion from unsigned integer to floating point format. It detects the number of leading zeros or ones in the input and shifts the input left by this amount. The number of leading zeros or ones is the exponent of the normalized number and the shifted number is the mantissa.

This operation can be specified using one of two functions. Use *norm*() to remove leading zeros, and use *norm1*() to remove leading ones. In either case, the mantissa is the first operand, the exponent is the second, and the data input is the third argument of the function.

If the width of the data input is not a power of 2, the input is left shifted to make the width a power of 2. The shifted input is then normalized. Finally, the mantissa is right-shifted by the amount of the left shift.

The exponent is unsigned by default. It is also not allowed to exceed the declared range of the exponent operand; you should declare the width of the exponent operand to control the maximum number of shifts used to normalize the input. The input can be signed. Any sign extension needed is performed before the normalization.

When the mantissa output is narrower than the input, the computation is performed with full precision (input width) and then the MSBs of the full precision result are truncated to form the mantissa of the correct width.

Following are a few examples showing the operation of normalize (leading zeros) for an exponent operand 2 bits wide.

Example 5-12 Leading-Zero Normalization

Input	Output (mantissa)	Output (exp)
10000000	10000000	0
01110101	11101010	1
00001110	01110000	3

Note that in the third example, the output is not fully normalized because the exponent output was defined with only two bits. Hence, the maximum exponent (shift) is 3 and not 4.

Example 5-13 Normalization

```
wire [31:0] MANT;
wire [3:0] EXP;                          must be unsigned
norm (MANT,EXP,IN);

wire signed [31:0] MANT;
wire [3:0] EXP;
norm (MANT,EXP,IN[63:40]<<2);           uses a temporary for IN[63:40]<<2
```

Norm() and *norm1*() use Wallace trees in the computation of the shift value and so deals well with arrival time skews in the high order bits. A one level lookahead technique is used to speed up the computation of the shift value.

Sequential Circuits

Sequential circuits can be described concisely using library functions, which have the same general format and style as combinatorial functions. Automatic pipelining provides a mechanism for automatically inserting pipelines in the design to achieve the desired delay goal. All synthesized sequential elements except enabled shift registers can be optionally stalled.

All designs can incorporate one or more clocks and delay goals. The **clock** and **delay** attributes are used to set the current clock and delay goal. All synthesized sequential elements use the current clock, which is not included in the sequential function call. The delay goal is used to determine the insertion point of automatic pipelines and to determine slack during logic optimization.

Both state and pipeline registers are provided. State registers, such as an accumulator, are a required part of the architecture. Pipeline registers are used only to increase the circuit clock rate. Pipeline registers cause latency to increase, while state registers do not. In general, MC deskews latency variations caused by pipelining, so that multiple paths to the same point maintain the correct cycle alignment.

Sequential Functions

The most basic sequential functions are *preg*() and *sreg*(), which generate a shift register of pipeline register and state register, respectively. The shift register can have from 0 to *n* stages. *preg*() creates latency effects, while *sreg*() does not. *sreg*() is used to produce the registers required in the architecture. *preg()* is used to insert pipelines manually. Because of the latency deskewing effects, using *preg*() for the state registers of an FIR filter would have no effect but to delay the output, since the inputs to the multipliers would not be in different clock cycles.

Consider the examples below. In the first statement, the old A is added to the new A, creating a very simple filter. *sreg*() produces no latency, so no latency deskewing takes place. In the second case, the old A is added to the new A, but the old A has 1 cycle more latency than the new A, so latency deskewing will first delay the new A by one cycle, and will then add to the old A. This case is not very interesting. The last example shows how to create a 2-stage shift register without latency effects.

```
Z1=A+sreg(A);      Z1(n)=A(n)+A(n–1)
Z2=A+preg(A);      Z2(n)=2*A(n–1)
Z3=A+sreg(A,2);    Z3(n)=A(n)+A(n–2)
```

Three functions provide variable-length shift registers that can be used to match the latency at different parts of the design: *eqreg*(), *eqreg1*(), and *eqreg2*(). For example, there may be three outputs, each driven by logic that has been automatically pipelined, so the latency could be different at each of the three outputs. You can use *eqreg*() to force all outputs to have the latency of the most delayed output, regardless of the number of automatically inserted pipeline stages.

State Registers

The *sreg*() function is used to create a state register of fixed length. The *ensreg*() function is used to create a fixed-length state-shift register with an active **HIGH** enable control. When the enable is 1, the shift register is active and the data shifts with each clock rising edge. When the enable is 0, the shift register is inactive and no outputs change.

When access to the taps is required, for a register of length *n*, up to *n*+1 outputs can be passed to the function at the end of the parameter list. The first is connected to the input, the second is the output of the first taps, and the last is the output of the *n*th tap.

The *sreg*() function is affected by the **delstate** synthesis attribute. If **delstate** is greater than 0, pipeline loaning occurs. To disable pipeline loaning, set **delstate** back to 0. For a discussion of pipeline loaning, see “Pipeline Loaning” in Chapter 5.

Example 5-14 State Register Example

```
X=sreg(A,1);                 1 tap state register
directive(delstate=0);       have access to all taps, X_0 is the same
                             as input, X_4 is the final output
X=sreg(A,4,X_0,X_1,X_2,X_3,X_4);
X=ensreg(A,Y[3],4);          4-tap enable state reg
```

Manual Pipelining

For manual pipelining, the *preg*() function provides shift registers built from pipeline registers with a fixed length that is known in advance. This function requires one input, one output, and the integer register length to be passed to it.

When access to the shift register taps is required, for a register of length *n*, up to *n*+1 outputs are passed to the function at the end of the parameter list. The first is connected to the input, the second is the output of the first tap, and the last is the output of the *n*th tap.

Automatic Pipelining

The **pipeline** attribute enables and disables automatic pipelining. When automatic pipelining is enabled, MC inserts registers automatically (with a corresponding increase in latency) when the delay goal is exceeded. The pipelining is performed in a general and fine-grained (individual instance) level, so that any structure can be pipelined automatically. Automatic pipelines can fall inside of any structure and work in conjunction with manual pipelines that were generated by *preg*().

When **pipeline** is set to **on**, the **pipeslack** attribute adjusts the delay goal used during automatic pipelining. When the value of **pipeslack** is positive, the pipelines are placed closer together and the delay goal is reduced. When **pipeslack** is negative, pipelines are placed farther apart and the delay goal increases.

Matching Latency

The equalization functions, *eqreg*(), *eqreg1*() and *eqreg2*(), are used to build pipeline shift registers with a length determined during synthesis. *eqreg1*() is used when there is a desired latency for a signal. The function takes the input signal and the desired latency as inputs. *eqreg*() is used when the latency of a signal should match the maximum latency from a group of signals. *eqreg2*() is used when the latency of a signal should match the sum of the latencies of a group of signals. In each case, the function constructs a shift register with a length required to increase the input signal latency to the desired value. If the input latency exceeds the desired latency, an error is generated during synthesis.

Example 5-15 Latency Equalization

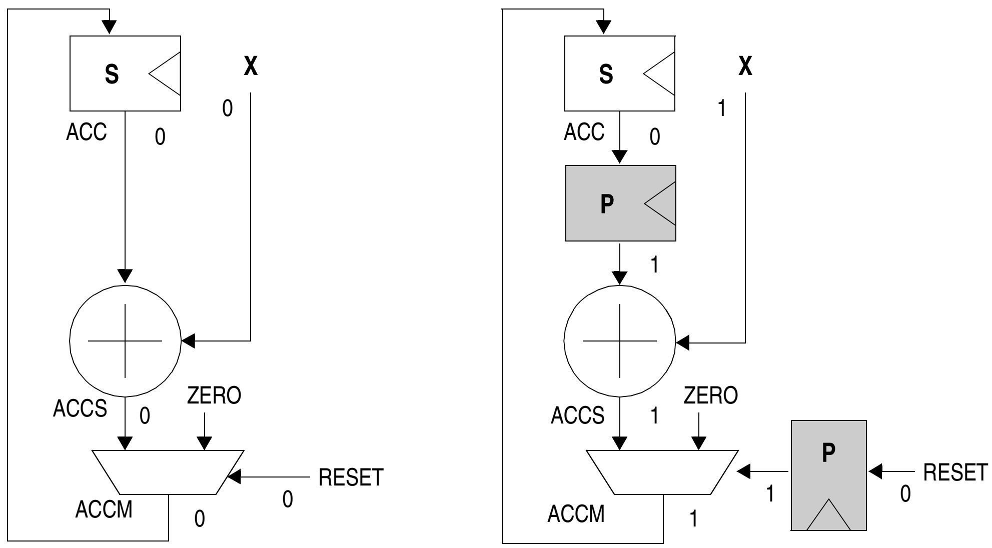
```
X=eqreg(A,3,B,C,D);           length is equal to max(lat(B),lat(C),lat(D))-lat(A)
X=eqreg1(A,3);                length is equal to 3-lat(A)
X=eqreg2(A,3,B,C,D);          length is equal to lat(B)+lat(C)+lat(D)- lat(A)

X=preg(A,2,B,C,D);            B is A delayed 0, C=A delayed 1, etc
```

Hiding Latency

You can generally allow MC to automatically deskew any latency differences created by pipelining. In some circumstances, it is essential to “hide” the latency to prevent automatic deskewing, especially when working with loops or pipeline loaning. In addition, because all inputs have 0 latency by default, you may want to treat the input as having whatever latency results in the least hardware. MC’s *hidelat*() function “removes” the latency of an operand and uses no hardware except for buffers that are normally removed during logic optimization. By default, the output of *hidelat*() is a signal with 0 latency. You can optionally provide a set of operands to *hidelat*() to force the latency of the output to track the minimum latency of the reference operands.

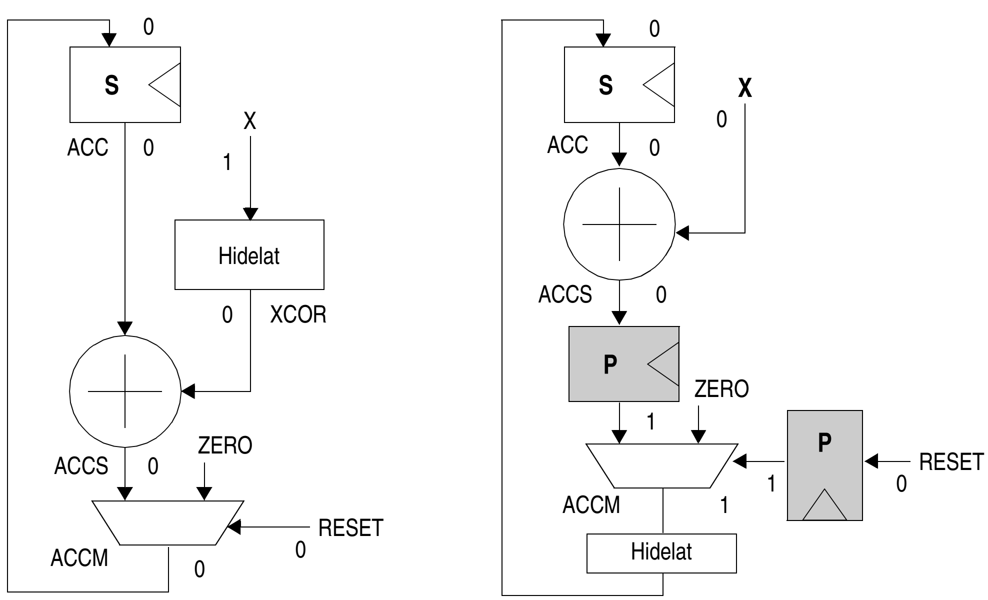
The following examples show the problems that occur when a signal with latency enters a loop. On the left is a simple case in which signal X has a latency of 0 that is being accumulated. Because the accumulator is a state register, there is no latency at its output. All signals inside the loop have a latency of 0 and no pipeline deskewing takes place. On the right, X now has a latency of 1. Pipeline deskewing occurs when X enters the loop, resulting in pipeline registers at the output of the accumulator and in the **RESET** signal path. Notice that the latency at the input to the accumulator is now 1 cycle, causing an error, as well as the unwanted registers.

Figure 5-1 Latency Deskewing



The problem above can be avoided by using the *hidelat*() function in the *X* signal path as shown below left. It causes the latency for XCOR to be 0 and hence no pipeline deskewing takes place. The case below right shows how pipelining within the loop can be accommodated, as long as the *hidelat*() function is used before the latency gets back to the feedback register input. Note that, for this case, there are two interleaved accumulators.

Figure 5-2 Hiding Latency



Hiding latency operates like a negative shift register by removing latency. Does this make any sense? How can you remove latency? Well, you aren't actually removing latency; you are hiding it to prevent deskewing. It is like converting the existing pipeline registers before the *hidelat*() function into state registers.

Example 5-16 Hiding Latency

```
module acc(Z,X,RESET);

     input signed [7:0] X;
     output [7:0] Z;
     input [0:0] RESET;
     wire signed [7:0] ACC,X1,XPR,ZA,RZA;

     XPR=preg(X,2);          used to create a latency problem
     X1=hidelat(XPR);        need to hide latency of XPR before entering the loop, reference to zero

     ACC=sreg(RZA,1);
     directive(muxtype="andor");
     ZA=X1+ACC;
     RZA=RESET ? ZA : 'h x;
     Z=ACC;

endmodule
```

This function requires one input representing the input operand. The number of reference inputs supplied can be specified. Any additional parameters are optional and represent the names of the reference operands. MC attempts to force the latency of the output of *hidelat*() to the minimum of the latencies of the reference inputs or to 0 if no reference inputs are supplied.

Example 5-17 Another Example of Latency Equalization

```
input [7:0] X,C0,C1;
wire [7:0] XD,C0COR,C1COR,XD_0,XD_1,XD_2;
wire [15:0] Z;
directive (delstate=2);
XD=sreg (X,2,XD_0,XD_1,XD_2);
C0COR=hidelat(C0,1,XD_0);        set latency to match XD_0
C1COR=hidelat(C1,1,XD_1);        set latency to match XD_1
Z=XD_0*C0COR+XD_1*C1COR;
```

Stalling and Scan Test

All synthesized flip-flops, whether state or pipeline registers, can be stalled by setting the **pipestall** attribute to the name of the stall control signal. By default, the pipeline is not stalled. The pipeline stalls when the stall control signal is **low**.

The **scan** attribute controls the conversion of flip-flops into their scan counterparts. When **scan** is **on**, the conversion takes place, and MC builds the circuit with a good area and delay estimates. When **scan** is set to **off**, no conversion takes place. During report generation, the scan FFs are converted back to the original cells.

Demultiplexing

Demultiplexing is the process or converting a high-speed serial data stream into *n* lower-rate parallel data streams. As the name implies, this process is the inverse of multiplexing which serializes a number of parallel streams.

Demultiplexors are implemented using a function called *demux*() that takes two signal inputs and a list of *n* outputs. The inputs are the data input and the select input. The data input is demultiplexed and the select input controls the demultiplexor. The integer input parameter specifies the demultiplexing ratio and the number of outputs. By default, the formats and widths of these outputs match that of the data input.

For proper operation, the select input must cycle through values of 0 to n–1, for each positive edge of *CLK*. The outputs change when *CLK* goes **high** and select has a value of 0.1 The input values that arrive when the select input has a value of 0, 1, 2, ... *n*–1 appear on the 0, 1, 2, ... *n*–1 indexed output, respectively. Below is an example for *n*=4.

Example 5-18 A Demux Example

```
input:  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
select: 0 1 2 3 0 1 2 3 0 1  2  3  0  1  2  3
out0:   x x x x x 0 0 0 0 4  4  4  4  8  8  8
out1:   x x x x x 1 1 1 1 5  5  5  5  9  9  9
out2:   x x x x x 2 2 2 2 6  6  6  6 10 10 10
out3:   x x x x x 3 3 3 3 7  7  7  7 11 11 11
```

1 to 3 demux:

```
wire signed [7:0] A, D0, D1, D2;
wire [1:0] S;
demux(A,S,D0, D1, D2);
```

1 to 2 demux:

```
wire signed [7:0] A, B,C;
wire [0:0] S;
demux(A,S,B,C);
```

The registers associated with demultiplexing are treated as state registers and hence no latency increase occurs in the demultiplexor. It is not possible to assign a latency to this structure because the delay between the input and the earliest output change varies from 2 to n+1 cycles.

A circuit with very conservative timing is used to implement the demultiplexor. The demux is built in two stages of enabled flip-flops. The first stage latches a value from the incoming data stream in the cycle in which the select input has the proper value. The second stage latches the outputs of the first stage when the select input has a value of 0. This approach might sacrifice area, but does guarantee that the critical path information will be correct.

Pipeline Loaning

The pipeline loaning option is fairly difficult to understand but simple to use. It is based on the concept that certain structures, primarily digital filters of various types, *require* the input data to be delayed. The direct implementation utilizes a state shift register at the input to generate the delayed versions of the input. The input and the outputs of the shift registers are then fed into a combinatorial function to compute the result. It should be obvious that, for symmetric functions, the critical path starts at the input to the shift register. By putting all the registers at the input, the registers are “wasted” in the sense that they are not being used to break up or to isolate the critical paths.

The concept of pipeline loaning is to convert some of the state registers into pipelines that can be used later to improve performance without increasing latency. The first n taps of the shift register are removed. (Actually, they are replaced by buffers that are later removed by the logic optimizer.) This has the net effect of progressively *decreasing* the latency at each point where a register was replaced, making it possible to effectively have negative latencies. These signals with reduced latency can now be pipelined without increasing the original latency. If all of the registers are removed from the input, the transposed form seen in many DSP books results. The transposed architecture suffers, in general, from an excessive use of flip-flops to improve performance. Pipeline loaning allows the architecture to move smoothly between the direct form and the transposed form without the need for changing the network description (except for the parameter *n*). In addition, because a small value of *n* generally provides most of the benefit, pipeline loaning results in areas close to that of the direct form and performance close to that of the transposed form.

To use pipeline loaning, you should write the network description to reflect the direct form. Set the number of stages to loan for pipelining as a parameter. A reasonable delay goal needs to be set (for example, don't optimize for speed) even if pipelining is not enabled (you don't need to enable pipelining), because the use of pipeline loaning allows, and in fact requires, pipelining for proper operation. The current delay goal is used to determine where the loaned pipelines should be placed. After a result has been formed that includes all the shift register outputs and its input, the delay goal can be set to any value; the pipeline loaning has completed. If the delay goal is set too low, the pipelines are quickly used up, providing little or no benefit.

Follow these steps to determine the parameter, *n*:

1. Start with $n = 0$ and a realistic delay goal
2. Is delay goal met?
3. If yes, quit, else increment *n*
4. Did performance improve?
5. If no, go back to previous *n* and quit, else continue
6. Is $n <=$ len?
7. If yes go to 2, else go back to previous *n* and quit

If the outputs of the shift register are the only operands that are connected to the combinatorial function, such as a fixed coefficient filter or correlator, this technique works transparently.

You should be careful to consider a couple of issues, particularly when other operands enter the function along with the shift register outputs. One case where this situation occurs is the variable coefficient FIR filter.

Assume that the coefficients and the shift register input have a latency of zero. The outputs of the shift register now appear to have a negative latency. When the coefficients merge with these negative latency signals at the multiplier, the shift register outputs are delayed to bring them back to latency 0, undoing the entire pipeline loaning. The latency of the coefficients needs to be "hidden" to avoid the latency deskewing.

In addition, the latency from the coefficients to the outputs is now different. The latency for the coefficient corresponding to the *i*th tap of the shift register is

i if $i <= n$

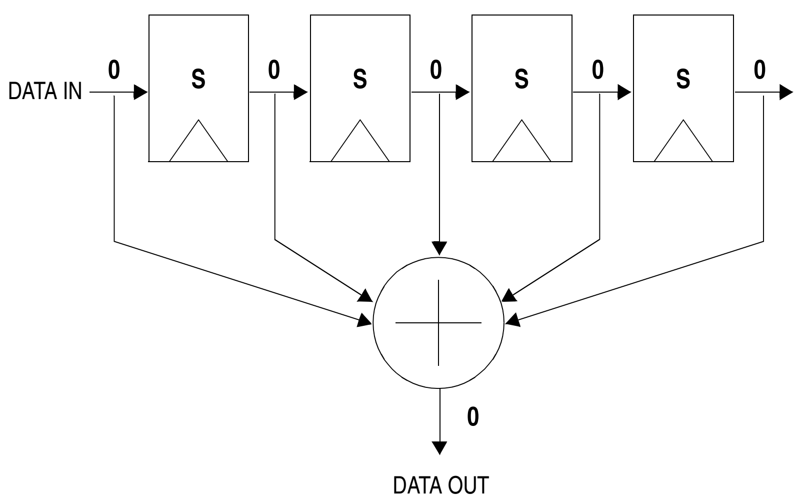
n if $i > n$

This change in latency can affect the performance of the algorithm and should be investigated.

Another problem can occur if the signals with decreased latency are not merged with a signal with normal (unadjusted) latency, because the initially requested latency has been removed. For example, an output of the shift register with pipeline loaning can be connected directly to a module output. MC checks all outputs to see if any “loaned” latency exists and corrects this situation automatically.

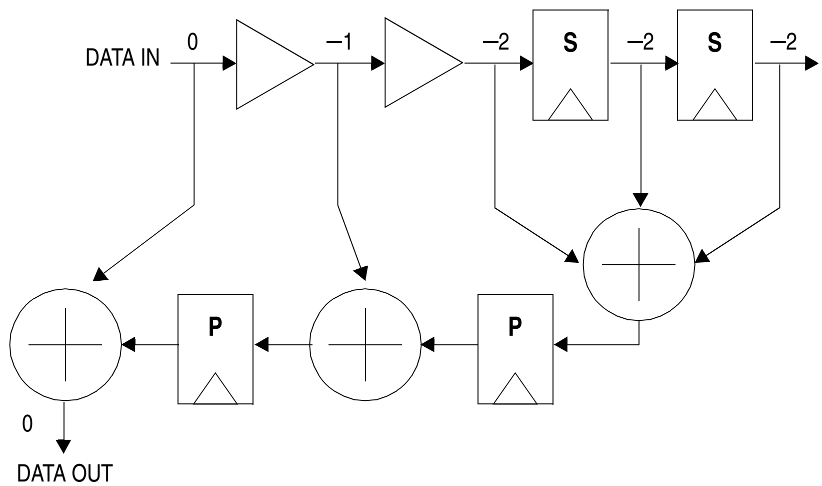
These points are illustrated in the diagrams below. The direct form implementation of a simple circuit that counts the number of ones in data stream is shown first, using all state registers (“S”). All signals have a latency of 0. The problem is that the critical signal, DATA IN, is not isolated from the less critical shift register outputs.

Figure 5-3 Direct Form FIR Filter



The pipeline loaning solution is shown below for two loaned stages. Note that the first two shift register taps have been removed and the latencies have been adjusted. Now, the least critical signals (with latency –2) are summed first. A pipeline register is automatically inserted when either the delay exceeds the current delay goal or the signal with latency –1 is encountered. Notice that two pipelines were inserted to replace the state registers that were removed. The latency of the output, DATA OUT, is zero in both cases. However, if the adder inputs were modified by another signal such as a coefficient, the latency from the coefficients to the output is now greater for some taps because of the inserted pipeline registers.

Figure 5-4 FIR Filter with Pipeline Loaning



If the **delstate** attribute were set to 4, we would have the transposed form implementation, with no registers in the input.

Signal Manipulation Functions

The MC Library provides several functions for manipulating signals. These functions do not perform any actual arithmetic or logical operation. Rather, they can be used to manipulate signal attributes such as size, timing, format, and so on.

Load Isolation and Buffering

Although the function synthesis routines automatically create buffer trees within each function to prevent overloading, your network description may contain large fanouts that cause overloading. If the overloading is severe enough, a rule violation is created that is corrected during optimization. During synthesis, however, the delay estimates of the overloaded nets will be inaccurate, potentially causing pipelining problems. MC provides two functions that help alleviate overloading: *buffer*() and *isolate*().

isolate

The *isolate*() function is provided to isolate heavy loads from the critical paths. It simply inserts a set of non inverting buffers between the input and output. The less critical paths should be driven from the output and the more critical paths from the input. The logic optimizer removes buffers that are not needed either because the circuit contains sufficient slack or because you made an incorrect assessment of which operand was more critical.

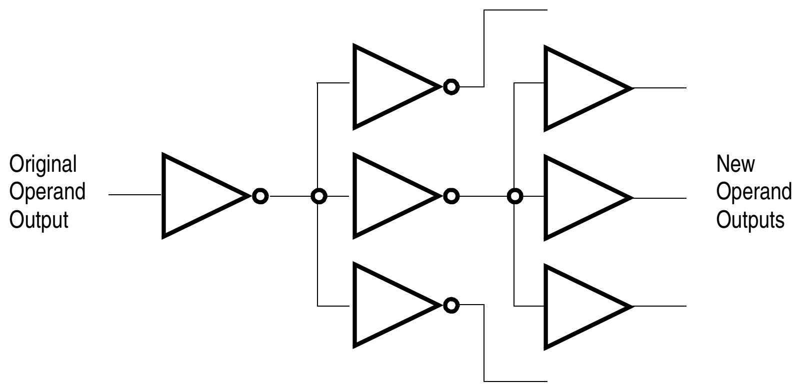
buffer

The *buffer*() function causes a buffer tree to be built with the depth specified (default 1). The maximum buffer depth supported is 5, which should be more than sufficient. Unlike the *isolate*() function, the *buffer*() function is used in situations requiring a symmetric buffer tree. There is no way to connect some paths to a part of the buffer tree closer to the root. However, you can always buffer the output of *isolate*().

```
input [7:0] A;
wire [7:0] ANC;          must match A!
ANC=isolate(A);          ANC has buffer depth 2
buffer (ANC,2);          build buffer tree at output of ANC
```

The instances produced by the buffer function are affected by the attributes in effect when the signal being buffered is defined. They are not affected by the attributes in effect when the buffer statement is encountered.

The buffer tree is built with inverters, except for the last stage, where the depth is odd and therefore uses noninverter buffers. In general, the logic optimizer removes and/or merges parts of the buffer tree whenever possible to improve circuit performance and area. A portion of a buffer tree of depth 3 is shown below. Note that only one stage is noninverting.



Signal Concatenation: cat() or ()

There are situations when it is necessary or convenient to create operands that are a concatenation of existing operands. The *cat*() function performs signal concatenation.

The *cat*() function takes a list of signals separated by commas. The bits are copied from the input signals, in order, to the output. The MSB of the left most input becomes the MSB of the result, while the LSB of the right most input becomes the LSB of the result. By default, the width of the result of concatenation is the sum of the widths of the inputs and the format is the same as the left most input.

MC allows a shorthand notation for *cat*() when there are two or more inputs: the name “cat” can be dropped, leaving only the parenthesis surrounding a comma-separated list of input signals.

Example 5-19 Signal Concatenation

```
input [7:0] A,B,C;
wire X;
X=cat(C[1:0],A[3:0],B[6:5]);          X=C[1],C[0],A[3],A[2],A[1],A[0],B[6],B[5]

input signed A;
input [1:0] B;
wire [7:0] X;                         shorthand style, no cat
X=(A,B)+C;
```

Tristates: join()

Although tristate drivers should be avoided when possible in ASIC designs, MC provides limited support for these constructs. The *join*() function can be used to connect two or more wires in a bit-wise fashion. A warning is generated if any of the drivers of the net are not tristate drivers. A module input cannot be an input to *join*().

In the example show below, the outputs of two 2-input tristate multiplexors are joined to form a 4-input MUX.

Example 5-20 Tristate Example

```
module mux1 (A,B,C,D,S,Z);
input [7:0] A,B,C,D;
input [1:0] S;
output [7:0] Z;
directive (muxtype="tristate");
wire [7:0] A1=S ? 'hx : 'hx : B : A;
wire [7:0] A2=S ? D : C : 'hx : 'hx;
wire [7:0] E=join(A1,A2);
Z=E;
endmodule
```

The Generic Cell Library

The MC generic cell library is a collection of low-level functions. Each function in the MC generic cell library is linked to a corresponding technology-specific cell if one exists, regardless of the cell and pin names used by the vendor. If there is no corresponding cell in the technology library, the generic cell function is synthesized from two or more technology-specific cells.

When the technology library contains several equivalent cells, MC chooses the best one during optimization, unless optimization has been disabled. MC always attempts to use the fastest cell during synthesis. The *Module Compiler Reference Manual* contains a complete list of functions in the MC generic cell library.

Note: If you need to describe a structure at gate level, you should use MC's generic cell library functions rather than the library provided by your vendor. This helps maintain technology and vendor portability.

Most generic functions accept bused inputs and outputs. Only those that already have bused inputs or outputs or those with more than one output cannot be bused. When a function is called with bused arguments, MC automatically generates an array of instances, one instance for each output bit. If any input is narrower than the output, the input is sign- or zero-extended in the same manner as other function calls.

The example below shows the construction of a technology-independent ripple adder, using the generic function *fa1a*(). Note how the *and2a*() function creates an array of 2-input **AND** gates. The *R* input is sign extended to the width of the *X* bus.

Example 5-21 Writing Standard Code

```
function adder (Z,X,Y);
     input X,Y;
     integer w=width(X);
     output [w:0] Z;
     wire [0:0] repl(i,w,",") {S{i},C{i}},C{w};
     C0=0;
     repl (i,w) {fa1a (S{i},C{i+1},X[i],Y[i],C{i});}        ripple adder
     Z=(C{w},repl(i,w,",") {S{w-i-1}});
endfunction

module foo (X,Y,Z,R);
     input signed [0:0] R;
     input [7:0] X,Y;
     output [8:0] Z;
     wire [7:0] XR;

     XR=and2a(X,R);                                           array of 2-input ORs
     Z=adder(Y,XR);
endmodule
```

Inserting Cells into the Design

Technology libraries provide a collection of technology-specific cells. “Foreign” cells are cells not represented in the technology library. Netlists are interconnections of technology-specific cells. Although MC has a rich library of synthesizable functions and generic cell library elements, there are occasions when you need to instantiate a cell from the technology-specific library or a netlist of library elements into the design. It is also possible, with some additional work, to insert a cell that is not in the installed technology library into your design.

For all cases, an MC language function is defined for the cell or netlist being inserted into the design. The interface to the function is in the library browser of the GUI. Netlists and cells not in the technology library are located in the “misc” category, while technology library cells are in one of the other categories. The cell or netlist is inserted by calling the appropriate function. Outputs of the function come before inputs and inouts in the parameter list.

There are two reasons for not overusing these options. First, these functions, unlike synthesized functions, are technology dependent. Moving your design to another technology could require changes to the MC language code or the netlist. Second, some advantages provided during synthesis will not be available, including automatic pipelining, latency deskewing, and multiple architectures for structures such as adders, multipliers, and multiplexors.

Technology-Specific Cells

It is simple to insert a cell from a technology library into your design. MC defines a function for each technology library cell. The functions for cells with one output and no buses accept bused inputs and outputs. MC automatically generates an array of instances for these cells, one instance for each output bit. If any input is narrower than the output, the input is sign extended in the same manner as other function calls.

The example below, which computes Z(*n*)=Y(*n*)+X(*n*–1), illustrates many of these points. The adder is implemented as a ripple adder using instances of the fa1a1 cell. An array of fd1a1 registers is generated to form *XR*. *CLK* is connected to the clock input of each fd1a1 because *CLK* is signed and thus is signed extended, while each fd1a1 receives a different data input. The output, Z, is driven by an array of OB4-type output buffers.

Example 5-22 Using Instances by Name

```
function adder (Z,X,Y);
     input X,Y;
     integer w=width(X);
     output [w:0] Z;
     wire [0:0] repl(i,w,",") {S{i},C{i}},C{w};
     C0=0;
     repl (i,w) {fa1a1 (S{i},C{i+1},X[i],Y[i],C{i});}        ripple adder
     Z=(C{w},repl(i,w,",") {S{w-i-1}});
endfunction

module pipe (X,Y,Z);
     input [7:0] X,Y;
     output [8:0] Z;
     wire [7:0] XR;

     XR=fd1a1(X,CLK);                                           array of FFs
     wire [8:0] Z1=adder(Y,XR);
     Z=OB4(Z1);                                                 output buffer instances
endmodule
```

The method used above is clearly not the best way to implement Z(*n*)=Y(*n*)+X(*n*–1). Using instances of library cells requires more lines of code and does not benefit from the multiple adder architectures and synthesis optimization available from MC. Also, the code in the example above is not guaranteed to work in every technology.

When inserting one of several equivalent technology specific cells directly into the design, you should insert the fastest (and likely the largest) equivalent cell. This choice has two effects: first the delays computed during synthesis are less sensitive to estimated loading inaccuracies, and second, the optimizer is more likely to find a good solution because the optimizer is better at reducing area than at improving performance.

Using Groups in Complex Designs

Some designs must be divided into sections sharing one or more common attributes or constraints. The MC directive can be used to set the **clock**, **delay**, **group**, **acswitch**, **dcduty**, **pipeline, dcopt** and **logopt** attributes for sections of the design. These attributes control the timing, power calculation, naming and optimization of the groups. When the value of one of these attributes is set in a directive, the value is in effect until another value is provided for the attribute.

Group Names

The **group** attribute is used to define a group and provide it with a name. There are three primary reasons to form a new group in the design. First, each group must have a single delay goal. If the delay goal is changed, a new group must be created. Second, it is convenient in large designs to break the design up into smaller groups for statistical and debugging purposes. Each group has a complete set of statistics (area, power, delay, etc.) and a critical path. A proper use of groups makes the job of determining the critical (delay, area or power) portion of the design much easier. Third, the groups can be used to assist in placement. By using the long instance name option, each instance name will have the group name as a suffix to allow grouping in the floorplanner or place and route system. MC allows you to use hierarchical group names as described later.

Group Timing and Pipelining

The **delay** and **pipeline** attributes are used to control the timing of a section of the design.

delay

The value of **delay** affects the synthesis of some structures and the optimization of all instances within the group. The value of the attribute is the current path delay goal and has units of ps. To prevent over constrained circuits, the **delay** attribute only affects the drivers of the path end points (either flip-flop inputs or module outputs) and not at the end points themselves. Therefore it is important to ensure that the drivers of all end points have the correct delay goal. In addition, because each group (as defined by the **group** attribute) must not have multiple delay goals, the **delay** attribute should be used in conjunction with the **group** attribute. Note that the command line option **-o** and the Optimization field in the GUI provide an initial value of the delay attribute.

pipeline

The **pipeline** attribute has a Boolean value indicating whether automatic pipelining is enabled within a section of the design. When pipelining is enabled (**on**), MC inserts pipelines when the delay exceeds the current value of the delay goal. When pipelining is disabled (**off**), automatic register insertion is not employed even if the delay exceeds the delay goal. (One exception to this is the case of pipeline loaning, which occurs transparently.) Pipelining sections can be smaller or larger than the groups defined with the **group** attribute. You can set he initial value of the **pipeline** attribute using the command line option or the GUI.

Multiple Clocks

MC supports multiple clocks in a design, although each group can have only one clock. All clocks are global signals that can be referenced throughout all levels of the design. The current clock can be declared by setting the **clock** attribute to the name of the current clock. Clocks should not be declared explicitly as wires.

All sequential circuits without explicit clock connections use the current clock. For example, *sreg*() and *preg*() have no clock argument and always use the current clock. Automatic pipelines are also connected to the current clock. You can also explicitly use any clock where ever a signal is required. For example, the RWN input of *ram1*() can be connected to a gated version of a clock**.**

The example below is a contrived circuit containing two clocks, *CLK* and *CLK1*. The registers in A1, A2, and A4 are connected to *CLK* while those in A3 are connected to *CLK1*. There are two I/O buffers used to drive the clock nets which have explicit connections to each of the clocks. Note that whenever the current clock is changed, the group must also change.

Example 5-23 Using Multiple Clocks

```
module clk (A,Z);
directive (logopt="off");
wire [0:0] clkout1,clkout2,PO1,PO2;
IBTU (PO1,clkout1,CLK,0);
clockbuf(clkout1);
directive (clock="CLK1",group="G1");
IBTU (PO2,clkout2,CLK1,0);
clockbuf(clkout2);
directive (logopt="on");
input [7:0] A;
output [7:0] Z;
directive (clock="CLK",group="G2");
wire A1=sreg(A);
wire A2=sreg(A1);
directive (clock="CLK1",group="G3");
wire A3=sreg(A2);
directive (clock="CLK",group="G2");
wire [7:0] A5=A+A3;
wire A4=sreg(A5);
Z=A4;
endmodule
```

Disabling Module Compiler Logic Optimization

When minimizing delay in a section of a design is not desired, the logic optimizer of MC can be disabled with the **logopt** attribute. This attribute is a Boolean that enables logic optimization when set to **on**. When it is set to **off**, all logic optimization is disabled, including fixing rule violations. Logic optimization should be disabled only on rare occasions and is **on** by default.

Disabling Design Compiler Optimization

Sections of MC code can be selectively optimized by Design Compiler. Generally, arithmetic logic benefits the least from optimization by DC, while AND-OR logic benefits the most. You can choose to optimize all, some, or none of your circuit by setting the **dcopt** attribute. This attribute is a Boolean that enables DC optimization when set to **on** and disables optimization when set to **off**. The entire circuit is sent to Design Compiler, but DC does not touch any instance that was created when **dcopt** was **off**. DC optimization is **off** by default.

Changing the Power Computations

MC employs a simple static power model in which you provide an AC switching factor and a DC duty cycle value for a section of code (not necessarily a group as defined by the **group** attribute). These values are used to compute the power for all instances within the section. You may want to adjust these values when it is known that the instances in the section have a different constraint than those of another section. For example, the logic might be clocked at a different frequency or it might be known to have a smaller or larger toggle rate because of data constraints or constraints related to the particular function being implemented.

The **acswitch** and **dcduty** attributes specify the AC switching factor and the DC duty cycle, respectively. Both are expressed as integer values representing percentage values. A value of 0 for both indicates no contribution to power. A value of 100 for **dcduty** indicates the instances (typically high density RAMs) consuming DC power are enabled and consuming DC power 100% of the time. A value of 100 for **acswitch** indicates that 100% of the nets are switching at 1/2 the global clock frequency. A typical value for **acswitch** is 20% indicating that the nets switch in 1 out of 5 possible transitions.

In the example below, note that the various attributes affect different and overlapping sections of code. Only the **clock** and **delay** attribute changes need to be aligned with **group** attribute changes.

Example 5-24 Computing Power

```
directive (group="old",delay=3000);        set the delay goal to 3 ns
directive (pipeline="off");                don't allow auto pipelining
wire [7:0] A,B,C;
wire [15:0] D,E,F;
directive (acswitch=10);                   D is switching at 10%
D=A*B;                                     can't pipeline D
directive (pipeline="on");
directive (acswitch=15);                   E and F are switching at 15%
E=D+C*C;                                   can pipeline E
directive (group="new",delay=5000);        set the delay goal to 5, new group
directive (logopt="off");                  don't optimize F
F=E+A;                                     pipelining is still on, can pipeline F
```

Multiple Delay Goals

Consider a design with multiple clocks in which all clocks can be formed by dividing a master clock by an integer. Although it is possible to use multiple clocks for this type of problem, the enable registers provide a simple mechanism for implementing multiple clocks. There is still a single master clock, but now there are many local enables which generate local clocks (within the flip-flops) of different frequencies.

The circuit should be divided into groups, such that all the logic within a group is operating at the same frequency. In addition, frequency changes of the data must occur at registers or primary inputs, and the groups must contain the registers or inputs that are referenced within the group. The delay goal, group name and **acswitch** synthesis attributes are then changed immediately before the group is described.

As an example, consider a circuit utilizing two clocks, one that is half the frequency of the other.

Example 5-25 Clock Groups

```
function clkgrp(name, delay, switch);
     string name;
     integer delay,switch;
     directive global
(group=name,delay=delay,acswitch=switch);
endfunction

module test (A,B,Z,RESET );
integer del=10000;                master clk cycle time
integer sw=20;                    master clk ac switching

clkgrp (fastA,del,sw);
     input [7:0] A,B;
     input [0:0] RESET;
     wire [7:0] Y;
     output [7:0] Z;
     Y=A+B;
     wire [0:0] EN,ENN;
     EN=sreg(ENN,1);              generate the enable circuit
     ENN=~EN&RESET;

clkgrp (slow,del*2,sw/2);
     wire [15:0] S1,S1A;
     wire [7:0] S2;
     S1=ensreg(S1A,EN,1);
     S2=ensreg(Y,EN,1);
     S1A=S1*S2;

clkgrp (fastB,del,sw);
     wire [15:0] S3;
     S3=sreg(S1);
     Z=S3[15:8]^S3[7:0]+A;
endmodule
```

In the simple example shown, there are three groups in the circuit. Two (fastA and fastB) are operating at 10.0 ns and the other (slow) is operating at 20.0 ns to allow more time to process S1*S2, which is only needed every other cycle. Note that there is a function *clkgrp*() that changes the delay, group name and ac switching factor together. Also note that with this setup, the inputs to the S1 and S2 registers are paths with 10.0 ns delay goal because the drivers of the end of path are in a group with 10.0 ns delay. The outputs of the registers and the multiplier have a 20 ns delay goal, however. At S3, the input path has a 20 ns delay goal, while the output has a 10 ns delay goal. The output, Z, changes every 10 ns and will be optimized with a 10 ns delay goal.

When a design has multiple delay goals, the most critical path is not always the longest path. This should be clear if you consider a path in slow with a delay of 19 ns and one in fastB with a delay of 11 ns. The path in slow has 1 ns of slack while that in fastB has –1 ns of slack. Therefore, the critical path reported will be that in fastB, rather than the one in slow. This is why it is important to look at slack rather than delay when using multiple delay goals.

Report Control

You can request additional group and critical path information be printed in the report file, by inserting functions in the MC language input file. Because these functions are placed in the input file, the design must be resynthesized each time the reporting functions are changed or added.

Groups in MC can be either hierarchical or flat (disjoint). Flat or disjoint groups are created by choosing group names without “.”. Each of these groups represents a non-overlapping portion of the design. Hierarchical groups are created by inserting “.” in the group name. Each portion of the group name following a “.” represents a division of the group with the name preceding the “.”. For example A.1 and A.2 are two disjoint divisions of the group A. A.1.1 and A.1.goo are two disjoint divisions of the group A.1.

Group Analysis

By default, MC provides two reports for the groups in the design. First, it generates the list of all top level groups. In this list, the groups A, A.1, A.2 and A.1.1 are combined to form the group A. If you have not used hierarchical group names, this list contains all groups in the design. Second, it generates the list of all groups. In this list A, A.1, A.2 and A.1.1 are reported independently.

You can request additional group information by using the *showgroup*() function. Each *showgroup*() function accepts a group name pattern and causes one list of groups to be inserted into the report file. The list contains all groups with names matching the pattern. The pattern is supplied as a dot separated list of names. MC locates all groups matching the names supplied in the pattern. You can use * to match any name at any level of the hierarchy.

All groups that match the pattern are merged, even if they have more levels of hierarchy than the pattern. For example, suppose we have a design with groups B.1, B.2, A, A.1, A.2 and A.1.1. The groups that are displayed and merged for several patterns are shown below.

Pattern	Group Name Displayed	Groups Merged
*	A	A, A.1, A.1.1, A.2
	B	B.1, B.2
A.*	A.1	A.1, A.1.1
	A.2	A.2
*.1	A.1	A.1, A.1.1
	B.1	B.1

When groups are merged, the area, power, number of flip-flops and the number of instances are summed. The latency is the maximum of the latencies in the sub-groups while the internal delay corresponds to the most critical path for all sub-groups.

The group information is available in the Design Report file and in the Stats option in the View menu.

Path Analysis

MC provides critical path analysis for the entire design and for each user-defined group. In addition, four MC language functions are provided to allow you to specify additional critical paths to analyze. These functions are summarized in the table below.

Table 5-6 Functions Used for Path Analysis

Function	Use
critpath (string start, string end, string name);	Find the critical path from start to end, use name
disablepath (string point);	Don’t allow paths to go through point
enablepath (string point);	Allow paths to go through point
critmode (string mode);	Set the reporting mode.

User-defined critical paths have two modes, **short** and **full**. The *critmode*() function sets the current reporting mode. Use **full** mode to display the full path, such as the critical paths shown for the design and groups. Use **short** mode to display only the name and the critical path length, providing a datasheet-like output. The mode affects all critical paths printed until the mode is changed by calling *critmode*() again. By default, the reporting mode is **full**.

It is possible to prevent critical paths from passing through internal operands by using the *disablepath*() function. This function takes one string argument that is the operand name or operand bit range that the critical path is not allowed to pass through. Using a value of * disables all internal paths. Input operands cannot be disabled with this command.

The *enablepath*() function does the opposite of the *disablepath*() function. It takes one string argument that is the operand name or operand bit range that the critical path is allowed to pass through. Using a value of * enables all internal paths.

The *critpath*() function takes three string values and finds the most critical path—the path with least slack at the endpoint—from **start** to **end**. The critical path does not go through internal operands that have been disabled and not subsequently enabled. The path is named with **name** and is listed in the User Critical Path section of the report file. The value of **start** can be an operand name, operand bit range, or * to start at any input. Both **end** and **start**, takes values of an operand name, bit range, or * to end at any output. Additionally, you can use ** to end at any output or other path end point such as a flip-flop D input). *CLK* can be used as an end name to enable paths that stop at register inputs.

The order of any *critpath*() function relative to the other three functions is very important, because *critmode*(), *enablepath*() and *disablepath*() determine how subsequent *critpath*() functions behave.

A Complete Example

Suppose you want to create a video front-end processor that uses an RGB-to-YUV converter. In addition, you need to process each output of the converter with an FIR filter. Since you want a *compiler* that can be called with different values to generate different video processors, rather than a static piece of code, you need to pass some parameters to the module. This compiler can be built as shown in the following example.

Example 5-26 A Complete Example

```
/* define some macros for use through out this exercise
*/
#define COEFFS1 replicate(integer i=0;i<taps;i=i+1){ C{i},}
#define COEFFS replicate(i=0;i<taps;i=i+1){ C{i},}
#define TAPS replicate(i=0;i<=taps;i=i+1){ TAP{i},}

/* build an FIR filter using the given coefficients.
*/
function fir1 (OUT,IN, taps, COEFFS1 wOut);
     integer taps;                          the number of taps in the filter
     input IN;                              this the data input, declared outside
     input COEFFS;                          taps number of C inputs
     integer wOut;
     output signed [wOut-1:0] OUT;          declare OUT with enough bits
     wire if (formatStr(IN)==signed)
      { signed } [width(IN)-1:0] OUT_DELIN,TAPS;

     OUT_DELIN=sreg(IN,taps,TAPS);          the state shift register

     /* compute the inner product */
     OUT=replicate(i=0;i<taps;i=i+1){+TAP{i+1}*C{i}};
endfunction

/* build a converter from RGB format to YUV format
*/
function RGBtoYUV (Y, U, V, R , G, B, width);
     integer width;                         width is the number of bits in the output
     input R,G,B;                function inputs are not declared, must be declared elsewhere
     output signed [width-1:0] U,V;         Y,U,V are created here
     ouput unsigned [width-1:0] Y;
     Y=87*R+G*37+B*15;
     U=-33*R+15*G-97*B;
     V=109*R-49*V+65*B;
endfunction

/* build the compiler: taps, wIn and wC control the size of the video
* processor
*/
module video(taps, COEFFS1 R,G,B,Y,U,V,wIn,wC);
     directive (pipeline="on",delay=9999999);

     integer taps;
     integer wIn;
     integer wC;
     input unsigned [wIn-1:0] R,G,B;
     input signed [wC-1:0] COEFFS;
     output Y,U,V;                          width of these determined by the fir
     wire Y1,U1,V1;

     RGBtoYUV(Y1,U1,V1,R,G,B,16);
     Y=fir1(Y1[15:6],taps,COEFFS 21);
     U=fir1(U1[15:6],taps,COEFFS 21);
     V=fir1(V1[15:6],taps,COEFFS 21);

endmodule
```

This example creates a compiler called “video” with three parameters that control the number of taps in the filters (“taps”), the width of the input data (“wIn”), and the width of the filter coefficients (“wC”). With each run of MC, you can specify a value for the top level parameters that is propagated through the hierarchy. This compiler uses functions to achieve hierarchy in the input description, but the synthesis and optimization processes are performed on the flattened description.

Optimizing Performance and Area

The MC language provides you with many tools for describing your circuit. To illustrate how much control you have over the result, a progression of examples is provided which take the design from a poor solution to a good solution without changing the functionality of the circuit.

The following example performs a color space conversion as shown in some of the previous examples. This is a rather simple operation, but important in video applications.

```
Y=77R+150G+29B
U=128R-107G-21B
V=-43R-85G+128B
```

Clearly, we need to perform nine multiplications and six additions or subtractions. We could pursue a somewhat naive design approach: construct a module with nine multiplications and the six adders/subtractors, and supply the coefficients later (outside of MC) as shown below. Note that we went to extra work to break the equation for Y into sub equations for Y1, Y2, Y3 and Y4. Now each of the these internal values is generated with a carry propagate adder.

Example 5-27 A Complete RGB-to-YUV Design

```
module RGB_var_fastcla_serial_nocs (Y, U, V, R , G,
B, C00, C01, C02, C10, C11, C12, C20, C21, C22);
     directive(fatype="fastcla",delay=1);
     input [7:0] R,G,B;
     input signed [7:0] C00, C01, C02, C10, C11, C12,
C20, C21, C22;
     wire signed [15:0] U1,U2,U3,U4;
     wire signed [15:0] V1,V2,V3,V4;
     wire unsigned [15:0] Y1,Y2,Y3,Y4;
     output signed [15:0] U,V;
     output unsigned [15:0] Y;
     Y1=C00*R; Y2=C01*G; Y3=C02*B; Y4=Y1+Y2; Y=Y4+Y3;
     U1=C10*R; U2=C11*G; U3=C12*B; U4=U1+U2; U=U4+U3;
     V1=C20*R; V2=C21*G;V3=C22*B; V4=V1+V2; V=V4+V3;
endmodule
```

After running the previous example, we obtain the following table file:

```
Module                        Sections Delay     Latency   Power
RGB_var_fastcla_serial_nocs   7143     17.26     0         1.068
```

It is hard to tell how well this case worked until we compare it to another implementation. For comparison, let's change the adder type from **fastcla** to **clsa** that is expected to perform better for skewed delay cases like multipliers. We simply change the directive statement to get the following input.

```
module RGB_var_clsa_serial_nocs (Y, U, V, R , G, B,
C00, C01, C02, C10, C11, C12, C20, C21, C22);
     directive(fatype="clsa",delay=1);
     input [7:0] R,G,B;
     input signed [7:0] C00, C01, C02, C10, C11, C12,
C20, C21, C22;
     wire signed [15:0] U1,U2,U3,U4;
     wire signed [15:0] V1,V2,V3,V4;
     wire unsigned [15:0] Y1,Y2,Y3,Y4;
     output signed [15:0] U,V;
     output unsigned [15:0] Y;
     Y1=C00*R; Y2=C01*G; Y3=C02*B; Y4=Y1+Y2; Y=Y4+Y3;
     U1=C10*R; U2=C11*G; U3=C12*B; U4=U1+U2; U=U4+U3;
     V1=C20*R; V2=C21*G; V3=C22*B; V4=V1+V2; V=V4+V3;
endmodule
```

Now we can rerun MC to get the following table file.

```
Module                        Sections      Delay       Latency     Power
RGB_var_fastcla_serial_nocs   7143          17.26       0           1.068
RGB_var_clsa_serial_nocs      6051          16.46       0           0.831
```

As expected, we made some progress in both area and delay due solely to the ability of the clsa adder to optimize its structure around the delay skews in the circuit.

Now it should be clear that we can make some significant improvements by merging the five equations for each color component output. For each output, we will have a single Wallace tree implementing three multiplications and two additions followed by a single carry propagate adder. We should have done this first, since the input is much simpler, as shown below.

```
module RGB_var_clsa_par_nocs (Y, U, V, R , G, B, C00,
C01, C02, C10, C11, C12, C20, C21, C22);
directive(fatype="clsa",delay=1);
     input [7:0] R,G,B;
     input signed [7:0] C00, C01, C02, C10, C11, C12,
C20, C21, C22;
     output signed [15:0] U,V;
     output unsigned [15:0] Y;
     Y=C00*R+C01*G+C02*B;
     U=C10*R+C11*G+C12*B;
     V=C20*R+C21*G+C22*B;
endmodule
```

After running this case, we have the following table file.

Module	Sections	Delay	Latency	Power
RGB_var_fastcla_serial_nocs	7143	17.26	0	1.068
RGB_var_clsa_serial_nocs	6051	16.46	0	0.831
RGB_var_clsa_par_nocs	4612	13.18	0	0.605

Clearly, reducing the number of carry propagate adders by merging the equations results in even greater savings of area and performance than simply changing the adder types. There is another method of achieving nearly identical results: using the **carrysave** directive attribute. In the following example we have not merged the equations, but instead have defined all the internal nodes to be carrysave and hence MC does not generate carry propagate adders at these nodes.

```
module RGB_var_clsa_serial_cs (Y, U, V, R , G, B,
C00, C01, C02, C10, C11, C12, C20, C21, C22);
     directive(fatype="clsa",delay=1);
     input [7:0] R,G,B;
     input signed [7:0] C00, C01, C02, C10, C11, C12,
C20, C21, C22;
     wire signed [15:0] U1,U2,U3,U4;
     wire signed [15:0] V1,V2,V3,V4;
     wire unsigned [15:0] Y1,Y2,Y3,Y4;
     output signed [15:0] U,V;
     output unsigned [15:0] Y;
     directive(carrysave="on");
     Y1=C00*R; Y2=C01*G; Y3=C02*B; Y4=Y1+Y2;
     U1=C10*R; U2=C11*G; U3=C12*B; U4=U1+U2;
     V1=C20*R; V2=C21*G; V3=C22*B; V4=V1+V2;
     directive(carrysave="off");
     Y=Y4+Y3;
     U=U4+U3;
     V=V4+V3;
endmodule
```

The table file after these four runs now becomes:

Module	Sections	Delay	Latency	Power
RGB_var_fastcla_serial_nocs	7143	17.26	0	1.068
RGB_var_clsa_serial_nocs	6051	16.46	0	0.831
RGB_var_clsa_par_nocs	4612	13.18	0	0.605
RGB_var_clsa_serial_cs	4725	13.30	0	0.609

The carrysave case provides only slight degradation of area and delay over the fully merged case. This example shows the power of using carrysave; area and delay are improved and access to internal nodes such as Y1, Y2, and Y3 is possible.

Finally, because the coefficients are already known, MC will optimize the circuit with these coefficients. Note that in the example below, we have a level of hierarchy through a function that looks like a variable coefficient matrix multiplier. However, in the module we call the function with the fixed coefficient values. MC automatically determines that the multiplications can be optimized.

```
function RGB (Y, U, V, R , G, B, C00, C01, C02, C10,
C11, C12, C20, C21, C22);
     input R,G,B;
     input C00, C01, C02, C10, C11, C12, C20, C21,
C22;
     output U,V;
     output Y;
     Y=C00*R+C01*G+C02*B;
     U=C10*R+C11*G+C12*B;
     V=C20*R+C21*G+C22*B;
endfunction

module RGB_fixed_clsa_par (Y, U, V, R , G, B);
     directive(delay=1,fatype="clsa");
     integer width;
     input [7:0] R,G,B;
     output signed [15:0] U,V;
     output unsigned [15:0] Y;
     RGB (Y,U,V,R,G,B,77, 150, 29, 128, -107, -21,
-43, -85, 128);
endmodule
```

Now when we run the example, we get the following table file.

```
Module                       Sections     Delay    Latency     Power
RGB_var_fastcla_serial_nocs  7143         17.26    0           1.068
RGB_var_clsa_serial_nocs     6051         16.46    0           0.831
RGB_var_clsa_par_nocs        4612         13.18    0           0.605
RGB_var_clsa_serial_cs       4725         13.30    0           0.609
RGB_fixed_clsa_par           1652         10.03    0           0.181
```

Obviously, the use of the fixed coefficients has provided enormous benefits. The area dropped by nearly 66% from the previous best case and delay decreased by nearly 25%. From our original case the gains are even greater.

It should be clear that we could take this case further by utilizing pipelining to achieve even high performance levels, but this is left as an exercise to the reader.

Technology Library Support

6

This chapter provides an overview of how to use third-party technology libraries with the Module Compiler product.

Chapter 6 discusses the following topics:

- Library functionality
- How MC models delay, capacitance, cell timing, wire load capacitance, wire load resistance, derating, and operating conditions.
- Required cell functionality
- Recommended cell functionality
- How to use the library report to determine the degree to which your technology library contains this functionality

Library Functionality

The technology library provided by your vendor supplies critical information to MC. This information includes:

- The functionality, timing and loading of all cells in the library
- The estimated wire load models
- The operating conditions
- The derating models

MC reads one or more industry standard Synopsys DB format files.

MC's computation algorithms and highly streamlined internal data structures for storing these models allow MC to run fast. Unfortunately, not all Synopsys DB models and objects map directly into MC data structures. In some cases, there may be small differences between the results obtained with MC and other Synopsys tools. Keep in mind that MC is a prelayout synthesis tool, in which wire load capacitances are not known. No tool can produce exact timing results under these conditions.

Delay, Capacitance, and Area Units

MC operates in technology-independent units to make the input constraints and output values relatively insensitive to vendor library variations. MC stores all values as integers to speed computation. Any floating point numbers in the vendor library must be scaled and converted to integers.

Table 6-1 Technology-Independent Units

Element	Unit
Timing constraints	integer ps
Delay values in reports	floating-point ns
Loading constraints	integer tenths of standard loads
Load values in reports	floating-point standard loads

A **standard load** is defined to be the smallest load of any pin in the library. This approach allows you to provide input constraints and read the output reports in a technology-independent manner even when the vendors have chosen vastly different units within each library.

CBA and Non-CBA Libraries

MC has two distinct ways of computing area: one for CBA (Cell-Based Array) libraries and another for all other types of libraries. The CBA architecture is a heterogeneous array containing compute and drive sections, making area computation more complex. Homogeneous architectures can represent area by a scalar value, while CBA libraries require a two-dimensional vector area measure. MC automatically detects CBA libraries that contain the true vector measure and optimizes the design to minimize the actual area. Other Synopsys tools currently do not recognize the vector area measure and use a different scalar area measure. The value from MC should be considered correct. When MC encounters a homogeneous architecture library, its area calculations should match those of other Synopsys tools, with the exception that MC rounds area to the nearest integer. Note that MC ignores the wire area in all area calculations.

Timing Models

The timing model provides a basis for calculating the delay through a cell. A variety of approaches have been used in the past, each with a different tradeoff between accuracy and computational complexity. Below is a brief summary of common timing models.

Table 6-2 Timing Models Supported by Module Compiler

Model	Description
Linear	Delay is perfectly linear with respect to all output capacitance values
Piecewise Linear	Delay is linear within each of several regions of output capacitance
Nonlinear	Delay is computed as a function of both output capacitance and input transition time

MC uses the nonlinear timing model for all delay calculations. This model uses the input transition time in addition to the output capacitance to determine the delay through a cell. There are two variations of this model: in one, the cell delay and transition time are provided; in the other, the propagation delay and the transition time are provided (cell delay = propagation delay + transition time).

Simple linear models and piecewise linear models are subsets of the nonlinear model; the transition time dimension is not used. Runtime performance of delay calculation improves when simpler models are employed. Reducing the number of breakpoints in either or both dimensions or using only one dimension both speed up MC.

All Synopsys timing models are mapped into the nonlinear timing model. With the exception of the edge-rate effects of the CMOS 2 model, there is very little error in the mapping. Because MC ignores the CMOS 2 edge-rate effects, MC results are somewhat optimistic relative to other Synopsys tools when this model is employed. The nonlinear timing model is quickly becoming the industry standard, so MC is well positioned to provide timing estimates that are as accurate as possible.

Setup and Holdtime Models

MC supports scalar setup times and ignores hold times entirely. Therefore, there is some inaccuracy for libraries containing transition-dependent setup times. Since there is only one setup time per path, this effect is not cumulative.

Wire load Models

The wire load model provides estimates of the load of the yet-unrouted nets in the design as a function of the number of pins (or fanouts) on the net. The loading is estimated based on statistical properties of the place and route tools and the size of the region in which the design is placed.

MC supports the Synopsys piecewise linear wire load model. You can select any wire load model used by MC at any time. However, the design has only one active wire load model at a time.

For convenience, MC defines several pure linear wire load models. These models can be used for comparing technology libraries that have inconsistent wire load models. The names of the predefined models and the number of loads per fanout is summarized in Table 6-3.

Table 6-3 Predefined Linear Wire Load Models

Model name	Standard Loads/fanout
synlinear0	0
synlinear1	1
synlinear2	2
synlinear2.5	2.5
synlinear3	3
synlinear5	5
synlinear10	10

The wire load model name is stored as a technology-dependent environment variable. When you change technologies, MC automatically remembers the wire load model you last used for that technology.

Derating Models

The derating model provides a method for computing the loading, delay, and resistance effects in the circuit as the process, temperature, and voltage are changed from those under which the library data was measured.

The derating model used in MC is linear for each variable. That is, the actual delay can be computed for any process, voltage, or temperature as follows.

$$t(P, V, T) = t(P_0, V_0, T_0) \cdot (1 + K_P(P - P_0)) \cdot (1 + K_V(V - V_0)) \cdot (1 + K_T(T - T_0))$$

where P_0, V_0, and T_0 are the process, voltage, and temperature under which the library data was measured. MC supports the independent derating of the rise and fall values of cell delay, transition delay, propagation delay, and setup time. Wire load and pin capacitance are derated using the same linear technique.

You do not select the values of P, V, and T directly. Rather, you select one of the named operating conditions (opconds). Each named opcond corresponds to a value of process, voltage and temperature. As with wire load names, the named opconds are stored as technology-dependent environment variables, so you can change technologies without having to reenter the appropriate opcond information.

MC automatically creates the opcond, **synlibcond**, which corresponds to the conditions under which the library data was measured.

The derating models for all timing models are mapped into this one derating model. The Synopsys linear and the piecewise linear timing models have derating parameters that do not directly correspond to those in the implemented model. For these cases, MC uses the average value of the derating parameters. The derating is accurate unless the derating parameters are not equal. This simplification is little cause for concern. It is unlikely that you will be using these simple timing models and even less likely that a vendor will provide unequal derating parameters for these simple models. Also, the inaccuracies caused by MC mapping the derating model are trivial compared to the inherent inaccuracies of these simple models.

Resistance Models

Currently MC ignores wire resistance. This is equivalent to using the “Best Case Tree” type.

Sequential Models

MC does not support the state table method of representing sequential elements. This method appears to be obsolete. If you need to use a library employing this method, please contact your MC applications support representative for a workaround.

Library Functionality

This section covers the required and recommended cell sets for use with MC. It is organized by the type of function being synthesized and gives the requirements and recommendations for each. Look in the Library Report to see if a given type of cell is available in the currently loaded technology library.

All cells, excluding the required basic cells, the basic D flip-flop, all latches, and the tristate buffers can be constructed as pseudo-cells if they are not available directly in the vendor’s library. MC has the ability to build all of the appropriate pseudo-cells. Of course, properly designed and implemented native cells provide advantages over the pseudo-cells in area, delay, power, and place and route complexity.

The cell names are the MC generic cell library names. The functionality of these cells can be found in the *Module Compiler Reference Manual*.

Basic Cells

The following cells are required for MC to run in any mode.

- inv1a
- buf1a or buf2a
- nand2a, and2a, or2a, and nor2a
- xnor2a and xor2a

The following cells are required for MC to run when not building the pseudo library. If these cells don't exist in the vendor's library, they must be constructed as pseudo-cells.

- mx2a

MUX-Based Multiplexors, Shifters, and Rotaters

MUX-based multiplexors, shifters, and rotators can be built with the required basic cells, but for best results, the following cell is highly recommended:

- mx2d

In addition, the following cells are recommended to build the most efficient multiplexors:

- mx3a
- mx4a

Tristate-Based Multiplexors

The following cell is required to synthesize tristate MUXs:

- tri1a

Flip-Flops

The following cells are required to synthesize sequential elements:

- fd1a (for *sreg*(), *preg*() and autopipelining without stall)
- fde1a (for ensreg and stall modes)

The following inverted versions are recommended to minimize inverters:

- fd1c
- fde1c

To fully support **scan test mode**, the following must be provided:

- fdm1a (when fd1a is needed)
- fdem1a (when fde1a is needed)

For most efficient results, the following should also be provided:

- fdm1c
- fdem1c

MC can use cells with both true and inverted outputs to replace fd1a, fde1a, fdm1a, and fdem1a.

Latches

The following cell is required to synthesize latches and netlist memories. Equivalent cells with multiple outputs can also be used.

- ld1a or ld1b

For most efficient results, the following cell should also be provided:

- ld1c

AND-OR Trees

MC can use the following cells when it is building trees based on AND and OR functions; these cells are not required.

- and3a-and8a
- nand3a-nand8a
- nor3a-nor8a
- or3a-or8a

XOR trees

The following cells are highly recommended for building XOR trees:

- xor3a
- xnor3a

Adder Cells

The following cell is required to build any adder structures:

- fa1a

The following cell is highly recommended for building any adder structures:

- ha1a

In addition, the following cells are required to build optimized RIPPLE adder types:

- fa2a
- fa1b

In addition, the following cells are required to build CSA and CLSA adder types:

- facs2a (2 architectures recommended)
- facs1b (2 architectures recommended)
- facs3a
- facs4a
- mx2d

In addition, the following cells are recommended to get efficient incrementors and comparators when for CSA and CLSA:

- hacs2a
- hacs1b
- ha2a
- ha1b
- facs2a
- faccs1b

The following cells are required to build CLA and FASTCLA type adders, incrementors and comparators:

- ao1f
- oa1f

The optional XOR cells are also recommended for efficient optimization of all adder structures.

Multiplier Cells

The following cells are required to build Booth-encoded multipliers:

- mule2a
- mulpa1b
- mulpa2b

Library Report

You can see many aspects of the technology library by looking the library report (choose Library Report from the View menu). In general, you don't need to look at this file, but you may find it helpful in some cases. It has several primary sections:

- List of named opconds and the associated values of P, V, T
- List of wire load models
- List of MC generic cells and the technology specific equivalent cell, if any.
- List of technology specific cells mapped to MC synthesis cells
- List of pseudo-cells
- List of Dont Use Cells
- List of Untyped Cells
- List of Equivalent Cells

Named Opconds

Use this section to locate a valid named opcond and the values of P, V, and T associated with it. The last column, K, shows the overall delay derating factor.

Wire Load Models

Use the this section to select a valid wire load model name.

Generic Cells

This section shows how the technology library supplies the functionality of cells defined in the MC generic library. The first column is the name of the MC generic library element, and the second column is the name of the corresponding cell in the technology library. If the second column is empty, the MC generic cell has no equivalent cell in the technology library. The third column contains the MC synthesis cell handle, if any. That is, if the third column contains a name, the generic cell is also a synthesis cell (a target during synthesis). The value of the handle is unimportant. However, it is important (but not required) to have native cells that map into these special synthesis cells. The fourth column is the human-readable description of the logic function of the generic cell.

Synthesis Cells

The mapped synthesis cells are listed in this section. The first column is the name of the technology-specific cell. It is followed by the area and the synthesis handle. Unmapped synthesis cell handles do not appear in this section.

Pseudo-Cells

MC creates pseudo-cells to enrich the library for specific datapath functionality. Pseudo-cells are normally inserted into the design only during synthesis and are flattened into non-pseudo-cell primitives before optimization. This section lists the name and area of any pseudo-cells that have been loaded. You can control the loading of pseudo-cells with the **-pl** switch. See Chapter 3 of the *Module Compiler Reference Manual* for details on command-line options.

Dont Use Cells

This section shows all cells that have been marked as “dont use” in the Synopsys library file or through the MC property file. MC does not insert these cells into the design during synthesis or optimization, but you can instantiate them.

Untyped Cells

This section contains a list of the cells that have no special types. These are “normal” library cells that you can instantiate and that MC can insert into the design during optimization.

Equivalent Cells

This section shows cells that are considered logical equivalents by MC. All cells listed on a single line are equivalent and can be swapped for one another.

Example 6-1 Sample Library Report

```
Library Report.  Internal library name lca500k

Operating conditions (PVT)
              Name        P       V        T        K
-------------------------------------------------
        synlibcond     1.00    3.30   25.00    1.00
               NOM     1.00    3.30   25.00    1.00
             WCCOM     1.31    3.13   70.00    1.00
             WCIND     1.31    3.13   85.00    1.00
             WCMIL     1.32    2.97  125.00    1.00
             BCCOM     0.74    3.46    0.00    1.00
             BCIND     0.75    3.46  -40.00    1.00
             BCMIL     0.75    3.63  -55.00    1.00
               TST     1.31    3.30   25.00    1.00

Wire load Models
-------------------------------------------------
      synlinear0      synlinear1      synlinear2    synlinear2.5      synlinear3
      synlinear5     synlinear10            B0X0        B0.5X0.5            B1X1
            B2X2            B3X3            B4X4            B5X5            B6X6
            B7X7            B8X8            B9X9          B10X10          B12X12
          B14X14          B16X16          B18X18          B20X20        L500024B
        L500024C        L500024D        L500024E        L500024P        L500043B
        L500043C        L500043D        L500043E        L500043P        L500055B
        L500055C        L500055D        L500055E        L500055P        L500076B
        L500076C        L500076D        L500076E        L500076P        L500117B
        L500117C        L500117D        L500117E        L500117P        L500157B
        L500157C        L500157D        L500157E        L500157P        L500185B
        L500185C        L500185D        L500185E        L500185P        L500222B
        L500222C        L500222D        L500222E        L500222P        L500290B
        L500290C        L500290D        L500290E        L500290P        L500362B
        L500362C        L500362D        L500362E        L500362P        L500453B
        L500453C        L500453D        L500453E        L500453P        L500529B
        L500529C        L500529D        L500529E        L500529P        L500608B
        L500608C        L500608D        L500608E        L500608P        L500685B
        L500685C        L500685D        L500685E        L500685P        L500795B
        L500795C        L500795D        L500795E        L500795P        L500946B
        L500946C        L500946D        L500946E        L500946P        L500A88B
        L500A88C        L500A88D        L500A88E        L500A88P        L500C94B
        L500C94C        L500C94D        L500C94E        L500C94P        L500F18B
        L500F18C        L500F18D        L500F18E        L500F18P            EXER

  Generic Cells     Maps to
-------------------------------------------------
     fdm1a   FD1SQPTFFScan flip flop
     fdm1c     TIFF Scan flip flop, inverted output
     fdem1a  FD1SLQPTENFFScan flip flop with enable
     fdem1c    TIENFFScan flip flop with enable, inverted output
     fde1a     ENFF Enable flip flop
     fde1c     IENFFEnable flip flop, inverted output
     fd1a    FD1QPFFD flip flop
     fd1c      IFF  D flip flop, inverted output
     ld1b    LD2QPNDLD latch, active low enable
     ld1c      IDL  D latch, inverted output
     ld1a    LD1QPDLD latch
     mule2a    BOOTHENCABooth Encoder
     mulpa1b   BPPA Booth Partial Product Generator
     mulpa2b   BPPSABooth Partial Product Generator
     facs3a    CSAE101 bit full carry select adder, no carry in
     facs4a    CSAE10I1 bit full carry select adder, no carry in
     mx2a    UX21HPX2:1 Mux
     mx2d    UX21LPVMUX2:1 Mux, inverting output
     mx3a    MUX31HP  MUX33:1 Mux
     mx4a    MUX41P  MUX44:1 Mux
     buf1a   BUF9   BUFNon-inverting internal buffer
     inv1a   B4IP   INVInverting internal buffer
```

```
tri1a   BTS4P   TRINon-inverting internal 3-state buffer
ao1f    AO7P   GIIAND2C into OR2B
oa1f    AO6P   GTTOR2C into AND2B
ha1a    HA1P   HA11 bit half adder
fa1a    FA1AP   FA11 bit full adder
fa2a      CSAE01 bit full adder, COUT inverted
fa1b      CSAO01 bit full adder, CI inverted
xor2a   EOP  XOR22-input XOR gate
xor3a   EO3P  XOR33-input XOR gate
xnor2a  ENP XNOR22-input XNOR gate
xnor3a  EN3P XNOR33-input XOR gate
nor2a   NR2P  NOR22-input NOR gate
nor3a   NR3P  NOR33-input NOR gate
nor4a   NR4P  NOR44-input NOR gate
nor5a   NR5P  NOR55-input NOR gate
nor6a   NR6P  NOR66-input NOR gate
nor7a      NOR77-input NOR gate
nor8a   NR8P  NOR88-input NOR gate
or2a    OR2P OR22-input OR gate
or3a    OR3P OR33-input OR gate
or4a    OR4P OR44-input OR gate
or5a      OR5  5-input OR gate
or6a      OR6  6-input OR gate
or7a      OR7  8-input OR gate
or8a      OR8  8-input OR gate
and2a   AN2PAND22-input AND gate
and3a   AN3PAND33-input AND gate
and4a   N4PND4 4-input AND gate
and5a     AND5 5-input AND gate
and6a     AND6 6-input AND gate
and7a     AND7 7-input AND gate
and8a     AND8 8-input AND gate
nand2a  ND2PNAND22-input NAND gate
nand3a  D3PND3 3-input NAND gate
nand4a  D4PND4 4-input NAND gate
nand5a  D5PND5 5-input NAND gate
nand6a  ND6PNAND66-input NAND gate
nand7a    NAND77-input NAND gate
or8i    ND8PNAND88-input OR gate, eight inputs inverted
facs1b  CSAO1  1 bit full carry select adder, CI inverted
facs2a  CSAE1  1 bit full carry select adder, COUT inverted
and2b          2-input AND gate, one input inverted
and2c   NR2P   2-input AND gate, two inputs inverted
and3b          3-input AND gate, one input inverted
and3c          3-input AND gate, two inputs inverted
and3d   NR3P   3-input AND gate, three inputs inverted
and4b          4-input AND gate, one input inverted
and4c          4-input AND gate, two inputs inverted
and4d          4-input AND gate, three inputs inverted
and4e   NR4P   4-input AND gate, four inputs inverted
and5b          5-input AND gate, one input inverted
and5c          5-input AND gate, two inputs inverted
and5d          5-input AND gate, three inputs inverted
and5e          5-input AND gate, four inputs inverted
and5f   NR5P   5-input AND gate, five inputs inverted
and6g   NR6P   6-input AND gate, six inputs inverted
and8i   NR8P   8-input AND gate, eight inputs inverted
ao1a           AND2A into OR2A
ao1b           AND2B into OR2A
ao1c           AND2C into OR2A
ao1d           AND2A into OR2B
ao1e           AND2B into OR2B
ao2a           AND2A into OR3A
ao2b           AND2B into OR3A
ao2c           AND2C into OR3A
ao2d           AND2A into OR3B
ao2e           AND2B into OR3B
ao2f           AND2C into OR3B
ao2g           AND2A into OR3C
```

ao2h		AND2B into OR3C
ao2i	AO3P	AND2C into OR3C
ao3a		AND3A into OR2A
ao3b		AND3B into OR2A
ao3c		AND3C into OR2A
ao3d		AND3D into OR2A
ao3e		AND3A into OR2B
ao3f		AND3B into OR2B
ao3g		AND3C into OR2B
ao3h		AND3D into OR2B
ao4a		AND2A, AND2A into OR2A
ao4b		AND2B, AND2A into OR2A
ao4c	EON1P	AND2C, AND2A into OR2A
ao4d		AND2B, AND2B into OR2A
ao4e		AND2B, AND2C into OR2A
ao4f	AO4P	AND2C, AND2C into OR2A
ao5a		AND3A, AND2A into OR2A
ao5b		AND3B, AND2A into OR2A
ao6a		Majority gate
ao7a		Three AND2A into OR3A
ao7g		Three AND2C into OR3A
ao8a		Two AND2A into OR3A
ax1a		AND2A into XOR2A
buf2a		Inv, Non-inverting internal buffer
fac1b		1 bit full adder, CI inverted
fac2a		1 bit full adder, COUT inverted
ha1b		Half Adder, active low carry in
ha2a		Half Adder, inverted carry out
faccs1b		1 bit full carry select adder, CI inverted
faccs2a		1 bit full carry select adder, COUT inverted
faccs3a		1 bit full carry select adder, no carry in
hacs1b		1 bit full carry select half adder, active low carry in
hacs2a		1 bit full carry select half adder, inverted carry out
hacs3a		1 bit full carry select half adder, no carry in
mulpa3b		Booth Partial Product Generator
mx4e		4:1 Mux inverted output
nand2b		2-input NAND gate, one input inverted
nand2c	OR2P	2-input NAND gate, two inputs inverted
nand3b		3-input NAND gate, one input inverted
nand3c		3-input NAND gate, two inputs inverted
nand3d	OR3P	3-input NAND gate, two inputs inverted
nand4b		4-input NAND gate, one input inverted
nand4c		4-input NAND gate, two inputs inverted
nand4d		4-input NAND gate, three inputs inverted
nand4e	OR4P	4-input NAND gate, four inputs inverted
nand5b		5-input NAND gate, one input inverted
nand5c		5-input NAND gate, two inputs inverted
nand5d		5-input NAND gate, three inputs inverted
nand5e		5-input NAND gate, four inputs inverted
nand5f		5-input NAND gate, five inputs inverted
nand8a	ND8P	8-input NAND gate
nor2b		2-input NOR gate, one input inverted
nor2c	AN2P	2-input NOR gate, two inputs inverted
nor3b		3-input NOR gate, one inputs inverted
nor3c		3-input NOR gate, two inputs inverted
nor3d	AN3P	3-input NOR gate, three inputs inverted
nor4b		4-input NOR gate, one input inverted
nor4c		4-input NOR gate, two inputs inverted
nor4d		4-input NOR gate, three inputs inverted
nor4e	AN4P	4-input NOR gate, four inputs inverted
nor5b		5-input NOR gate, one input inverted
nor5c		5-input NOR gate, two inputs inverted
nor5d		5-input NOR gate, three inputs inverted
nor5e		5-input NOR gate, four inputs inverted
nor5f		5-input NOR gate, five inputs inverted
oa1a		OR2A into AND2A
oa1b		OR2B into AND2A
oa1c		OR2C into AND2A
oa1d		OR2A into AND2B

oa1e		OR2B into AND2B
oa2a		OR2A into AND3A
oa2b		OR2B into AND3A
oa2c		OR2C into AND3A
oa2d		OR2A into AND3B
oa2e		OR2B into AND3B
oa2f		OR2C into AND3B
oa2g		OR2A into AND3C
oa2h		OR2B into AND3C
oa2i	AO1P	OR2C into AND3C
oa3a		OR3A into AND2A
oa3b		OR3B into AND2A
oa3c		OR3C into AND2A
oa3d		OR3D into AND2A
oa3e		OR3A into AND2B
oa3f		OR3B into AND2B
oa3g		OR3C into AND2B
oa3h		OR3D into AND2B
oa4a		OR2A, OR2A into AND2A
oa4b		OR2B, OR2A into AND2A
oa4c	EO1P	OR2C, OR2A into AND2A
oa4d		OR2B, OR2B into AND2A
oa4e		OR2B, OR2C into AND2A
oa4f	AO2P	OR2C, OR2C into AND2A
oa5a		OR3A, OR2A into AND2A
oa5b		OR3B, OR2A into AND2A
oa7a		Three OR2A into AND3A
oa7g	AO11P	Three OR2C into AND3A
oa8a		Two OR2A into AND3A
or2b		2-input OR gate, one input inverted
or2c	ND2P	2-input OR gate, two inputs inverted
or3b		3-input OR gate, one input inverted
or3c		3-input OR gate, two inputs inverted
or3d	ND3P	3-input OR gate, three inputs inverted
or4b		4-input OR gate, one input inverted
or4c		4-input OR gate, two inputs inverted
or4d		4-input OR gate, three inputs inverted
or4e	ND4P	4-input OR gate, four inputs inverted
or5b		5-input OR gate, one input inverted
or5c		5-input OR gate, two inputs inverted
or5d		5-input OR gate, three inputs inverted
or5e		5-input OR gate, four inputs inverted
or5f	ND5P	5-input OR gate, five inputs inverted
or6g	ND6P	6-input OR gate, six inputs inverted
xa1a		XOR2A into AND2A
xa1b		XOR2B into AND2A
xa1d		XOR2B into AND2B
xor2b	ENP	2-input XOR gate, one input inverted
xor3b	EN3P	3-input XOR gate, one input inverted
fd1b	N1QP	D flip flop, active low clock
fd2a		D flip flop, active low clear
fd3a		D flip flop, active low preset
fd4a		D flip flop, active low clear and preset
fd4b		D flip flop, active low clear, preset and clock
fd6a		D flip flop, with Q & QN
fd7a		D flip flop, active low clear, with Q & QN
fd8a		D flip flop, active low preset, with Q & QN
fd9a		D flip flop, active low clear and preset, with Q & QN
fde2a		enable flip flop, active low clear
fdm1b		Scan flip flop, active low clock
fdm1e		Scan flip flop, D0 inverted
fdm1i		Scan flip flop, D1 inverted
fdm2a		Scan flip flop, active low clear
fdm3a		Scan flip flop, active low preset
fdm4b		Scan flip flop, active low clear, preset and clock
fdm5a		3:1 Mux D flip flop
fdm7a		Scan flip flop, active low clear, with Q & QN
fjk1a	K1QP	JK flip flop
1d2a		D latch, active low clear

ld2b		D latch, active low clear and enable
ld4a		D latch, active low clear and preset
ldm1a		Scan latch
ldm1b		Scan latch, active low enable
ldm1c		Scan latch, inverted output
ldm2a		Scan latch, active low clear
tri1b	BTS5P	Inverting internal 3-state buffer

Synthesis Cells	Area	
LD1QP	5.00	DL
BUF9	6.00	BUF
NR2P	2.00	NOR2
ND2P	2.00	NAND2
MUX41P	6.00	MUX4
NR3P	3.00	NOR3
FD1SQP	9.00	TFF
LD2QP	5.00	NDL
ND3P	3.00	NAND3
B4IP	4.00	INV
NR4P	4.00	NOR4
AO6P	3.00	GTT
MUX21LP	4.00	INVMUX
OR2P	2.00	OR2
MUX31HP	6.00	MUX3
ND4P	4.00	NAND4
NR5P	5.00	NOR5
ENP	4.00	XNOR2
AO7P	3.00	GII
OR3P	3.00	OR3
FD1QP	7.00	FF
ND5P	5.00	NAND5
NR6P	5.00	NOR6
EOP	4.00	XOR2
FA1AP	8.00	FA1
AN2P	2.00	AND2
EO3P	6.00	XOR3
OR4P	3.00	OR4
ND6P	5.00	NAND6
HA1P	6.00	HA1
AN3P	3.00	AND3
NR8P	6.00	NOR8
AN4P	3.00	AND4
MUX21HP	5.00	MUX
EN3P	6.00	XNOR3
ND8P	6.00	NAND8
BTS4P	4.00	TRI
FD1SLQP	11.00	TENFF

Pseudo Cells	Area
mc__buf1a0	0.00
mc__mx2a0	3.00
mc__mx2d0	2.00
mc__buf1a1	5.00
mc__mx2a1	7.00
mc__mx2d1	6.00

Dont Use Cells	Area
D004GBVA	2.00
DIFAMP2	37.00
DIFAMP1	12.00
H0802P	2.00
CLKC16I	0.00
PHASE90CH	150.00
PLL2540GA	328.00
CLKC8I	0.00
BAL1	-1.00

BAL1A	1.00
VARDELIN	1.00
D004VA	4.00
PLL5080GA	310.00
CAP24PF	448.00
PLLDLYQ	8.00
H0261A	13.00
D0CSAH	11.00
D0CSAL	11.00
DIFH2C	0.00
CLKC2I	0.00
CLKC12I	0.00
H0804	2.00
H0802	2.00
CLKC4I	0.00
LCLKBUF1	3.00
PLL80GA	310.00
LCLKBUF2	4.00
LCLKBUF3	5.00
CMLREFCORE	99.00
PHASE360CH	150.00
PLL5590GA	310.00
CLK2QFD1S	9.00
CLK2QLD1	4.00
PORPGM	10.00
D004GA	4.00
D004	10.00

Untyped Cells	Area
FDN2Q	7.00
AO3P	4.00
FD1SSQ	10.00
FD1SSO	9.00
FJK3SQP	13.00
FD2SQP	10.00
FD2SQ	9.00
DELAY1	6.00
DELAY2	10.00
FD1SSQP	11.00
DELAY4	14.00
EON1P	4.00
BUF1	1.00
BUF2	2.00
BUF3	2.00
BUF4	2.00
FJK2SQ	12.00
FD3QP	9.00
BUF5	3.00
BUF6	3.00
BUF7	5.00
BUF8	4.00
FJK3QP	11.00
AO4P	4.00
LD4Q	5.00
LD1S2Q	9.00
ROSC3060GA	120.00
FD4SQP	10.00
EN	3.00
EO	3.00
B1A	3.00
LD3Q	5.00
AN2	2.00
AN3	2.00
AN4	3.00
FD1SSOP	10.00
BTS4	3.00
BTS5	3.00
FD3SQ	10.00

FD3SQM	13.00
LSR2BUF	10.00
FD3SQP	11.00
NR2	1.00
MUX81P	12.00
NR3	2.00
NR4	2.00
B2A	4.00
NR5	4.00
NR6	5.00
LD1S2QP	9.00
NR8	6.00
CHAN4	270.00
FJK3SQ	13.00
FD4QP	8.00
FD2SL2	13.00
AO1	2.00
LD2Q	5.00
FDN2SQ	9.00
EN3	6.00
AO2	2.00
AO3	2.00
FD1Q	6.00
AO4	2.00
AO6	2.00
FD2ESS	11.00
AO7	2.00
MUX81	12.00
SFD2	8.00
IVA	1.00
EO1	3.00
LD1Q	4.00
EO3	6.00
MUX61HP	14.00
FD2Q	7.00
FA1A	7.00
IVP	1.00
FJK1SQP	11.00
DELAY05	4.00
FJK1QP	9.00
FJK3Q	10.00
BHD1A	-1.00
LSR2	6.00
FD4SQ	9.00
B4I	2.00
LSR0	3.00
LD3QP	5.00
FDN2QP	8.00
EON1	3.00
FD3Q	9.00
OR2	2.00
B5IP	4.00
OR3	2.00
OR4	3.00
MUX21H	4.00
MUX21L	3.00
AO11P	6.00
FJK2Q	9.00
AO11	5.00
AO12	6.00
HA1	5.00
IVAP	2.00
IV	1.00
B5I	2.00
ND2	1.00
ND3	2.00
ND4	2.00
FD4Q	7.00
ND5	4.00

ND6	5.00
FD2ESSP	12.00
ND8	6.00
FJK2SQP	13.00
MUX61H	14.00
FD1SQ	9.00
AO12P	6.00
AO1P	4.00
FJK1Q	8.00
SFD2P	9.00
FJK1SQ	11.00
FD2QP	8.00
BTS5P	4.00
FD2SL2P	14.00
FD1SLQ	10.00
FJK2QP	10.00
EO1P	4.00
LD4QP	5.00
FDN1QP	7.00
SCN4IM	18.00
FDN1Q	6.00
AO2P	4.00
MUX31H	5.00
MUX41	6.00
FT2Q	8.00
FDN2SQP	10.00

```
Equivalent Cells
---------------------------------------------------
      LD2QP LD2Q ld1b
      ld1c
      LD1QP LD1Q ld1a
      ldm1b
      ldm1c
      ldm1a
      FDN1QP FDN1Q fd1b
      fd1c
      FD1Q FD1QP fd1a
      fd6a
      fde1c
      FJK1QP FJK1Q fjk1a
      fde1a
      SFD2 SFD2P
      fdm1b
      fdm1c
      fdm1i
      fdm1e
      FD1SQP FD1SQ fdm1a
      FD1SSO FD1SSOP
      fdem1c
      FJK1SQP FJK1SQ
      FD1SLQ FD1SLQP fdem1a
      gnd_generic gnd_real
      vdd_generic vdd_real
      B1A B4IP B2A IVA IVP B4I B5IP IVAP IV B5I inv1a
      DELAY1 DELAY2 DELAY4 BUF1 BUF2 BUF3 BUF4 BUF5 BUF6 BUF7 BUF8 BUF9 DELAY05 buf1a mc_
_buf1a0 mc__buf1a1
      buf2a
      BTS5 BTS5P tri1b
      BTS4 BTS4P tri1a
      NR2P NR2 nor2a and2c
      and2b nor2b
      EO EOP xor2a
      ND2P ND2 nand2a or2c
      AN2 AN2P and2a nor2c
      EN ENP xnor2a xor2b
      nand2b or2b
      OR2P OR2 or2a nand2c
      faccs3a
```

```
ha1b
hacs3a
ha2a
HA1P HA1 ha1a
NR3P NR3 nor3a and3d
and3c nor3b
AO6P AO6 oa1f
and3b nor3c
xa1d
oa1e
oa1d
fac2a
MUX21LP MUX21L mx2d mc__mx2d0 mc__mx2d1
AO7 AO7P ao1f
xa1a
oa1c
ao1e
EN3 EN3P xnor3a xor3b
ax1a
ND3P ND3 nand3a or3d
AN3 AN3P and3a nor3d
xa1b
oa1b
fac1b
ao1d
EO3 EO3P xor3a
oa1a
ao1c
MUX21H MUX21HP mx2a mc__mx2a0 mc__mx2a1
ao1b
nand3b or3c
ao6a
ao1a
nand3c or3b
OR3P OR3 or3a nand3d
hacs1b
fa2a
hacs2a
fa1b
FA1A FA1AP fa1a _test_fa1a
facs4a
facs3a
mule2a
NR4 NR4P nor4a and4e
and4d nor4b
AO1 AO1P oa2i
and4c nor4c
oa2h
oa2g
oa2f
oa3h
and4b nor4d
oa2e
oa2d
oa3g
oa3f
oa3e
ao3h
ao3g
AO4P AO4 ao4f
ao4e
AO2 AO2P oa4f
oa2c
ao3f
ao4d
oa4e
EO1 EO1P oa4c
AO3P AO3 ao2i
oa3d
```

```
ao2h
ND4P ND4 nand4a or4e
AN4 AN4P and4a nor4e
oa2b
ao3e
oa2a
EON1P EON1 ao4c
ao4b
oa3c
oa4d
ao2g
oa3b
oa4b
oa3a
ao3d
ao3c
ao3b
ao2f
ao2e
nand4b or4d
ao3a
ao4a
ao2d
oa4a
ao2c
ao2b
nand4c or4c
ao2a
nand4d or4b
OR4 OR4P or4a nand4e
faccs2a
faccs1b
NR5 NR5P nor5a and5f
and5e nor5b
and5d nor5c
and5c nor5d
and5b nor5e
mulpa1b mulpa3b
ND5P ND5 nand5a or5f
and5a nor5f
ao5b
oa8a
MUX31HP MUX31H mx3a
mulpa2b
nand5b or5e
ao5a
oa5b
oa5a
nand5c or5d
ao8a
nand5d or5c
nand5e or5b
or5a nand5f
facs2a
facs1b
NR6 NR6P nor6a and6g
AO11P AO11 oa7g
ao7g
mx4e
ND6P ND6 nand6a or6g
and6a
MUX41P MUX41 mx4a
oa7a
ao7a
or6a
NR8 NR8P nor8a and8i
AO12 AO12P
ND8 ND8P or8i nand8a
and8a
```

7 Layout Support

This chapter presents an overview of layout issues, options, and strategies. It describes the types of layout information provided by Module Compiler and suggests ways in which that information can be used to produce effective layouts.

Chapter 7 discusses the following topics:

- Layout issues
- Layout information provided by MC
- The format of the layout file
- Strategies for using the layout information
- A detailed datapath example

Layout Issues

MC provides detailed placement information that can be used to control the placement of instances in the design in a variety of placement approaches. The entire datapath can be bit-sliced, bit-stacked, or floorplanned, or a combination of the these approaches can be used. You can choose any technique for each block of the design, based on high-level floorplanning constraints and the complexity and regularity of the design.

Note that MC provides only relative placement information. You must convert this information into detailed placement information for the specific tools and vendor library you are using.

For the sake of simplicity, it is assumed that bits are arranged in rows, and that cells can be abutted along the bit slice. The instances making up a function span multiple bits or rows and are arranged vertically in a column. Of course, your layout system might require that the bits be vertical rather than horizontal; this change does not impact the general operations discussed in this section.

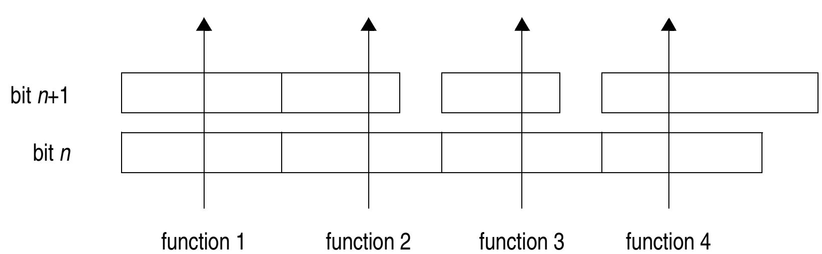
Bit-Slicing

Bit-slicing, as the name suggests, arranges the design into slices (rows), one for each bit. The bits are generally arranged in consecutive order. In bit-slicing, the columns represent similar functional elements, such as flip-flops or multiplexors. Control signals, which run vertically through a column, cost very little in a bit-sliced design. Forcing similar cells to be arranged in columns arranges the control signal pins in a straight (or nearly straight) vertical line, This eliminates the need for jogs and routing resources when you wire the control nets.

The drawback of bit-slicing is the possible under-utilization of area that can occur when a column contains cells of different widths. Because each column is aligned, these different sized cells cause wasted area in each column, even if the average width of all slices is the same.

In the simple bit-slice figure below, the two bit slices are stacked one above the other. Control signals are shown running vertically over the instances. Note that the instances are column-aligned, so that some area is wasted when different size instances are placed into the same column.

Figure 7-1 Bit-Sliced Structure

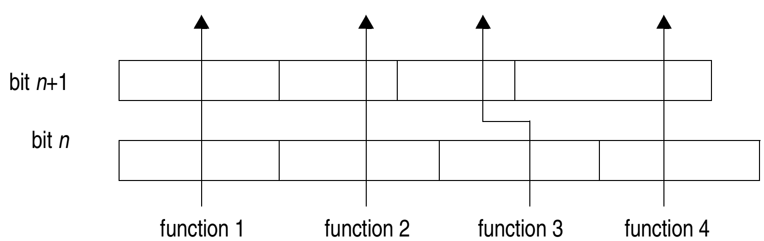


Bit-Stacking

This technique is similar to bit-slicing except that there are no aligned columns. Each bit or row is packed to remove any space between cells. This avoids the waste associated with different-sized cells in a column, and the overall utilization is determined solely by the variation of the average widths of the rows. At the same time, control signals may no longer be straight wires.

In the simple bit-stacked circuit below, the two bits are stacked one above the other. Control signals are shown running vertically over the instances and might need more jogs and routing resources. Note that the instances are not column (function) aligned, so the amount of wasted area relative to the bit slice is reduced.

Figure 7-2 Bit-Stacked Structure



Information Provided

Layout Information

MC provides two types of layout information for each block or signal in the design, depending on whether placement is known. When the placement is known, instances are assigned to a row and column in the block, based upon the architecture of the block. When the placement is not known, instances are associated with the block, and should be placed near the other instances in the block. All instances are associated with some block in the design.

The layout information in the layout file is arranged in an input-to-output order, which is the order in which the blocks of the design are synthesized. As long as there are no loops in the input description, the order of blocks in the layout file is the same as their order in the input description.

Statistical Information

MC also provides statistical information that you can use to determine which type of layout technique to pursue on a block-by-block basis.

- For each synthesized block with placed instances, the size in rows, columns, and total area is provided.
- Two measures of utilization are provided. The **slot utilization** provides the percentage of slots (there are row*column slots in the block) occupied. The **area utilization** provides the ratio of occupied area to bounding box area for a bit-sliced implementation.

Utilization and Layout Strategies

When these utilization measures are high, bit slicing and bit stacking are more effective. When utilization is low, floorplanning and traditional place and route techniques are more effective. A utilization value of 1.0 indicates there is no wasted space while a value of 0.0 indicates all space is wasted.

A Layout Example

The example below shows the syntax of the layout file for a four-input adder with a **fastcla** final adder. In this example, there is only one block (*Z*). It has 80 slots arranged as 8 bits by 10 columns. The total used area is 5790, and the area occupied by the bit slice is 7400, resulting in an area utilization of 78%. 67 of 80 slots are occupied resulting in a slot utilization of 84%.

The placed instances are arranged by bit (row) and are listed in one of the **bit** constructs. Following **bit** is the bit number. The instances for each column are then listed in order with empty slots denoted by “---”.

Any unplaced instances are listed in the **associated** construct. These are instances that belong in or near the block Z, but which have no specific row-column placement. These instances can be placed in the unused slots.

Example 7-1 A Sample Layout File

```
(placement Z
         (size 8 10)
         (areautil 5790 7400  0.78)
         (slotutil 67 80  0.84)
         (bit   0 I118  --- I135 I136 I151  --- I169  ---  --- I197 )
         (bit   1 I119 I120 I137 I138 I153  --- I171  ---  --- I198 )
         (bit   2 I121 I122 I139 I140 I155 I156 I173  ---  --- I199 )
         (bit   3 I123 I124 I141 I142 I157 I158 I175  ---  --- I200 )
         (bit   4 I125 I126 I143 I144 I159 I160 I177 I178 I187 I201 )
         (bit   5 I127 I128 I145 I146 I161 I162 I179 I180 I189 I202 )
         (bit   6 I129 I130 I147 I148 I163 I164 I181 I182 I191 I203 )
         (bit   7 I131 I132  ---  --- I165 I166 I183 I184 I193 I204 )
         (associated I300 I301)
)
```

This example was chosen to illustrate some interesting aspects of the placement information. The Wallace tree full adders are in the first two columns. Columns three and four contain the initial G and P logic. The next five columns contain the carry propagation tree, and the last column contains the XOR gates to form the sum.

Using the Layout Information

It is possible to exploit the layout information in several ways.

- All layout information can be ignored and the design can be placed by a traditional ASIC place and route tool.
- The instances for one or more blocks can be grouped together in a floorplan-guided layout.
- The row and column information can be used to place each instance in a bit-slice structure.
- The row information can be used to create a bit-stacked placement.

For the last two cases, the place and route tool must perform the circuit routing but not the placement. You can also use a combination of these techniques.

Traditional ASIC Place and Route

This is the approach normally taken for synthesized designs. There is nothing new here: you ignore the information in the placement file and proceed with place and route as with any other ASIC design. Instance names can be tagged using the naming options described earlier to facilitate grouping in the placer.

Floorplanning

The layout information makes it very easy to floorplan your design. In this case, you ignore the row and column information and group the instances within each block together. One or more blocks from the design can be combined to form the desired number of floorplanning groups. It is also possible use groups and the instance naming options to floorplan the design without using the layout file.

Bit-Slicing

In bit-slicing, bits are stacked vertically in the order provided. The instances of each bit are placed horizontally in the order provided, and the instances in a column are aligned vertically. Slots corresponding to empty column markers are left empty. Component instances of pseudo-cells, separated by colons (:), are placed horizontally in the appropriate column in the order given. Essentially, pseudo-cell components are bit-stacked within a bit-slice column.

Bit-Stacking

In bit-stacking, bits are stacked vertically in the order provided by the layout file. The instances of each bit are placed horizontally in the order provided, but no column alignment is performed. Empty column markers ("---") are ignored. Component instances of pseudo-cells are separated by colons (:) and placed in order.

How MC Uses the Information

As MC synthesizes a circuit, the relative placement information is attached to each instance. This information is carried throughout the entire synthesis and optimization process. Some structures are inherently regular and tend to have high utilization, while others are inherently irregular and tend to have lower utilization. Even regular structures can degenerate into irregular structures under certain conditions, so you should always be aware of the balance between reducing area and maximizing utilization. The logic optimizer sometimes reduces area and improves performance by creating a less regular circuit with lower utilization.

What Bit-Slices Well

It is important to consider which structures bit-slice or bit-stack well. Highly regular structures are better candidates for bit-slicing. Below is a list of structures and some comments about the regularity of each.

Table 7-1 Regularity of Datapath Structures

Structure	Regularity
Shifters	High, some portion could degenerate in AND gates
Rotators	High
Shift Registers	High
Latch and FF RAMs	High
MUX-based Multiplexors	High
AND-OR Multiplexors	Med, decoders are separate structure
fastcla adder	Med-High, end effects limit utilization
cla adder	Low-Med, sparse carry tree limits utilization
clsa adder	Low-High, depends on incoming delay skews and performance requirements
csa adder	Med-High
ripple adder	High
Multipliers	Low, different input and output widths cause problems.
magnitude comparators	Med-High (if compressed to remove empty slots)
equality comparators	Med-High (if bit oriented logic is force into one column)
buffer-trees	Low (left unplaced)

Effects of Logic Optimization

Be careful not to confuse high utilization with high circuit quality. It is possible to build very high utilization circuits that are not very efficient. The logic optimizer makes local changes to the network to improve the total area and performance of the circuit without concern for utilization. It can make several types of changes, such as instance removal, instance up/down sizing, instance reduction. None of these optimizations destroys the inherent structure of the circuit. Rather, the optimizations might make the design locally less uniform. In some cases, these optimization result in a smaller but equally regular circuit.

Instance removal is performed when a given instance is not needed. In the adder example above, some portions of the Wallace tree and the carry propagate adder were simply not needed and were removed from the circuit. Removing these instances improves performance by reducing net loading and increases the total amount of routing resource available. Clearly, utilization would be improved by leaving these instances in the circuit.

Instance upsizing and downsizing is used to reduce area in noncritical portions of the design and to improve performance in the critical portions. When a column contains both high and low drive-strength versions of a cell, utilization typically falls, because low drive-strength cells are generally smaller than their high-drive counterparts. Reducing some of the cells reduces the net loading and makes more routing resource available and improves overall circuit quality.

Instance reductions create effects similar to those produced by upsizing and downsizing. In instance reduction, asymmetric circuit constraints cause a column to contain fundamentally different types of cells. For example, part of a barrel shifter column might contain AND gates because zero is being shifted in from the left or right. These swaps improve the occupied area and performance of the circuit. As in the other cases, overall utilization can drop.

In general, MC produces the highest utilization circuit when logic optimization is disabled. The resulting circuit is generally larger and slower than that obtained after optimization, so leaving logic optimization enabled is recommended even when bit-slicing.

A Detailed Example

The datapath example below contains elements commonly found in bit-sliced implementations: shifts, rotates, multiplexors, technology independent gate instantiation, registers, and adders.

Example 7-2 Alu Example

```
module dp (n,A,B,C,Z,S1,S2,fa);
integer n=8;
     string fa="fastcla";
     input [2:0] S2;
     input [1:0] S1;
     input [n-1:0] A,B,C;
     output [n-1:0] Z;
     wire [n-1:0] Z1=S1 ? sreg(A)<<<S2 : sreg(B) :
               sreg(C,2);
     wire [n-1:0] Z2=xor2a(Z1,C);
     wire [n-1:0] Z3=sreg(Z2)>>S2;
     directive (fatype=fa);
     Z=Z3+A+B+C;
endmodule
```

Before logic optimization, the area is 2167 with a delay of 4.4 nanoseconds. Table 7-2 shows the connection between the names in the placement file and those in the input description. Note that complex statements like Z1 are associated with several subdesigns.

See “Naming” in Chapter 10 for a more complete description of naming objects in Module Compiler.

Table 7-2 Names in Placement Files and Input Descriptions

Placement File Name	Input Function
Z1_out__1	sreg(A), part of Z1
Z1_out_6__1	sreg(B), part of Z1
Z1_out_9__1	sreg(C,2), part of Z1
Z1_10_	left rotator, part of Z1
Z1	3 to 1 Multiplexor, end of Z1
Z2	2-input XOR
Z3_out__1	sreg(Z2), part of Z3
Z3	right shift, end of Z3
Z_1_	4 input adder

The following example shows the placement information before logic optimization. Note that the blocks are listed in an input-to-output order; a complete bit-slice could be created by concatenating all of the blocks in a left to right order.

An experienced designer will quickly recognize that MC is generating traditional circuits. The *sreg*() functions result in a column of *n* flip-flops for each shift register stage. Shifters and rotators have log2(n) columns of *n* 2-to-1 multiplexors. MUX-based multiplexors are similar, but may have columns of multiplexors with more than 2 inputs.

Example 7-3 Relative Placement Before Logic Optimization

```
(placement S2
     (size 3 1)
     (areautil 210 210  1.00)
     (slotutil 3 3  1.00)
     (bit   0 I53 )
     (bit   1 I44 )
     (bit   2 I35 )
)
(placement Z1_out__1
     (size 8 1)
     (areautil 1400 1400  1.00)
     (slotutil 8 8  1.00)
     (bit   0  I3 )
     (bit   1  I4 )
     (bit   2  I5 )
     (bit   3  I6 )
     (bit   4  I7 )
     (bit   5  I8 )
     (bit   6  I9 )
     (bit   7 I10 )
)
(placement Z1_out_6__1
     (size 8 1)
     (areautil 1400 1400  1.00)
     (slotutil 8 8  1.00)
     (bit   0 I11 )
     (bit   1 I12 )
     (bit   2 I13 )
     (bit   3 I14 )
     (bit   4 I15 )
     (bit   5 I16 )
     (bit   6 I17 )
     (bit   7 I18 )
)
(placement Z1_out_9__1
     (size 8 2)
     (areautil 2800 2800  1.00)
     (slotutil 16 16  1.00)
     (bit   0 I19 I27 )
     (bit   1 I20 I28 )
     (bit   2 I21 I29 )
     (bit   3 I22 I30 )
     (bit   4 I23 I31 )
     (bit   5 I24 I32 )
     (bit   6 I25 I33 )
     (bit   7 I26 I34 )
)
(placement Z1_10_
     (size 8 3)
     (areautil 2040 2040  1.00)
     (slotutil 24 24  1.00)
     (bit   0 I36 I45 I54 )
     (bit   1 I37 I46 I55 )
     (bit   2 I38 I47 I56 )
     (bit   3 I39 I48 I57 )
     (bit   4 I40 I49 I58 )
     (bit   5 I41 I50 I59 )
```

```
    (bit   6 I42 I51 I60 )
    (bit   7 I43 I52 I61 )
)
(placement Z1
    (size 8 1)
    (areautil 1160 1160  1.00)
    (slotutil 8 8  1.00)
    (bit   0 I62 )
    (bit   1 I63 )
    (bit   2 I64 )
    (bit   3 I65 )
    (bit   4 I66 )
    (bit   5 I67 )
    (bit   6 I68 )
    (bit   7 I69 )
)
(placement Z2
    (size 8 1)
    (areautil 680 680  1.00)
    (slotutil 8 8  1.00)
    (bit   0 I70 )
    (bit   1 I71 )
    (bit   2 I72 )
    (bit   3 I73 )
    (bit   4 I74 )
    (bit   5 I75 )
    (bit   6 I76 )
    (bit   7 I77 )
)
(placement Z3_out__1
    (size 8 1)
    (areautil 1400 1400  1.00)
    (slotutil 8 8  1.00)
    (bit   0 I78 )
    (bit   1 I79 )
    (bit   2 I80 )
    (bit   3 I81 )
    (bit   4 I82 )
    (bit   5 I83 )
    (bit   6 I84 )
    (bit   7 I85 )
)
(placement Z3
    (size 8 3)
    (areautil 2040 2040  1.00)
    (slotutil 24 24  1.00)
    (bit   0 I86  I94 I102 )
    (bit   1 I87  I95 I103 )
    (bit   2 I88  I96 I104 )
    (bit   3 I89  I97 I105 )
    (bit   4 I90  I98 I106 )
    (bit   5 I91  I99 I107 )
    (bit   6 I92 I100 I108 )
    (bit   7 I93 I101 I109 )
)
(placement Z_1_
    (size 8 11)
    (areautil 8095 9200  0.88)
    (slotutil 79 88  0.90)
```

```
    (bit   0 I110  --- I126 I127 I142 I143 I160 I161  ---  --- I188 )
    (bit   1 I111 I112 I128 I129 I144 I145 I162 I163  ---  --- I189 )
    (bit   2 I113 I114 I130 I131 I146 I147 I164 I165  ---  --- I190 )
    (bit   3 I115 I116 I132 I133 I148 I149 I166 I167  ---  --- I191 )
    (bit   4 I117 I118 I134 I135 I150 I151 I168 I169 I178 I179 I192 )
    (bit   5 I119 I120 I136 I137 I152 I153 I170 I171 I180 I181 I193 )
    (bit   6 I121 I122 I138 I139 I154 I155 I172 I173 I182 I183 I194 )
    (bit   7 I123 I124 I140 I141 I156 I157 I174 I175 I184 I185 I195 )
    (associated  I187 I186 I177 I176 I159 I158 I125)
)
```

Now consider the same circuit after logic optimization. The area has dropped to 1651 and the delay has dropped to 4.1 ns. Notice that the utilization of some blocks has dropped. The rotator utilization dropped due to downsizing of some of the instances. The shifter utilization dropped due to downsizing and logical reduction of some instances. However, many of the blocks sustained substantial changes during logic optimization while retaining high utilization.

Example 7-4 Relative Placement after Logic Optimization

```
'(placement S2
     (size 3 1)
     (areautil 210 210  1.00)
     (slotutil 3 3  1.00)
     (bit   0 I53 )
     (bit   1 I44 )
     (bit   2 I35 )
)
(placement Z1_out__1
     (size 8 1)
     (areautil 920 920  1.00)
     (slotutil 8 8  1.00)
     (bit   0  I3 )
     (bit   1  I4 )
     (bit   2  I5 )
     (bit   3  I6 )
     (bit   4  I7 )
     (bit   5  I8 )
     (bit   6  I9 )
     (bit   7 I10 )
)
(placement Z1_out_6__1
     (size 8 1)
     (areautil 920 920  1.00)
     (slotutil 8 8  1.00)
     (bit   0 I11 )
     (bit   1 I12 )
     (bit   2 I13 )
     (bit   3 I14 )
     (bit   4 I15 )
     (bit   5 I16 )
     (bit   6 I17 )
     (bit   7 I18 )
)
(placement Z1_out_9__1
     (size 8 2)
     (areautil 1840 1840  1.00)
     (slotutil 16 16  1.00)
     (bit   0 I19 I27 )
     (bit   1 I20 I28 )
     (bit   2 I21 I29 )
     (bit   3 I22 I30 )
     (bit   4 I23 I31 )
     (bit   5 I24 I32 )
     (bit   6 I25 I33 )
     (bit   7 I26 I34 )
)
(placement Z1_10_
     (size 8 3)
     (areautil 1840 2040  0.90)
     (slotutil 24 24  1.00)
     (bit   0 I36 I45 I54 )
     (bit   1 I37 I46 I55 )
     (bit   2 I38 I47 I56 )
     (bit   3 I39 I48 I57 )
     (bit   4 I40 I49 I58 )
     (bit   5 I41 I50 I59 )
```

```
    (bit   6 I42 I51 I60 )
    (bit   7 I43 I52 I61 )
)
(placement Z1
    (size 8 1)
    (areautil 1160 1160  1.00)
    (slotutil 8 8  1.00)
    (bit   0 I62 )
    (bit   1 I63 )
    (bit   2 I64 )
    (bit   3 I65 )
    (bit   4 I66 )
    (bit   5 I67 )
    (bit   6 I68 )
    (bit   7 I69 )
)
(placement Z2
    (size 8 1)
    (areautil 680 680  1.00)
    (slotutil 8 8  1.00)
    (bit   0 I70 )
    (bit   1 I71 )
    (bit   2 I72 )
    (bit   3 I73 )
    (bit   4 I74 )
    (bit   5 I75 )
    (bit   6 I76 )
    (bit   7 I77 )
)
(placement Z3_out__1
    (size 8 1)
    (areautil 1220 1400  0.87)
    (slotutil 8 8  1.00)
    (bit   0 I78 )
    (bit   1 I79 )
    (bit   2 I80 )
    (bit   3 I81 )
    (bit   4 I82 )
    (bit   5 I83 )
    (bit   6 I84 )
    (bit   7 I85 )
)
(placement Z3
    (size 8 3)
    (areautil 1790 2120  0.84)
    (slotutil 24 24  1.00)
    (bit   0 I86  I98 I108 )
    (bit   1 I87  I99 I109 )
    (bit   2 I88 I100 I110 )
    (bit   3 I89 I101 I111 )
    (bit   4 I91 I102 I112 )
    (bit   5 I93 I103 I113 )
    (bit   6 I95 I105 I114 )
    (bit   7 I97 I107 I116 )
)
(placement Z_1_
    (size 8 10)
    (areautil 5925 7400  0.80)
    (slotutil 67 80  0.84)
```

```
    (bit   0 I117  --- I133 I134 I149  --- I167  ---  --- I195 )
    (bit   1 I118 I119 I135 I136 I151  --- I169  ---  --- I196 )
    (bit   2 I120 I121 I137 I138 I153 I154 I171  ---  --- I197 )
    (bit   3 I122 I123 I139 I140 I155 I156 I173  ---  --- I198 )
    (bit   4 I124 I125 I141 I142 I157 I158 I175 I176 I185 I199 )
    (bit   5 I126 I127 I143 I144 I159 I160 I177 I178 I187 I200 )
    (bit   6 I128 I129 I145 I146 I161 I162 I179 I180 I189 I201 )
    (bit   7 I130 I131  ---  --- I163 I164 I181 I182 I191 I202 )
)
```

8 Advanced Topics

This chapter provides a behind-the-scenes look at synthesis in Module Compiler and describes some advanced design techniques. The level of detail provided is related to the complexity of the particular synthesis function. As a novice, you can choose to ignore the information contained here. As you become more expert, you can use this information to get the most out of Module Compiler.

Chapter 8 discusses the following topics:

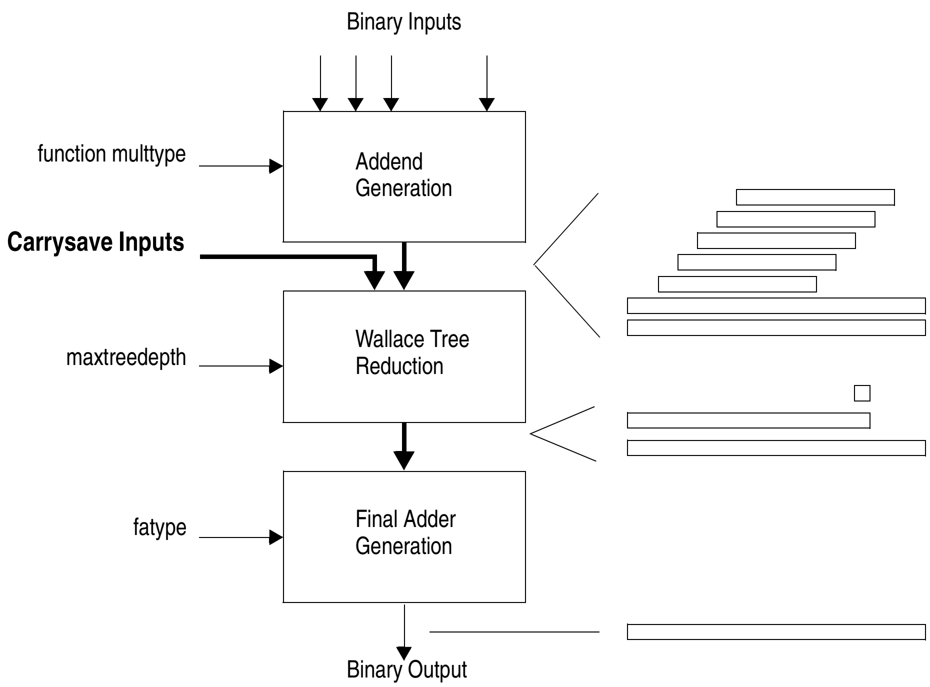
- Arithmetic computation
- Logical operators

Arithmetic Computation

Of all built-in functions, the integer arithmetic functions are the most complex and often most greatly influence the performance and area of the circuit. Addition, subtraction and multiplication are treated together, since all three use addition as the base function.

The processes involved with addition are shown below. The figures on the right show an example of the bit patterns that might exist at each stage of the process for the case of a 10×5 multiplication being summed with a wider signal. The carrysave bit format is shown for the case in which the **carrysave** attribute has be set to **optimize**.

Figure 8-1 Addition Architecture



After the addend generation is performed, a potentially large queue of bits is formed. The two carrysave inputs contribute the two wide sets of bits, while the multiplication contributes the parallelogram-shaped set of bits. After the Wallace tree reduction, which included a partial carry propagate reduction, there are two sections in the bit queue. The one to the right has only 1 bit per bit position and needs no further processing. The section to the left, beginning with the bit position that contains three bits, must be processed by a **carry propagate** adder. The final adder generator creates an output that has only 1 bit per bit position.

Sign Extension

To prevent excessive use of hardware and to improve performance, sign extension is performed using a well-known technique in which addition by a constant is substituted for replicating the sign bit:

```
        s s s s s s s s s s b b b b b b b b
->      1 1 1 1 1 1 1 1 1 1
        +                 s̄ b b b b b b b b
```

The conversion above results in the substitution of constants for most of the variable sign bits. The only drawback is that the sign bit must be inverted and that in the position of the original MSB there are now two bits; this is usually not a problem. To convince yourself that this technique works, you only need to look at two cases: s=0 and s=1. If s=0, then $\bar{s}$=1 and we get the situation below:

```
        s s s s s s s s s s b b b b b b b b

->      1 1 1 1 1 1 1 1 1 1
        +                 1 b b b b b b b b
        -----------------------------------
      1 0 0 0 0 0 0 0 0 0 0 b b b b b b b b
```

Note that the answer is the correct sign extended result, ignoring the carry out, which is discarded (however, when dealing with carrysave formats, one needs to worry about the carry out). When s=1, then $\bar{s}$=0 and you can see that the scheme also works:

```
        s s s s s s s s s s b b b b b b b b


->      1 1 1 1 1 1 1 1 1 1
        +                 0 b b b b b b b b
        -----------------------------------
        1 1 1 1 1 1 1 1 1 1 b b b b b b b b
```

The real advantage of this technique comes when many addends must be sign extended and summed. The constants can be added in advance, resulting in no additional sign extension hardware.

This scheme has potential problems for a few simple cases, such as that shown below. In this case, two signed operands are summed which have different widths.

```
              s s s s s b b b b b b b b b b b b
            + S S S S S S S S B B B B B B B B B B
              ---------------------------------

->            1 1 1 1 1
                      s̄ b b b b b b b b b b b b
              1 1 1 1 1 1 1 1
                            s̄ B B B B B B B B B B
              ---------------------------------

->            1 1 1 1 0 1 1 1
                      s̄ b b b b b b b b b b b b
                            s̄ B B B B B B B B B B
              ---------------------------------
                           11              2 1 0
```

The problem should be clear. The original solution with no fancy tricks requires a simple two input adder. After applying the sign extension trick, we have a problem in bit position 10 where three items must be added including an inverted signal. Not only is this solution slower, it is also likely to be larger. To handle this problem, all addition-based functions have an option to use simple sign extension. MC does not perform any extension when the sign bit of an addend is aligned with the sign bit of the result.

Another potential inefficiency exists when the output bit range is wider than needed. The internal sign extension works properly; however, the final adder depth and width increase to propagate the carries to the sign extension bits, resulting in a larger, slower circuit. It is, in general, much more efficient to compute only as many bits as needed and to perform the sign extension after the addition-based operation. This is a choice that you, as the designer, make manually.

Addition and Subtraction

Addition and subtraction result in simple addend generation. For addition, the addend generated for summation in the Wallace tree is formed by sign extending the input operand as discussed previously. Addends are generated for subtracted operands by inverting, sign extending, and adding a constant 1 to the input operand.

Multiplication

Multiplication affects only the first part of the addition operation, the generation of addends. Each multiplication architecture generates the addends in a slightly different way. There are currently four multiplication architectures that are implemented with addition: a simple non-Booth encoded multiplier, a Booth-encoded multiplier, a sign multiplier, and a multiplier architecture optimized for squaring. All multipliers adjust automatically to any combination of formats—signed or unsigned—at the two inputs. The product can also be shifted to the left with respect to the LSB of the result. For all multipliers, the first input is *X* and the second input is *Y*.

Non-Booth Multipliers

Non-Booth encoded multipliers generate addends using only simple logic: inverters for buffering, NOR gates for the basic partial product generators, and OR gates for the sign bits of the partial product generators. This type of multiplier generates N partial products of M bits each where N is the width of the Y input and M is the width of the X input. The non-Booth multiplier is relatively efficient when N and M are small numbers.

Booth-Encoded Multipliers

Booth-encoded multipliers use special library cells to encode the Y inputs and to generate the partial products. The number and width of the partial products are summarized below.

Table 8-1 Partial Products of Booth-Encoded Multipliers

Y Input with Width N	Num PP
Signed-even N	N/2
Signed-odd N	(N+1)/2
Unsigned-even N	N/2+1*
Unsigned-odd N	(N+1)/2*
* one partial product is simple (NOR gate based)	

X Input with Width M	Width PP
Signed	M+1
Unsigned	M+2

Booth multipliers are most efficient for signed *X* and *Y* and, in particular, signed and even *Y*. These multipliers are not as efficient for narrow and/or unsigned *X* or *Y*. Booth-encoded multipliers provide one additional trick for free: the product X*(Y+Z) can be computed at no additional cost if *Z* is a single unsigned bit (this operation is available via the *multp*() function). By default *Z* is zero, but a nonzero operand can be specified. The offset can be used to advantage in a couple of ways. First, *–XY* can be computed as X*(~Y+1). Second, it can be used to generate a "true 1" coefficient to the multiplier, by setting *Z* to 1 and *Y* to a full scale positive number.

Signed Multipliers

Signed multipliers are used to multiply an operand (*X*) of any format and width by plus or minus 1 (the sign of *Y*). Only the sign bit of the *Y* input is used; if *Y* is negative the result is *–X*; otherwise it is *+X*. If *Y* is unsigned, MC issues a warning.

Constant Multipliers

Multiplication of a constant by a variable operand deserves some special mention, even though no special syntax is required. The constant operand is used to generate a set of addends that are scaled versions (positive, negative, and shifted) of the variable operand. The constant is optimized to minimize the total number of addends generated in a manner similar to, yet more efficient than, Booth encoding. This type of operation is affected by **fatype** but not by **multtype**.

Squaring Circuits

Expressions of the form X*X result in a special multiplier type that is smaller (usually 40 to 50 percent) and faster than a normal multiplier. **multtype** has no effect on squaring circuits, but **fatype** works as for other multiplier types.

Rounding

Two types of rounding are available in MC. Simple rounding is easy to use. Internal rounding requires a little more attention, but provides useful tradeoffs in some situations.

Simple Rounding

Simple biased rounding adds a constant 1 in the bit position below the final LSB. When this option is used, all LSBs below the one specified by the value of the **round** attribute are erroneous and must be ignored. There are techniques, however, that can make this biased rounding operation unbiased by examining the normally discarded bits and adjusting the result.

Internal Rounding

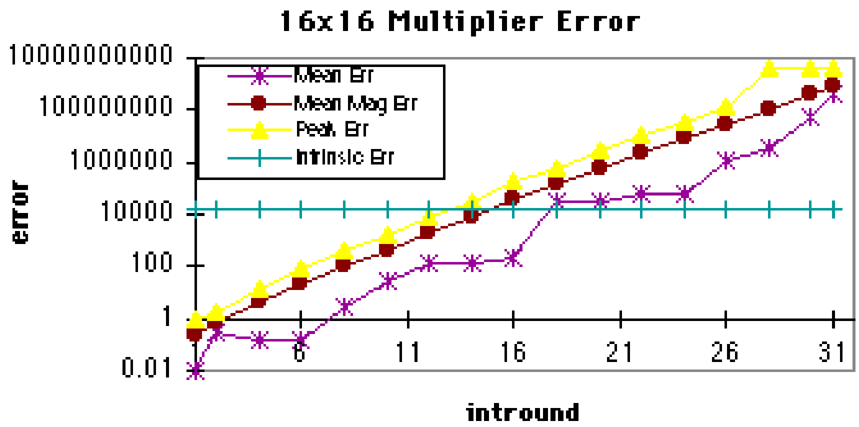
Use the **intround** attribute to make tradeoffs between area and precision in arithmetic expressions. By default, the value of this attribute is 0. Increasing the value of **intround** increases the number of bits that are discarded from each addend in the arithmetic expression. Because the addends are rounded before being summed, savings in area occur. There are also some small performance improvements. MC outputs the correct behavioral and gate level netlist for each value of **intround**. These files can be used to verify system performance.

Internal rounding is used only in DSP and other applications in which the inputs or computation results have already been rounded or truncated. For these cases, the results are never perfectly accurate. If the inputs are considered exact and you require an exact output, do not use internal rounding. Unless a very large value is used for **intround**, this technique introduces only very small biases, as required by many recursive algorithms.

Example Assume that the X and Y bits of a 16×16 multiplier have been rounded to 16 bits before multiplication. The multiplier has an inherent error, shown by the horizontal line in Figure 8-2. The horizontal line also represents the error generated by rounding the output to 16 bits. The errors due to internal rounding become appreciable when intround is approximately 14. Below that value, the error in the output is dominated by the error incurred by rounding the inputs, not by the internal rounding. Note that the mean error is much smaller than the mean magnitude error.

Notice that the area decreases as the amount of internal rounding increases. In fact, 25% of the area can be saved at the point where internal rounding errors approach the intrinsic errors. Performance improvements are insignificant unless intround is greater than 16, which indicates that more than half of the multiplier has been removed.

Figure 8-2 The Effect of the **intround** Attribute on Multiplier Error



The Wallace Tree Reduction

After all of the addends have been generated with the constructs described previously, the Wallace tree algorithm is used to reduce the number of signals to a maximum of two or three per bit position. (In fact, MC automatically determines when three signals are allowed in a given bit position without degradation of the timing of the final adder.) A final carry-propagate adder is used to generate a binary result. This reduction happens automatically and is very efficient; it does not result in any hardware if none is needed.

The **maxtreedepth** attribute is used to limit the depth, or scope, of the Wallace trees. In general, large Wallace trees are used to increase performance. However, Wallace trees are global structures by nature, and at some point, utilization suffers if the design includes very large trees. The proper use of this directive allows you to effectively create a serial connection of Wallace trees without changing the network description.

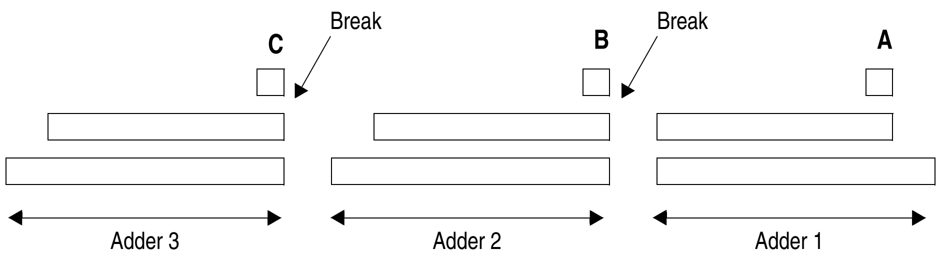
It works by allowing only the number of signals in each bit position of the Wallace tree queue to reach the value given by the attribute. When this number is reached, a Wallace tree reduction is performed. For example, suppose you want to build a sum of products with 64 8×8 products. Assuming the use of a non-Booth encoded multiplier, the middle bit positions would contain 64*8=512 bits! This will surely result in poor utilization. Setting **maxtreedepth** to 32 causes a loss of performance, but significantly improves utilization. Using a value up of 32 seems to provide no utilization degradation.

By default this attribute is set to a very large number.

Carry Propagate Adder Optimization

MC automatically breaks the carry propagate adder into multiple adders if possible and allows the greatest number of signals in each bit position. This optimization makes it possible to have three bits in the lowest bit of the adder without a significant area or performance penalty. In general, MC determines which bit positions can have a carry input. If no carry input from a preceding stage is possible, the adder is broken at that point and three bits are allowed in the next bit position. For example, consider the sum of signals shown below.

Figure 8-3 Carry Propagate Adder Optimization



This complex example involving the sum of six different operands illustrates several interesting concepts. Because of the breaks between the operand groups, the problem can be solved with three small adders. The three adders operate in parallel and are smaller and faster than a single large adder. In addition, three bits are allowed at points A, B, and C without invoking a Wallace tree reduction, even though point A is not the first bit of the stage. Note that the bits to the right of A are not input to any carry propagate adder. Less sophisticated approaches might solve this problem with a Wallace tree reduction because of the bits at C and B (and perhaps even at A), and generate a single carry propagate adder.

The Carry Propagate Adders

In most cases, you need the result to be binary, and must use a final carry propagate adder.

Five different architectures are supported through the **fatype** attribute, each with its own advantages and disadvantages, which are summarized below.

Figure 8-4 **fatype** Attributes

fatype	Description	Area	Delay	Use Arrival Times	Use Desired Delay	When Default
csa	carry-select	O(n)	O$\sqrt{n}$	Yes	Yes	Never
cla	carry-lookahead	O(n)	O(2 $\log_2(n)$)	No	No	pipeline=on
fastcla	fast-carry-lookahead	O($n \log_2(n)$)	O($\log_2(n)$)	No	No	pipeline=off opt for speed
clsa	carry-lookahead-select	Variable ripple-> fastcla	Variable ripple-> fastcla	Yes	Yes	pipeline=off not opt for speed
ripple	ripple	O(n)	O(n)	No	No	opt for size

The selection of these architectures is not automated. However, MC has good defaults for different circumstances. In critical situations it can pay off to manually try a different architecture.

The **csa** adder is not a particularly high performance adder, ideally achieving only O$\sqrt{n}$ delay. In reality, the growing loading on the carry select lines degrades performance below the expected level.

When pipelining is enabled, an attempt is made to break the **csa** adder into stages that fall into different pipeline sections. This is in contrast to allowing pipelining inside a stage. This addition to the algorithm often provides an advantage when there are large delay skews, as in a multiplier.

The **clsa** adder is a good general choice, especially with large delay skews, but it does not pipeline well. It is by far the most flexible architecture and automatically creates a structure ranging from a **ripple** to **fastcla** adder, depending on the desired delay.

The **fastcla** adder is usually the fastest architecture, but it is also the largest. It uses a dense carry tree to propagate the carries to each bit in only $\log_2(n)$ inverting **AND-OR** delays. Besides the carry tree, an **XOR** delay occurs in the sum generation, while one **NAND** delay occurs in the initial G and P generation. The fanouts on the drivers in the carry tree are constant, yet the actual routing complexity grows with the number of bits. This structure is very balanced and tends to improve only minimally during logic optimization.

The **cla** adder uses a sparse carry tree that roughly doubles the delay—actually $2*(\log_2(n)–1)$—in the carry tree relative to the **fastcla** adder, but provides significant area savings. Because the tree is sparse, there is a great deal of slack on many of the nets, making logic optimization very successful for this structure.

The **ripple** adder is the smallest and slowest adder structure and is useful in noncritical portions of the design.

Carry/Save Operands

The carry propagate adders cost a great deal in terms of delay and area. In some cases, it is essential to avoid them. For these cases, it is possible to bypass the final adder and leave the output in carry/save format.

Note: Carry/save operands cannot be modeled behaviorally until they have been added to another operand and then only if no significant bits have been lost through bit ranging or other nonlinear operators. That is, they are modeled just like normal binary signals.

Three varieties of carry/save signals can be selected by changing the **carry/save** attribute as summarized below.

Table 8-2 **carry/save** Modes

carry/save	Constants	Maxbits	Ripple Add	MC Language Use
off	merged	1	No	need binary result
on	not merged	3	No	when summed with a carrysave signal
optimize	merged	3	Yes	when summed with a critical noncarrysave signal
convert	merged	2	No	when input to convert

When the **carry/save** attribute is set to **on**, the resulting signal does not have constants merged with variables because it is expected to be summed with another carry/save signal with unmerged constants. Merging the constants early hurts performance and area. It allows up to three bits (actually three signals and one constant) in each bit position. If only two bits were allowed, half-adders would have to be used which are very inefficient (they convert two input signals into two output signals resulting in virtually no reduction). Half adders are assumed to be used only immediately before the final addition.

The **optimize** carry/save signal has fewer total number of bits which must be summed with a non-carry/save signal; the assumption is that the other inputs to the sum are more critical and should not be slowed further. Constants are merged and a ripple addition is performed on the LSBs to remove as many bits as possible without increasing the delay.

The **convert** carry/save signal is the traditional carry/save signal and is required when converting a carry/save operand to two signed operands. It has no more than two bits in any bit position.

A carry/save signal can be used in only a few circumstances. First, it can be added, subtracted, or compared (>, >=, <. <=) with any operand and optionally shifted by a constant. Second, it can be input to *sreg*(), *preg*() or any *eqreg*(). Finally, it can be input to *hidelat*() or *convert*(), which converts the carry/save operand to two signed operands. This option should be used with extreme care.

Due to limitations of current implementation, the carry/save signals should be declared with a bit width. However, during synthesis, the true bit range is determined automatically and that provided by the user is ignored. You should write all code using actual bit ranges, even for the carry/save signals. Then the **carry/save** attribute can be toggled to try both carry/save and binary implementations without any other code changes.

Note that the assignment operator alone always converts a carry/save signal to binary, regardless of the setting of the **carry/save** attribute. To have the assignment produce a carry/save signal, use the + operator as shown below.

```
directive (carrysave = "on");
Z2 = A+B;                          Z2 is a carry/save
Z3=+Z2;                            Z3 is a carry/save
Z4=Z3;                             Z4 is not a carry/save
```

In Example 8-1, a simple 32×32 multiplication is broken into four pieces that are kept in carry/save format. The four carry/save signals are summed to produce the final 64-bit product.

Example 8-1 Example of **carry/save** Usage

```
module mult32 (Z,X,Y);

     input [31:0] X;
     input [31:0] Y;
     output [63:0] Z;

// no final adders for Z0,Z1,Z2,Z3
     directive(carrysave="on");
     wire [0:0] Z0,Z1,Z2,Z3;
     Z0=Y[7:0]*X;
     Z1=Y[15:8]*X;
     Z2=Y[23:16]*X;
     Z3=Y[31:24]*X;

     directive(carrysave="off");
     Z=Z0+(Z1<<8)+(Z2<<16)+(Z3<<24);  Z must have final adder

endmodule
```

The following example shows the case in which a carry/save accumulator is used. In this case, convert is required to allow the feedback of the carry/save signal.

Example 8-2 Example of a **carrysave** Accumulator

```
module acc(Z,X,RESET);
     input signed [7:0] X;
     output [7:0] Z;
     input [0:0] RESET;
     wire signed [7:0] ACC0,ACC1,X1,XPR,ZA,RZA0,RZA1;
     wire ZA0,ZA1;
     wire [9:0] ACC;
     wire [9:0] ZA0R,ZA1R;

     ACC0=sreg(RZA0);                       need two sreg's for carrysave
     ACC1=sreg(RZA1);

     directive(carrysave="convert");        must use convert option here
     ZA=X+ACC0+ACC1;

     convert(ZA0, ZA1, ZA);                 generate two signed signals, ZA0, ZA1

     directive(MUXtype="andor");            now we can MUX the carrysave signal
     RZA0=RESET ? ZA0 : 0;                  to allow the loop to be reset
     RZA1=RESET ? ZA1 : 0;

     directive(carrysave="off",fatype="clsa");
     ACC=ACC0+ACC1;                         true binary to be used elsewhere
     Z=ACC[7:0];
endmodule
```

AND, OR and XOR

Overview

Each of these functions computes a bitwise logical function over the inputs. As with the addition-based functions, any number of inputs can be accommodated and degenerate cases are handled efficiently. It should be noted that missing bits are treated as zero. For **OR** and **XOR**, there should be no confusion, as the zeros do not change the result. For **AND**, however, a zero in any bit position causes the result for that bit position to be zero.

MC directly supports the inversion of any input (including the missing bits that are inverted to ones). **NAND** and **NOR** can be implemented by inverting all the inputs and using the complementary function. Bit ranging, shifting and the selection of the output operand format are provided in the same manner as in the addition based functions. Sign extension of inputs is accomplished in the direct manner.

There are only two stages in the generation of the result: gathering of the signals and the Wallace tree reduction. Because there is no interaction between bits, the Wallace tree algorithm is used to reduce the inputs down to the final binary result. It has been modified slightly to allow both true and inverted bits in the Wallace tree queue to increase the use of inverting logic that is generally faster and smaller than noninverting logic.

Optimization

Each function can be optimized for either speed or area. There is no direct control over this except through the current optimization criterion: the circuit is optimized for speed unless the delay goal is set very large, in which case these functions are optimized for area. The **AND** and **OR** operations are particularly sensitive to the optimization style because of the wide range of cells available (for example, 2 to 8 inputs).

Analysis and Optimization

9

This chapter describes Module Compiler’s various output files and how to use them to interpret your results and plan future design modifications.

Chapter 9 discusses the following topics:

- 5
- output files
- Object naming
- Verilog simulation
- How to call Design Compiler from MC
- Debugging a design built by MC
- Optimizing a design built by MC

Module Compiler Output Files

MC reports its results in a group of files. The generation of these files is controlled through various options available in the MC input language, the GUI, and the command line interface.

It is also possible to call Design Compiler directly from MC. This option is provided to allow you to take advantage of Design Compiler's additional capabilities. For example, when complex Boolean logic is on the critical path, Design Compiler can provide significant performance advantages.

The output files produced by MC are summarized in Table 9-1. The root name of the output is generally the same as the name of the module being synthesized. You can use the **modname** directive to change the root name. See the "Naming" section in this chapter for more information on the significance of names in MC. The generation of individual files can be enabled or disabled by using the options listed below.

Table 9-1 MC Output Files

File	Default File Name	Command Line Option	Contains
log	-	-l <*name*>	Runtime status of MC (- for no file)
Design Report	<*module*>.report	-r +|-	Design, group, operand, and cell summary
Behavioral Model	<*module*>.bvrl	-b +|-	Behavioral simulation model without timing
Verilog Structural Model	<*module*>.vrl	-v +|-	Verilog gate-level simulation model
EDIF Structural Model	<*module*>.edif	-e +|-	EDIF structural netlist
Table	table	-t <*name*>	Running summary of design statistics
Design Compiler Report	<*module*>.dc.rep	NA	Design Compiler report file
Design Compiler output netlist	<*module*>.dc.vrl	NA	Verilog netlist generated by Design Compiler
Library Report	<*technology*>.rep	NA	Summary of vendor's technology library
Layout Information	<*module*>.layout	NA	Relative placement of instances within the design

The Log File

The log file contains the runtime status of MC. It contains the progress report, warnings and errors. Extended messages can be obtained in a verbose mode by using the command line option, **-m verbose** or a more terse output is obtained with **-m normal**. Informational messages are only output in verbose mode. The log file can be sent to standard output when not in graphical mode by setting the name of the file to **-** (minus sign); otherwise it is sent to the file with the name provided. If a file name is provided, the file open mode is specified by the option, **-logmode**; common values are **w** to start a new file or **a** to append to the existing file.

The synthesis status is reported in one of two ways depending on the setting of the **verbose** switch. In Verbose mode, all operands except shift registers generate one line summaries as they are synthesized. In addition, the code from the input file is displayed as it is processed. The summary shows area and timing values. In Normal mode, each operand produces one “.”.

In Verbose mode, a summary of the design and of each group is output before final logic optimization. The summary provides the name of each group or design, the number of instances, the number of flip-flops, the total area, the maximum final delay (delay at the outputs for the design or the last pipeline stage for groups), the largest internal delay, and the latency. For CBA libraries, the compute-to-drive ratio is also given.

Any overloaded nets found before optimization are also provided in the information messages if Verbose mode is selected. If the overloaded nets appear on critical paths, you can use this information to try to correct the problems during synthesis rather than letting the optimizer correct the problems.

During optimization, the log contains a progress report indicating the current critical path delay (delay the net end point with the least slack), the slack, the number of instances, and the area. For CBA libraries, the compute-to-drive ratio is included. The optimization step being performed is identified, followed by the number of instances changed in the step, and the net change in the number of instances and sections. Finally, the group containing the critical path is identified. When there are multiple timing groups, it is important to observe the slack rather than the delay, because the delay values may not be comparable. Example 9-7 shows the optimization log for a fairly complex design.

After optimization, a final group and design summary is provided in the same format as the preoptimization report. Note that the internal delay is the delay for the net within the group or design that has the minimum slack. This is slightly different than the data provided during optimization which only shows path end points.Example 9-1 shows the design summary for a complex design with a CBA library.

Example 9-1 Design Summary for a Complex Design

```
              GROUP    TIMING (ns)   POWER           AREA          LATENCY
               name  final internal  (W)    ff  inst   sect   c/d   cycles
           -------------------------------------------------------------------
         Diagnostic   10.9    10.9   0.00    0    10     30   Inf      0
              Error   11.4    11.6   0.03   42   334    822   1.8      1
          FB_Filter   11.7    11.7   0.61   22    43   8034   2.9      0
          FB_Update    8.8    11.6   0.02  120   262    794   2.8      2
          FF_Filter   11.6    11.7   0.10  276   974   3771   5.0      2
          FF_Update   11.7    11.7   0.07  133   933   2116   2.5      2
               Gain   22.4    22.4   0.01   17   130    424   4.8      0
               LFSR    8.1     8.1   0.00   24    44    179   3.1      0
                PLL   11.7    11.7   0.04   94   511   1287   2.2      2
             Slicer    8.0     9.3   0.00    1    59     92   0.6      0
               Sync   10.4    11.1   0.02   90   218    658   2.4      0
             Timing   11.5    11.7   0.02   40   240    587   3.0      2
               misc    2.0     2.0   0.00    0    13     25   0.8      0

       DESIGN    TIMING (ns)   POWER           AREA          LATENCY
         name  final internal  (W)    ff  inst   sect   c/d   cycles
     --------------------------------------------------------------------
          dfe   10.9    11.7   0.92  859  3771  18819   3.0     2
```

Finally the design critical path is reported. The beginning of the path is reported at the bottom, the end of the path at the top. The following is an example for another design:

```
Critical Path Summary ...
     Path Ends at: Z_1_[62] Z[62]
     Endpoint is in group: misc, slack:  -8.736, delay goal:   0.001
          delta delay rise  fall load gload pins
     setup:0.008.748.65 8.74
     /Z_1_[62]/I2323/EN3P(C->Z)/N2450:0.608.748.658.74 4.3 3.02
     /Z_1_[62]/I2255/AO6P(A->Z)/N2382:0.348.148.137.62 2.6 1.32
     /Z_1_[30]/I2125/AO7P(A->Z)/N2252:0.597.807.307.80 6.4 3.93
     /Z_1_[14]/I1963/AO6P(A->Z)/N2090:0.857.207.216.8612.810.33
     /Z_1_[6]/I1817/AO7P(C->Z)/N1944:0.656.366.126.3612.810.33
     /Z_1_[6]/I1687/AO6P(C->Z)/N1814:0.505.715.715.46 6.4 3.93
     /Z_1_[6]/I1557/AO7P(B->Z)/N1684:0.515.214.815.21 6.4 3.93
     /Z_1_[5]/I1428/NR2P(A->Z)/N1555:0.554.704.704.41 9.4 5.74
     /Z_1_[5]/I1299/FA1AP(CI->S)/N1309:1.024.164.064.16 8.6 4.94
     /Shift[5]/I1229/MUX21LP(A->Z)/N1229:0.223.143.143.11 2.6 1.32
     /Shift[6]/I1162/MUX21LP(B->Z)/N1162:0.412.922.852.9210.9 8.43
     /Shift[6]/I1096/MUX21LP(A->Z)/N1096:0.312.512.512.5310.9 8.43
     /Shift[10]/I1026/MUX21LP(A->Z)/N1026:0.402.202.132.2010.9 8.43
     /Shift[18]/I968/MUX21LP(A->Z)/N968:0.391.801.801.8210.9 8.43
     /Shift[34]/I889/ND2P(B->Z)/N889:0.441.401.351.4010.9 8.43
     /Shift_out__1[34]/I821/FD1QP(CP->Q)/N821:0.970.970.971.00 8.7 6.23
     /CLK:0.00  0.00  0.00  0.00  684.9  288.0  321

Format is for names:

<group>/<signal>[bit]/<inst>/<cell>/(<in pin>-><out pin>)/<net>

<group> is only printed if more than one group
<inst>, <net>, <signal> follow appropriate naming conventions
```

The meaning of the columns is given in Table 9-2.

Table 9-2 Columns in the Design Critical Path Report

Column Name	Meaning
delta	The change in the critical path delay to this net
delay	The delay of the critical path at this net
rise	The rise delay at this net
fall	The fall delay at this net
load	The total load on the net (gload plus wire load)
gload	The total gate input loading on the net
pins	The total number of pins on the net

The Design Report File

The Design Report file contains area and timing summaries, critical paths and slack histograms for each group and the design. The summaries provides many statistics that should be familiar. The slack histogram provides an indication of the relative number of path end points which exist within the group or design with each value of slack. It is possible that a group may not contain any path endpoints even though it does contain instances.

For CBA libraries, the report also provides the Maximum Utilization, which reflects the theoretical maximum utilization that can be obtained in a 100% floor plan based on the ratio of used compute to used drive sections. That is, it is impossible to achieve a utilization higher than this value; the maximum utilization is not the expected utilization. This number should be kept within the expected utilization bounds (greater than 75%) to prevent poor place and route results. For small circuits, this number is not particularly important as the final compute to drive ratio and utilization is determined by the bulk of the circuit.

An extensive summary of the I/Os is provided. For each bit of each input, the load, fanout, arrival time (delay) and the relative slack are provided. This information should make asymmetries between input operands and bits of a single operand more clear. For each bit of the outputs, some additional information is provided: the latency at the output and the start point of the critical path ending at the output. The absolute slack is reported for outputs.

Absolute slack is the difference between the delay goal and the actual delay. Relative slack indicates the amount by which the delay at a point can be increased without violating the delay goal, or, if the delay goal has not been met, without increasing the critical path length. Relative slack is influenced by delay equalization when the delay goal is not met as summarized below.

Table 9-3 The Influence of Delay Equalization on Relative Slack

Delay Equalization	Critical Path Used in Relative Slack
None	Most critical path at the point of interest
Local	Most critical path in the same group as the point of interest
Global	Most critical path in the same timing group as the point of interest

Consider the simple example below. Note that there are three groups with the same delay goal, so all three groups belong to the same timing group.

```
module foo (A,B,C,D,E,F,G,H);
     input [0:0] A,B,C,G;
     output [0:0] D,E,F,H;

     directive (logopt="off");
     directive (group="G1",delay=1000);
          D=isolate(A);
     directive (group="G2",delay=1000);
          E=isolate(isolate(B));
     directive (group="G3",delay=1000);
          F=isolate(isolate(isolate(C)));
          H=isolate(isolate(isolate(isolate(G))));
endmodule
```

The relative slacks at *A*, *B*, *C* and *G* are shown below for different delay equalization cases. Consider the case of no equalization first. Only the path from *A* meets the delay goal, so only A shows positive relative slack. The other inputs show zero slack because these inputs cannot be delayed further without increasing the critical path lengths from these points. With local equalization, *C* now shows positive relative slack, indicating that it can be delayed further without increasing the critical path in group *G3* which starts at input *G*. With global equalization, all inputs except *D* have positive relative slack because *A*, *B* and *C* can be delayed without increasing the critical path length in the timing group which starts at *G*.

Table 9-4 Examples of the Effect of Delay Equalization on Relative Slack

	Delay Equalization		
Input	**None**	**Local**	**Global**
A	.24	.24	1.54
B	0	0	1.03
C	0	.52	.52
G	0	0	0

Operand summaries are provided for all user-defined and automatically created temporary operands. The bit range and the format are provided in addition to area and timing information which is similar to that provided for groups and the design. The value of any constant operands (user-defined or computed) are provided in this summary.

The user-defined critical paths, if any, are listed in either short or long form as determined by the mode set in the input description. The long form is similar to the other critical path generated as part of the group and design summaries. The short form is suitable for datasheets in which the length of the path rather than the path itself is important.

A cell-use summary is also provided, which indicates the number of instances and the percentage of total instances for each cell. The area of the cell is followed by the area occupied by all instances of the cell and the percentage of the total area occupied by that cell. The cell summary is divided into three sections: the cell's usage by type of cell (I/O, combinatorial, flip-flop, or RAM), the cells sorted in order of decreasing area occupied, and the cells sorted in alphanumeric order.

The following is an example of a Design Report file for a simple 8-bit adder:

Example 9-2 Sample Design Report File

```
Synopsys Module Compiler Report
MC Version: 1.0
Input File: /src/dp/lib/dp//dplite//Adder.comp.dpa
Module Name: adder8
Parameters: Name=adder8,Width=8,AdderType=fastcla,CarryIn=0,CarryOut=0
Date: Thu Nov 21 16:20:10 1996


Options
     Technology Lib Dir: /src/dp/lib/tech/
     Technology: lca500k
     Operating Condition: WCCOM
     Operating Temperature: 70
     Operating Voltage:  3.13
     Wireload Model: B5X5

     Optimization Criterion: speed

     Logic Optimization Steps (-1)
          Synthesis Logic Min: on
          Gate Eater: on
          Rule: on
          Reorder: on
          Logic Min 1: on
          Logic Min 2: on
          Logic Min 3: on
          Logic Min 4: on
          Logic Min 5: on
          Timing: on
          Area/Power: on
          Synthesis Min Slack: on
          Compute/Drive: on
     Local Opt Iterations: 4
     Global Opt Iterations: 2
     Equalization Passes: 1
     Pipelining Margin: 0

     Clock Frequency: 40

     Default Input Max Load: 400
     Default Output Load: 30
     Default Operand Format: unsigned
     Top Level Mode: off
     Behavioral Model File: ./adder8.bvrl
     Logic Model File: ./adder8.vrl

Summary

                     DESIGN   TIMING (ns)   POWER         AREA        LATENCY
                       name  final internal  (W)    ff  inst   area    cycles
-------------------------------------------------------------------------------
                     adder8    2.2     2.2   0.00    0    46    108      0
```

```
Design: adder8
--------------------------------------------------------------------------------
               Number of instances:       46
                      Number of ff:        0
                    Number of nets:       62
                    Number of pins:      156
                     pin/net ratio:      2.5
                              Area:      108
           Longest final path (nS):     2.16
        Longest internal path (nS):     2.16
                           Latency:        0
                         Power (W):    0.001
--------------------------------------------------------------------------------

     Critical Path Summary ...
     Path Ends at: Z1[7] Z_1_[7] Z[7]
     Endpoint is in group: misc, slack:  -2.155, delay goal:   0.001
                                   delta  delay   rise   fall   load  gload   pins
                         setup:    0.00   2.16   2.16   2.16
   /Z1[7]/I65/EO3P(C->Z)/N65:    0.48   2.16   2.16   2.16    4.3    3.0     2
   /Z1[6]/I54/AO7P(A->Z)/N99:    0.45   1.67   1.49   1.68    3.3    2.0     2
    /Z1[2]/I38/AO6(B->Z)/N81:    0.36   1.23   1.23   1.18    5.7    3.2     3
    /Z1[2]/I23/NR2(A->Z)/N73:    0.42   0.87   0.70   0.86    4.4    1.9     3
     /Z1[2]/I7/NR2P(B->Z)/N7:    0.44   0.44   0.44   0.19    7.4    3.7     4
                           /Y[2]:    0.00   0.00   0.00   0.00    9.0    5.3     4
--------------------------------------------------------------------------------

     Slack Histogram ...
     slack   %   num
     -2.25  62     5:*****
     -2.00  75     1:*
     -1.75  88     1:*
     -1.50  88     0:
     -1.25  88     0:
     -1.00  88     0:
     -0.75 100     1:*
     -0.50 100     0:
     -0.25 100     0:
      0.00 100     0:
      0.25 100     0:
      0.50 100     0:
      0.75 100     0:
      1.00 100     0:
      1.25 100     0:
      1.50 100     0:
      1.75 100     0:
      2.00 100     0:
      2.25 100     0:
      2.50 100     0:
```

```
Input Summary
-----------------------------------------------------------------------------
    CLK[0:0] (signed)
          Bit   Load    Fanout    Delay  Relative Slack
            0    0.0       0       0.00   100000.00

    X[7:0] (unsigned)
          Bit   Load    Fanout    Delay  Relative Slack
            0    4.5       2       0.00      0.18
            1    7.6       3       0.00      0.12
            2    7.6       3       0.00      0.00
            3    7.6       3       0.00      0.05
            4    7.6       3       0.00      0.01
            5    7.6       3       0.00      0.21
            6    6.6       3       0.00      0.06
            7    2.4       1       0.00      1.26

    Y[7:0] (unsigned)
          Bit   Load    Fanout    Delay  Relative Slack
            0    5.5       2       0.00      0.18
            1    9.0       3       0.00      0.12
            2    9.0       3       0.00      0.00
            3    9.0       3       0.00      0.05
            4    9.0       3       0.00      0.01
            5    9.0       3       0.00      0.21
            6    8.0       3       0.00      0.06
            7    3.6       1       0.00      1.58


Output Summary
-----------------------------------------------------------------------------
    Z[7:0] (unsigned)
         Bit    Load    Int   Total  Slack  Latency Path
                       Delay  Delay                 Start
           0     4.3    0.64   0.64  -0.64    0      X[0]
           1     4.3    1.51   1.51  -1.51    0      Y[0]
           2     4.3    1.86   1.86  -1.85    0      Y[1]
           3     4.3    1.96   1.96  -1.96    0      Y[2]
           4     4.3    1.98   1.98  -1.98    0      Y[1]
           5     4.3    2.14   2.14  -2.14    0      Y[4]
           6     4.3    2.08   2.08  -2.08    0      Y[4]
           7     4.3    2.16   2.16  -2.15    0      Y[2]

Clock Pin Summary
-----------------------------------------------------------------------------
           Clock Pin    Fanout    Gate Load   Total Load
                 CLK       0         0.00        0.00


=============================================================================
```

```
Operand Summary

          USER CONSTANTS  BIT RANGE              FORMAT
-----------------------------------------------------------------------------
                 dpa_one      [0:0]                 0x1
                dpa_zero      [0:0]                 0x0

          INPUT OPERANDS  BIT RANGE              FORMAT
-----------------------------------------------------------------------------
                     CLK      [0:0]              signed
                       X      [7:0]            unsigned
                       Y      [7:0]            unsigned

         OUTPUT OPERANDS  BIT RANGE              FORMAT
-----------------------------------------------------------------------------
                       Z      [7:0]            unsigned

       COMPUTED OPERANDS  BIT RANGE              FORMAT
-----------------------------------------------------------------------------
                      Z1      [7:0]            unsigned
                    Z_1_      [7:0]            unsigned

         UNUSED OPERANDS  BIT RANGE              FORMAT
-----------------------------------------------------------------------------

          USER CONSTANTS   TIMING (ns)   POWER        AREA       LATENCY
                    name  final internal  (W)    ff  inst   area   cycles
------------------------------------------------------------------------
                 dpa_one   0.0     0.0   0.00    0     0      0     0
                dpa_zero   0.0     0.0   0.00    0     0      0     0

          INPUT OPERANDS   TIMING (ns)   POWER        AREA       LATENCY
                    name  final internal  (W)    ff  inst   area   cycles
------------------------------------------------------------------------
                     CLK   0.0     0.0   0.00    0     0      0     0
                       X   0.0     0.0   0.00    0     0      0     0
                       Y   0.0     0.0   0.00    0     0      0     0

         OUTPUT OPERANDS   TIMING (ns)   POWER        AREA       LATENCY
                    name  final internal  (W)    ff  inst   area   cycles
------------------------------------------------------------------------
                       Z   2.2     0.0   0.00    0     0      0     0

       COMPUTED OPERANDS   TIMING (ns)   POWER        AREA       LATENCY
                    name  final internal  (W)    ff  inst   area   cycles
------------------------------------------------------------------------
                      Z1   0.0     2.2   0.00    0    46    108     0
                    Z_1_   0.0     0.0   0.00    0     0      0     0

         UNUSED OPERANDS   TIMING (ns)   POWER        AREA       LATENCY
                    name  final internal  (W)    ff  inst   area   cycles
------------------------------------------------------------------------


=============================================================================
```

```
Cell Use Summary

     By Group:
     count (  %)      Group       total (  %)
        46 (100)       Comb         108 (100)

     Core Sorted by die area:
     count (  %)       cell area  total (  %)
         4 (  9)       EN3P    6     24 ( 22)
         3 (  7)       EO3P    6     18 ( 17)
         5 ( 11)       AO7P    3     15 ( 14)
         6 ( 13)       NR2P    2     12 ( 11)
        10 ( 22)        ND2    1     10 (  9)
         5 ( 11)        AO6    2     10 (  9)
         4 (  9)        AO7    2      8 (  7)
         5 ( 11)        NR2    1      5 (  5)
         1 (  2)         EO    3      3 (  3)
         2 (  4)         IV    1      2 (  2)
         1 (  2)        IVP    1      1 (  1)

     Core Sorted by name:
     count (  %)       cell area  total (  %)
         5 ( 11)        AO6    2     10 (  9)
         4 (  9)        AO7    2      8 (  7)
         5 ( 11)       AO7P    3     15 ( 14)
         4 (  9)       EN3P    6     24 ( 22)
         1 (  2)         EO    3      3 (  3)
         3 (  7)       EO3P    6     18 ( 17)
         2 (  4)         IV    1      2 (  2)
         1 (  2)        IVP    1      1 (  1)
        10 ( 22)        ND2    1     10 (  9)
         5 ( 11)        NR2    1      5 (  5)
         6 ( 13)       NR2P    2     12 ( 11)
```

The Verilog Behavioral File

A simulatable Verilog HDL behavioral model provides a way to quickly check the network description. There are no continuous time delays modeled, but all cycle delays, including those created by automatic pipelining, are modeled accurately. The behavioral model and gate-level netlist match on a cycle-by-cycle basis except for a few details.

The Verilog Netlist

The Verilog gate-level netlist matches the behavioral model on a cycle-by-cycle basis. This file can be used to simulate the design with pre- and post-layout delay annotation and to integrate the MC output with the rest of the design.

The EDIF Gate-Level Netlist File

This file is equivalent to the Verilog gate-level netlist, except that it utilizes EDIF syntax and the internal operands are not accessible. Instance names in the EDIF file match those in the Verilog file.

The Table File

The table file contains a running summary of all designs for quick comparison. Each design is given one line in the file which contains the design name, number of sections, critical path delay (ns for the net with the minimum slack) latency, power (W) and the parameters (if any). Following is an example of the table format.

```
dfe   18559    11.99      2    0.918  width=22
dfe   22805    12.67      2    0.976  width=26
dfe   22239    13.14      2    0.964  width=25
dfe   18559    11.99      2    0.918  width=22
dfe   18556    12.00      2    0.917  width=22
```

The GUI displays the last line (the most recent design) at the top of the window.

The Design Compiler Report and Netlist

These files are generated by Design Compiler when it completes. See the Design Compiler documentation for details about these files. In particular, Design Compiler may use a different measure of area than MC.

Naming

MC provides you with control over the naming of instances, nets, and wires. If you do not provide names, MC creates the names following certain guidelines. The following sections describe how to control names, and how MC creates names for objects when you do not provide them.

Note: The following sections make reference to the Sim Debug Mode option and the Use Group Names option. Use Group Names is accessed from the Synthesis menu, and the Sim Debug Mode option is on the Reports menu.

Instance Names

Instance names can be used to enhance debugging and to guide the floorplanning of soft cells by providing groups of instances with a common prefix. Instance names have one of the four formats shown in Table 9-5, depending on the status of Use Group Names and Sim Debug Mode.

Table 9-5 Instance Names

Sim Debug Mode	Use Group Names	Instance Name
Enabled	Enabled	I<*group name*>_<*op name*>_<*bit position*>_<*cell name*>_I<*unique number*>
Disabled	Enabled	I<*group name*>_I<*unique number*>
Enabled	Disabled	I<*op name*>_<*bit position*>_<*cell name*>_I<*unique number*>
Disabled	Disabled	I<*unique number*>

You can use <*op name*> and <*bit position*> to identify or group instances belonging to a particular operand or to a particular bit of an operand and to place these instances together. Optionally, the name can be extended to include the group name as shown below.

Using short names (Sim Debug Mode disabled) is recommended when you will be going to place and route.

- Instances can be identified in all modes by I <*unique number*>.

Net Names

Net names follow a pattern similar to that for instance names.

Table 9-6 Net Names

Sim Debug Mode	Use Group Names	Instance Name
Enabled	Enabled	NI<*instance_name*>__N<*unique number*>
Disabled	Enabled	N< *group name*>__N<*unique number*>
Enabled	Disabled	NI<*instance_name*>__N<*unique number*>
Disabled	Disabled	N<*unique number*>

In the table above, <*instance_name*> is an instance name that has been generated following the algorithm in Table 9-5.

- Nets can be identified in all modes by N<*unique number*>.

Wire Names

MC creates unique names for all wires in the design as the hierarchy is flattened and whenever temporary operands are created. In all cases, an MC-created signals have names that end in “_”. User-defined signal names are not allowed to end in “_”.

Specifically, MC creates new signal names as follows:

- Module outputs

  These signals are referred to by a local name; at the end of synthesis, the local named variable is assigned to the module output.

  The local name is of the form <*name*>_<*integer*>_ where <*name*> is the name of the output as declared and <*integer*> is an integer quantity.

- Temporary variables created to compute an expression.

  The local name is of the form <*name*>_<*integer*>_ where <*name*> is the name of the signal on the left-hand side of the statement containing the expression. If the expression is an argument to a function—for example, sat(A+B, ...)—then the local name is the first signal argument to the function.

- Wires created inside a function

  These signals are referred to by using a local name. The local name is of the form <*basename*>_<*name*>_<*integer*>_, where <*name*> is the name of the wire as declared and <*integer*>_ is a unique attachment which is created only if a <*basename*>_<*name*>_ already exists. MC creates <*basename*> as follows:

  - Name of the function instance, if provided as described under the function calling convention
  - Otherwise name of the first output of the function if it is declared before the wire statement which lead to the creation of the name
  - Otherwise name of the first signal argument to the function
  - Otherwise the string **temp.**

- Function inputs and outputs

  All function inputs and outputs are named <*basename*>_<*pin name*>.

- temporary variables:names

- Temporary variables created at function boundaries to perform conversion between mismatching parameter widths and/or formats

  Temporary variables are named <*basename*>_<*paramname*>_, where <*basename*> is determined as described above and <*paramname*> is the name of the parameter inside the function.

  The naming of temporary variables used as function outputs (and therefore as basenames for wires inside a function) can be complicated. However, the generated names are consistent with the above rules. For instance,

  A = fnX(B) + fnY(C)

  leads to two function calls such as fnX(A_5_, B) and fnY(A_6_, C). The wires declared inside fnX and fnY are named after A_5_ and A_6_, or the root name A, which is the same as the left-hand side of this expression.

Controlling Names

To control all names in MC, you must provide all functions with instance names, and you must not use complex expressions when assigning to module outputs. If you violate either of these guidelines, MC generates names for you, and you lose control of the naming.

Example 9-3 shows how the wires and I/Os inside a function are named.

Example 9-3 Wire and I/O Names Inside a Function

```
function func2 (H,I,J);
        input I,J;
        output [10:0] H;
        wire [10:0] K=I*J;
        H=K+J;
endfunction

function func1 (Z,X,Y);
        input X,Y;
        output Z;
        wire [10:0] Q=X*X;
        wire [10:0] Q1;
        func2 myname2(Q1,Y,X);
        Z=2*Q-Q1;
endfunction

module mod (D,A,B);

        input [7:0] A,B;
        output [10:0] D;

        wire [8:0] E,F;
        wire [10:0] G;
        E=func1(A,B);
        func1 myname(F,B,A);

        G=E^F;

        D=G;
endmodule
```

Note that *func1*() is called both with and without an instance name from module *mod*. Here are the hierarchical names available:

Table 9-7 Wire Names for Example 9-3

```
mod/A = A
mod/B = B
mod/D = D
mod/E = E
mod/F = F
mod/G =G

mod/E/X = E_X_ = A
mod/E/Y = E_Y_ = B
mod/E/Q = E_Q_
mod/E/Q1 = E_Q1_
mod/E/Z = E_Z_ = E

mod/E/myname2/I =  E_myname2_I_ = B
mod/E/myname2/J = E_myname2_J_ = A
mod/E/myname2/K = E_myname2_K_
mod/E/myname2/H = E_myname2_H_ = E_Q1_

mod/myname/X = myname_X_ = B
mod/myname/Y = myname_Y_ = A
mod/myname/Q = myname_Q_
mod/myname/Q1 = myname_Q1_
mod/myname/Z = myname_Z_ = F

mod/myname/myname2/I =  myname_myname2_I_ = A
mod/myname/myname2/J = myname_myname2_J_  = B
mod/myname/myname2/K = myname_myname2_K_
mod/myname/myname2/H = myname_myname2_H_ = myname_Q1_
```

As you can see, each hierarchical name is translated very simply and predictably. The rule to remember is that hierarchy is flattened by using the underscore (“_”) character rather than the dot (“.”) character.

Verilog Simulation

MC provides both behavioral and gate-level (structural) simulation files. Use the behavioral simulation as a quick way of verifying functionality and gate-level simulation for more detailed timing and functionality verification.

Behavioral Verification

You can simulate your design without ever looking into the behavioral simulation file. All internal wires can be accessed by naming objects according to the rules in the “Naming” section of this chapter. In some cases, MC creates additional operands that appear in the behavioral model. These extra operands can be ignored, since they do not cause user-defined operands to change in meaning.

A few functions are too complex to be accurately modeled behaviorally. Primarily, you should be careful when simulating designs with carrysave operands, pipelining, and/or pipeline loaning. The behavioral model treats these very simply; mismatches between the logic and behavioral models can exist within these structures. All other operands in the design, including the top level outputs, will be correct.

To aid in debugging, the context of the MC language file is placed as a comment before the generated behavioral code. This should help you understand the behavior of the MC functions and also how MC has resolved replication and parameterization.

When MC compiles RAMs and inserts them into the design, the RAM cell instantiated in the behavioral model may not match that in the gate-level netlist. The behavior of the RAM is equivalent, however. This happens when the optimizer swaps the original RAM for another equivalent and presumably better RAM.

Gate-Level Simulation

For simulation, the input, output, and most internal signals are accessible in the Verilog gate-level netlist. The internal signals—those defined by the user that are not inputs or outputs—are useful during detailed timing and functional debugging.

The instance and net names in this file are affected by both the Sim Debug Mode option and the Use Group Names option. See the “Naming” section in this chapter for a full description of naming in MC.

Note: You must set the Sim Debug Mode option in order to examine any wires in the design, other than the module inputs and outputs.

The instances in this file are broken into groups that are annotated with comments indicating the current group and operand.

Getting More Detailed Design Report Information

User-Defined Group Reports

Use hierarchical groups and the custom group reporting mechanism to get more detailed information on your design. The high level groups can be used to get a good idea of the general behavior of the design while lower-level groups are useful when debugging.

Consider the example below. We have broken the video processor into three top level groups: matrix, hide, and fir. Each group has three subgroups: Y, U, and V. By default, MC provides data for the complete matrix, hide, and fir groups. You can request information for all groups related to Y by calling *showgroup*(“*.Y”).

Example 9-4 Requesting More Detailed Design Report Information

```
module video (taps,replicate(integer i=0; i<taps; i=i+1) {YC{i},}R,G,B,Y,U,V);
integer taps;
directive (pipeline="on",delay=9999999);
input signed [7:0] replicate(i=0; i<taps; i=i+1) {YC{i},};
input [7:0] R,G,B;
output [20:0] Y,U,V;
buffer(R,2); buffer(G,2); buffer(B,2);
wire signed [15:0] U1,U_int,V1,V_int;
wire [15:0] Y1,Y_int;

directive (group="matrix.Y"); Y_int=R*89+G*138+B*47;
directive (group="matrix.U"); U_int=0-33*R+144*G+88*B;
directive (group="matrix.V"); V_int=53*R-91*G+102*B;

directive (group="hide.Y"); Y1=hidelat(Y_int,0);
directive (group="hide.U"); U1=hidelat(U_int,0);
directive (group="hide.V"); V1=hidelat(V_int,0);

wire unsigned [9:0] YSR,replicate(i=0; i<=taps; i=i+1) {Y_{i},};
wire signed [9:0] USR,replicate(i=0; i<=taps; i=i+1) {U_{i},};
wire signed [9:0] VSR,replicate(i=0; i<=taps; i=i+1) {V_{i},};
directive (group="fir.Y");
YSR=sreg(Y1[15:6],taps,replicate(i=0; i<=taps; i=i+1) {Y_{i},});
directive (group="fir.U");
USR=sreg(U1[15:6],taps,replicate(i=0; i<=taps; i=i+1) {U_{i},});
directive (group="fir.V");
VSR=sreg(V1[15:6],taps,replicate(i=0; i<=taps; i=i+1) {V_{i},});
directive (group="fir.Y");
Y=replicate (i=0; i<taps; i=i+1) {Y_{i+1}*YC{i}+} 0;
directive (group="fir.U");
U=replicate (i=0; i<taps; i=i+1) {U_{i+1}*YC{i}+} 0;
directive (group="fir.V");
V=replicate (i=0; i<taps; i=i+1) {V_{i+1}*YC{i}+} 0;

showgroup("*.Y"); showgroup("*.U"); showgroup("*.V");
showgroup("fir.*");
showgroup("matrix.*");
endmodule
```

The code above produces the following group information. The first two sections are generated automatically by MC.

```
    GROUP   TIMING (ns)   POWER            AREA            LATENCY
     name  final internal  (W)     ff  inst    sect   c/d   cycles
-------------------------------------------------------------------
      fir   18.3     18.3   0.70  120  1742    6989   4.2       0
     hide    0.0      0.0   0.00    0     0       0             0
   matrix    0.0     13.5   0.14    0   428    1512   5.0       0
     misc    2.5      0.0   0.01    0    66     104   2.9       0
              -----------------------------------------------------
  *   18.3     18.3   0.85  120  2236    8605   4.3       0

    GROUP   TIMING (ns)   POWER            AREA            LATENCY
     name  final internal  (W)     ff  inst    sect   c/d   cycles
-------------------------------------------------------------------
    fir.U    0.0     18.2   0.23   40   573    2290   4.1       0
    fir.V   18.3     18.3   0.23   40   573    2290   4.1       0
    fir.Y    0.0     18.3   0.24   40   596    2409   4.5       0
   hide.U    0.0      0.0   0.00    0     0       0             0
   hide.V    0.0      0.0   0.00    0     0       0             0
   hide.Y    0.0      0.0   0.00    0     0       0             0
 matrix.U    0.0     11.8   0.04    0   120     377   3.0       0
 matrix.V    0.0     13.5   0.05    0   161     629   4.8       0
 matrix.Y    0.0     13.1   0.05    0   147     506   9.1       0
     misc    2.5      0.0   0.01    0    66     104   2.9       0
              -----------------------------------------------------
       **   18.3     18.3   0.85  120  2236    8605   4.3       0

    GROUP   TIMING (ns)   POWER            AREA            LATENCY
     name  final internal  (W)     ff  inst    sect   c/d   cycles
-------------------------------------------------------------------
    fir.Y    0.0     18.3   0.24   40   596    2409   4.5       0
   hide.Y    0.0      0.0   0.00    0     0       0             0
 matrix.Y    0.0     13.1   0.05    0   147     506   9.1       0
              -----------------------------------------------------
      *.Y    0.0     18.3   0.29   40   743    2915   5.0       0

    GROUP   TIMING (ns)   POWER            AREA            LATENCY
     name  final internal  (W)     ff  inst    sect   c/d   cycles
-------------------------------------------------------------------
    fir.U    0.0     18.2   0.23   40   573    2290   4.1       0
   hide.U    0.0      0.0   0.00    0     0       0             0
 matrix.U    0.0     11.8   0.04    0   120     377   3.0       0
              -----------------------------------------------------
      *.U    0.0     18.2   0.27   40   693    2667   3.9       0

    GROUP   TIMING (ns)   POWER            AREA            LATENCY
     name  final internal  (W)     ff  inst    sect   c/d   cycles
-------------------------------------------------------------------
    fir.V   18.3     18.3   0.23   40   573    2290   4.1       0
   hide.V    0.0      0.0   0.00    0     0       0             0
 matrix.V    0.0     13.5   0.05    0   161     629   4.8       0
              -----------------------------------------------------
     *.V   18.3     18.3   0.28   40   734    2919   4.2       0
```

```
                GROUP   TIMING (ns)   POWER            AREA           LATENCY
                 name  final internal  (W)    ff  inst   sect   c/d   cycles
--------------------------------------------------------------------------------
                fir.U    0.0     18.2   0.23   40   573   2290   4.1     0
                fir.V   18.3     18.3   0.23   40   573   2290   4.1     0
                fir.Y    0.0     18.3   0.24   40   596   2409   4.5     0
               -----------------------------------------------------------------
                fir.*   18.3     18.3   0.70  120  1742   6989   4.2     0

                GROUP   TIMING (ns)   POWER            AREA           LATENCY
                 name  final internal  (W)    ff  inst   sect   c/d   cycles
--------------------------------------------------------------------------------
             matrix.U    0.0     11.8   0.04    0   120    377   3.0     0
             matrix.V    0.0     13.5   0.05    0   161    629   4.8     0
             matrix.Y    0.0     13.1   0.05    0   147    506   9.1     0
               -----------------------------------------------------------------
             matrix.*    0.0     13.5   0.14    0   428   1512   5.0     0
```

User-Defined Critical Paths

You can specify paths to analyze in addition to those automatically chosen by MC. You might want to do this if the paths reported by MC are false or you are interested in looking at paths that are not the most critical path of the design or any group. Another reason to define custom paths is to examine the delay between internal operands (those that are neither the start nor end of the critical paths). The paths are defined using special functions in the MC language (see “Path Analysis” in Chapter 5).

Below is an example of a very simple circuit with complex analysis of the critical paths.

There are two outputs in the circuit, D and F; D is a four-level buffered version of A, and F is the sum of A and B. Logic optimization is disabled to prevent all of the buffers from disappearing.

Example 9-5 A Simple Circuit with Complex Analysis of the Critical Paths

```
module foo (A,B,D,F);
input [7:0] B;
directive (indelay=3000,logopt="off");
input [7:0] A;
wire [7:0] A1=isolate(A), A2=isolate(A1), A3=isolate(A2);
wire [7:0] A4=isolate(A3);
output [7:0] D=A4;
wire [7:0] C=A+B;
wire [7:0] E=isolate(C);
output [7:0] F=E;

critpath("A","*","A_to_anywhere");
disablepath("F");
critpath("A","*","A_to_anywhere_but_F");
disablepath("D");
critpath("A","*","A_to_anywhere_but_F_or_D");
enablepath("D"); enablepath("F"); disablepath("E");
critpath("A","*","A_to_anywhere_but_E");
enablepath("E"); disablepath("C");
critpath("A","*","A_to_anywhere_but_C");
enablepath("C[4:0]");
critpath("A","*","A_to_anywhere_but_C[5:7]");
enablepath("C[7:5]");
critpath("A","*","A_to_anywhere");

critmode ("short");
critpath ("A2", "A3", "path2");
critpath ("A2[3]", "A3[3]", "path3");
critpath ("A2[3]", "A3[2]", "path4");
critpath ("*", "A3", "path5");
critpath ("B", "A3", "path6");
critpath ("A4", "A1", "path7");
critpath ("A1", "A4", "path8");
critpath ("A1", "*", "path9");

critmode ("full");
critpath ("A2", "A3", "path2");
critpath ("A2[3]", "A3[3]", "path3");
critpath ("A2[3]", "A3[2]", "path4");
critpath ("*", "A3", "path5");
critpath ("B", "A3", "path6");
critpath ("A4", "A1", "path7");
critpath ("A1", "A4", "path8");
critpath ("A1", "*", "path9");
endmodule
```

The results for the example are shown below. The critical path for the design goes from *A* through the adder to *F*. When *F* is deactivated, the next most critical path is to *D*. When *F* and *D* are disabled, there are no paths.

Example 9-6 Output from the Simple Circuit/Complex Analysis Example

```
User-Defined Critical Paths
--------------------------------------------------------------------------------

Critical Path 'A_to_anywhere': from A to *.  End point: F[7].
critical pin -> critical net    delta  delay   rise   fall   load  gload   pins
                       setup:    0.00   8.44   8.37   8.44
        A -> E_7_buf1a2_49_Y:    0.76   8.44   8.37   8.44   12.5   10.0      2
        CI -> C_7_fa1b1_41_S:    0.61   7.69   7.69   7.69    3.3    0.8      2
       CI -> C_6_fa2a1_40_CO:    0.50   7.08   7.08   7.05    7.6    5.1      2
       CI -> C_5_fa1b1_39_CO:    0.57   6.58   6.51   6.58    7.9    5.4      2
       CI -> C_4_fa2a1_38_CO:    0.50   6.01   6.01   6.00    7.6    5.1      2
       CI -> C_3_fa1b1_37_CO:    0.56   5.51   5.46   5.51    7.9    5.4      2
       CI -> C_2_fa2a1_36_CO:    0.51   4.95   4.94   4.95    7.6    5.1      2
       CI -> C_1_fa1b1_35_CO:    0.54   4.44   4.41   4.44    7.9    5.4      2
        B -> C_0_fa2a1_34_CO:    0.90   3.90   3.87   3.90    7.6    5.1      2
                        A[0]:    3.00   3.00   3.00   3.00    9.8    4.8      3

Deactivating path through F

Critical Path 'A_to_anywhere_but_F': from A to *.  End point: D[0].
critical pin -> critical net    delta  delay   rise   fall   load  gload   pins
                       setup:    0.00   5.50   5.23   5.50
       A -> A4_0_buf1a2_26_Y:    0.75   5.50   5.23   5.50   12.5   10.0      2
       A -> A3_0_buf1a2_18_Y:    0.58   4.75   4.54   4.75    3.3    0.8      2
       A -> A2_0_buf1a2_10_Y:    0.58   4.16   4.03   4.16    3.3    0.8      2
        A -> A1_0_buf1a2_2_Y:    0.58   3.58   3.52   3.58    3.3    0.8      2
                        A[0]:    3.00   3.00   3.00   3.00    9.8    4.8      3

Deactivating path through D

Critical Path 'A_to_anywhere_but_F_or_D': from A to *.
No Path found!

Reactivating path through D

Reactivating path through F

Deactivating path through E

Critical Path 'A_to_anywhere_but_E': from A to *.  End point: D[0].
critical pin -> critical net    delta  delay   rise   fall   load  gload   pins
                       setup:    0.00   5.50   5.23   5.50
       A -> A4_0_buf1a2_26_Y:    0.75   5.50   5.23   5.50   12.5   10.0      2
       A -> A3_0_buf1a2_18_Y:    0.58   4.75   4.54   4.75    3.3    0.8      2
       A -> A2_0_buf1a2_10_Y:    0.58   4.16   4.03   4.16    3.3    0.8      2
        A -> A1_0_buf1a2_2_Y:    0.58   3.58   3.52   3.58    3.3    0.8      2
                        A[0]:    3.00   3.00   3.00   3.00    9.8    4.8      3

Reactivating path through E
```

```
Deactivating path through C

Critical Path 'A_to_anywhere_but_C': from A to *.  End point: D[0].
critical pin -> critical net    delta  delay   rise   fall   load  gload   pins
                      setup:    0.00   5.50   5.23   5.50
       A -> A4_0_buf1a2_26_Y:    0.75   5.50   5.23   5.50   12.5   10.0     2
       A -> A3_0_buf1a2_18_Y:    0.58   4.75   4.54   4.75    3.3    0.8     2
       A -> A2_0_buf1a2_10_Y:    0.58   4.16   4.03   4.16    3.3    0.8     2
        A -> A1_0_buf1a2_2_Y:    0.58   3.58   3.52   3.58    3.3    0.8     2
                       A[0]:    3.00   3.00   3.00   3.00    9.8    4.8     3

Reactivating path through C[4:0]

Critical Path 'A_to_anywhere_but_C[5:7]': from A to *.  End point: F[4].
critical pin -> critical net    delta  delay   rise   fall   load  gload   pins
                      setup:    0.00   6.87   6.80   6.87
        A -> E_4_buf1a2_46_Y:    0.76   6.87   6.80   6.87   12.5   10.0     2
        CI -> C_4_fa2a1_38_S:    0.60   6.11   6.11   6.11    3.3    0.8     2
       CI -> C_3_fa1b1_37_CO:    0.56   5.51   5.46   5.51    7.9    5.4     2
       CI -> C_2_fa2a1_36_CO:    0.51   4.95   4.94   4.95    7.6    5.1     2
       CI -> C_1_fa1b1_35_CO:    0.54   4.44   4.41   4.44    7.9    5.4     2
        B -> C_0_fa2a1_34_CO:    0.90   3.90   3.87   3.90    7.6    5.1     2
                       A[0]:    3.00   3.00   3.00   3.00    9.8    4.8     3

Reactivating path through C[7:5]

Critical Path 'A_to_anywhere': from A to *.  End point: F[7].
critical pin -> critical net    delta  delay   rise   fall   load  gload   pins
                      setup:    0.00   8.44   8.37   8.44
        A -> E_7_buf1a2_49_Y:    0.76   8.44   8.37   8.44   12.5   10.0     2
        CI -> C_7_fa1b1_41_S:    0.61   7.69   7.69   7.69    3.3    0.8     2
       CI -> C_6_fa2a1_40_CO:    0.50   7.08   7.08   7.05    7.6    5.1     2
       CI -> C_5_fa1b1_39_CO:    0.57   6.58   6.51   6.58    7.9    5.4     2
       CI -> C_4_fa2a1_38_CO:    0.50   6.01   6.01   6.00    7.6    5.1     2
       CI -> C_3_fa1b1_37_CO:    0.56   5.51   5.46   5.51    7.9    5.4     2
       CI -> C_2_fa2a1_36_CO:    0.51   4.95   4.94   4.95    7.6    5.1     2
       CI -> C_1_fa1b1_35_CO:    0.54   4.44   4.41   4.44    7.9    5.4     2
        B -> C_0_fa2a1_34_CO:    0.90   3.90   3.87   3.90    7.6    5.1     2
                       A[0]:    3.00   3.00   3.00   3.00    9.8    4.8     3
               path2:   0.58
               path3:   0.58
               path4: No Path found!
               path5:   4.75
               path6: No Path found!
               path7: No Path found!
               path8:   1.92
               path9:   1.92

Critical Path 'path2': from A2 to A3.  End point: A3[0].
critical pin -> critical net    delta  delay   rise   fall   load  gload   pins
                      setup:    0.00   0.58   0.52   0.58
            A3_0_buf1a2_18_Y:    0.58   0.58   0.52   0.58    3.3    0.8     2

Critical Path 'path3': from A2[3] to A3[3].  End point: A3[3].
critical pin -> critical net    delta  delay   rise   fall   load  gload   pins
                      setup:    0.00   0.58   0.52   0.58
            A3_3_buf1a2_21_Y:    0.58   0.58   0.52   0.58    3.3    0.8     2
```

```
Critical Path ’path4’: from A2[3] to A3[2].
No Path found!

Critical Path ’path5’: from * to A3.  End point: A3[0].
critical pin -> critical net    delta  delay   rise   fall   load  gload   pins
                       setup:    0.00   4.75   4.54   4.75
       A -> A3_0_buf1a2_18_Y:    0.58   4.75   4.54   4.75    3.3    0.8     2
       A -> A2_0_buf1a2_10_Y:    0.58   4.16   4.03   4.16    3.3    0.8     2
        A -> A1_0_buf1a2_2_Y:    0.58   3.58   3.52   3.58    3.3    0.8     2
                        A[0]:    3.00   3.00   3.00   3.00    9.8    4.8     3

Critical Path ’path6’: from B to A3.
No Path found!

Critical Path ’path7’: from A4 to A1.
No Path found!

Critical Path ’path8’: from A1 to A4.  End point: A4[0].
critical pin -> critical net    delta  delay   rise   fall   load  gload   pins
                       setup:    0.00   1.92   1.71   1.92
       A -> A4_0_buf1a2_26_Y:    0.76   1.92   1.71   1.92   12.5   10.0     2
       A -> A3_0_buf1a2_18_Y:    0.58   1.16   1.03   1.16    3.3    0.8     2
            A2_0_buf1a2_10_Y:    0.58   0.58   0.52   0.58    3.3    0.8     2

Critical Path ’path9’: from A1 to *.  End point: D[0].
critical pin -> critical net    delta  delay   rise   fall   load  gload   pins
                       setup:    0.00   1.92   1.71   1.92
       A -> A4_0_buf1a2_26_Y:    0.76   1.92   1.71   1.92   12.5   10.0     2
       A -> A3_0_buf1a2_18_Y:    0.58   1.16   1.03   1.16    3.3    0.8     2
            A2_0_buf1a2_10_Y:    0.58   0.58   0.52   0.58    3.3    0.8     2
```

Running Design Compiler

Introduction

You can optionally choose to call Design Compiler (DC) at the end of the report generation phase to postprocess the network created by MC. Select Design Compiler from the Optimization menu, and Run Design Compiler from the submenu. When you select this option, MC creates a constraint and command file for Design Compiler and then runs Design Compiler. You can select Design Compiler Report and Design Compiler Output Netlist from the View menu to see the outputs from Design Compiler, but the network is not imported back to MC for further processing.

It is fairly simple to control Design Compiler from MC. First, make sure that Design Compiler is properly installed. Then set the **dcopt** attribute **on** in your MC language description for those parts of the circuit that Design Compiler is allowed to modify. By default, **dcopt** is **off**. Next, set the options for running Design Compiler, either from the command line or from the GUI. Design Compiler will now run automatically when the reports for the circuit are generated.

The Constraint and Command Files

MC creates a constraint file for DC to ensure that the constraints you entered in your MC language description are used during processing by Design Compiler. The constraint file contains the following information:

- Delay goal for all outputs and sequential element inputs
- Input arrival times
- Input maximum loading
- Output setup times
- Output external loads
- “Don’t Touch” attributes for all instances created with **dcopt off**

The beginning of the constraint file contains the commands to load the Verilog netlist and to link the circuit, in addition to commands to select the operating condition and wire load model.

MC creates a command file to specify the actions to be performed by Design Compiler. It contains the following commands:

- Generate timing reports for each group (optional)
- Compile (optional)
- Check the Design (optional)
- Generate an area and timing report for the design
- Write the final network in Verilog syntax to a file

The commands which are optional are controlled via GUI or command line options. The compile command is further modified by other options to select incremental mapping and the mapping effort. You should select only those commands that are needed to avoid excessive run time.

MC provides control over two options of the compile command: **Map Incremental** and **Map Effort**. Use incremental mapping to prevent Design Compiler from changing the circuit structure significantly. Disable this option to make more significant structural changes. Set the mapping effort higher to enable greater degrees of optimization with corresponding improvements in circuit quality and increases in run time.

Running Design Compiler with Designs that Contain RAMs

The DC interface of MC does not support RAMs. If you have a RAM in your design, DC treats it as a black box.

Customizing the Way Design Compiler Runs

Because of Design Compiler's complexity, MC does not attempt to provide control over all features. For more advanced use of Design Compiler, you can customize the way in which it is run.

You invoke Design Compiler using a shell script that concatenates the constraint and command files and passes the combination to `dc_shell`. MC executes the script specified by the value of the MC environment variable, *dp_dcscript_fname*. You can create your own script which uses or modifies the constraint and command files from MC or which inserts your own commands and constraints. The script should return the exit status of DC. Make sure you modify *dp_dcscript_fname*.

By default, *dp_dcscript_fname* is set to `run_dc`. This script is supplied with MC and makes a call to `dc_shell`. It concatenates the constraint and command files and runs Design Compiler with the combination.

Example

As an example of the use of the Design Compiler for post optimization, consider a double precision floating point multiplier. The bulk of the circuit is the 54×54 integer multiplier (Wallace tree and final adder occupying 4283 of 5052 total sections). You suspect that the exponent and exception handling logic could be improved by Design Compiler.

The table below shows the final delay after running DC for various input options. For all cases, MC's optimizer brought the delay from 33 nanoseconds to 28.5 nanoseconds in about 4 minutes before running DC. The multiplier/adder delay is 16 nanoseconds.

Table 9-8 The Effect of Various Design Compiler Input Options

Final Delay	Map Effort	Incremental Map	Don't Touch	Run Time
28.7	low	no	multiplier/adder	6 min
28.5	low	no	all	4 min
28.4	high	no	none	3 days
28.1	high	yes	multiplier/adder	12 min
27.8	high	yes	none	42 min
27.2	med	no	multiplier/adder	48 min
26.4	high	no	multiplier/adder	80 min

As expected, Design Compiler makes significant progress on the nonarithmetic part of the circuit if the options are set correctly. The value of properly setting the **dcopt** attribute can be clearly seen. When all of the circuit is processed by Design Compiler, the result degrades and the run time increases dramatically.

Debugging

Debugging designs created by the MC requires many of the same skills required to successfully debug any circuit description. Therefore, we will focus mainly on those techniques which are more specific to MC.

Flattening the Input

There may be occasions when it is unclear how the integer parameters, replicates, conditional statements, or other abstractions of the MC language are resolved. To help understand these effects, MC provides an option to flatten the input (the Flatten Input option in the File menu). When this option is selected, you can see the flattened input in the log window.

Before synthesis starts, the macros, integer parameters, replicates, conditions and functions are removed. Temporary signals are generated when complex expressions are broken into synthesizable expressions. In addition, any wire formats or widths which were not specified are determined.

Consider the example shown below with one level of hierarchy and some control flow constructs.

```
function choose (C,A,B);
input A,B;
wire [0:0] Parity=repl(i,width(A),"^") {A[{i}]};
output C=Parity ? A : A+B;
endfunction
module adder(X,Y,ZA,ZB);
integer width=8;
input [width-1:0] X,Y;
output [width-1:0] ZA=choose(X,Y);
if (width<16) {
     output [width-1:0] ZB=choose(ZA,Y);
} else {
     output [width-1:0] ZB=choose(Y,ZA);
}
endmodule
```

When flattened, the following is produced. The function calls have been flattened and the **repl, if** and integer constructs have been resolved. You can also see how the temporary variables were declared (for the addition) and how the hierarchical names are created.

```
module adder(X, Y, ZA, ZB);
input unsigned [7:0] X;
input unsigned [7:0] Y;
wire unsigned [7:0] ZA_1_;
output unsigned [7:0] ZA = ZA_1_;
wire unsigned [0:0] ZA_Parity_;/* declared as Parity */
ZA_Parity_ = X[0:0] ^ X[1:1] ^ X[2:2] ^ X[3:3] ^ X[4:4] ^ X[5:5] ^ X[6:6] ^
X[7:7];
wire unsigned [7:0] ZA_2_;
ZA_2_ = X + Y;
ZA_1_ = ZA_Parity_ ? X : ZA_2_;
wire unsigned [7:0] ZB_3_;
output unsigned [7:0] ZB = ZB_3_;
wire unsigned [0:0] ZB_Parity_;/* declared as Parity */
ZB_Parity_ = ZA_1_[0:0] ^ ZA_1_[1:1] ^ ZA_1_[2:2] ^ ZA_1_[3:3] ^ ZA_1_[4:4] ^ ZA_
1_[5:5] ^ ZA_1_[6:6] ^ ZA_1_[7:7];
wire unsigned [7:0] ZB_4_;
ZB_4_ = ZA_1_ + Y;
ZB_3_ = ZB_Parity_ ? ZA_1_ : ZB_4_;
endmodule
```

Syntax and Synthesis Errors/Warnings

When errors and warnings occur, try to resolve them first. Many warnings result from designer errors and should be examined carefully. You can often elaborate on errors in the input files by using **info** statements.

Generally, it is also a good idea to enable verbose mode. In verbose mode, MC reports the statistics of each operand after it is synthesized (except for shift register structures). MC also reports more informational messages regarding loading and pre-optimization statistics. If the network description contains errors, you will often notice area, delay, or flip-flop usage that is clearly unreasonable.

Logic Errors

Logic errors cannot be detected by MC. You must use standard debugging skills such as simulation and summary information.

The behavioral model can be used to debug logical errors in the network description. The structure and naming in this model is virtually identical to that of the network description. If you are familiar with Verilog HDL, it might also be beneficial to look at the behavioral model to see if the behavior matches what is expected. Statements in the behavioral model are preceded by a comment statement. This comment indicates the line in the input file which led to the generation of the following block of behavioral code. Checking this translation can help determine if unexpected replication or parameterization effects have occurred. Note the use of temporaries. In particular, when arithmetic operations (+,-,*,<<) require temporary operands, the intermediate result is likely to have a result smaller than desired, producing truncation errors.

The summary information is often a good starting point for detecting gross errors. Check for computed operands which have a constant output or have widths which are too large or too small. Also check for operands which have extreme areas or flip-flop usage. Operands which have no connections are also listed; check to make sure none of these should be connected.

Use common sense. It is important to know approximately what the area and delay of a function should be (see the *Module Compiler Reference Manual*). Logic errors can result in blocks which are much too fast or too slow, too big or too small. A 64-bit adder which is 100 ns or 1 ns is probably not correct.

Poor Combinatorial Timing

Poor timing can result from a number of factors: impossible delay constraints, selecting a poor architecture, operands loaded beyond the estimated load, or improper sign extension in integer functions. Using groups and critical path information can help to track the problem down. Following is a list of hints.

- **Use the critical path information** In general, start by checking the critical path to identify the operand(s) through which the path goes. Be careful when interpreting the critical path information. The use of delay equalization changes the meaning of the critical path when the delay goal is not met. With equalization enabled, paths can be slowed down to the speed of the critical paths (and may then become critical by a few ps if improvements are made on the critical path). Look for excessive numbers of logic levels or buffering. Also look for nets which are heavily loaded.

- **Use groups to divide the design**. The proper use of groups will help track down problems, because each group has its own statistics and critical path information. Breaking the design into groups and disabling global equalization helps determine where the problem areas are.

- **Use realistic delay constraints**. You have full control over the delay constraint and should realize that there are limits to the performance which can be obtained in a given technology. It may be necessary to use pipelining or to reduce the amount of computation performed.

- **Investigate alternative architectures**. For functions with multiple architectures, it may be beneficial to try an alternative architecture. Note that the default fastest architectures such as **fastcla** may not always be the fastest given the details of the network. Try to choose the architecture which fits the overall constraints most closely.

- **Use inversion if possible**. If the critical path includes *sat*(), *shift*(), *rotate*(), and/or *mux*() functions, try to use the inverting option. This reduces the delay but changes the functionality of the network. Additional inversion must be added to other parts of the network to compensate for the added inversion.

- **Don’t overload operands**. When there are heavily loaded nets on the critical path, try using *isolate*() to isolate the noncritical paths from the more critical paths. The verbose mode also helps to locate nets which were overloaded before optimization and were fixed during optimization.

- **Use direct sign extension when needed**. The default sign extension produces performance problems in a few degenerate cases. Check the section on sign extension to see if the **direxzt** attribute should be set to **on**. This can potentially save one inverter and one full adder delay.

- **Don’t disable logic optimization**. If logic optimization is completely disabled either locally or globally, performance suffers greatly for some cases.

- **If the delstate directive is used to pipeline, be sure that it has a reasonable value**. In general a value of 1 to 3 seems to work best.

- **Check operand widths** to ensure that mistakes were not made when creating a highly parameterized design. When complex parameterization is used, gross errors can easily occur without being noticed.

- **Don’t set the delay goal too low**. Some structures get stuck in local minimums when optimizing for speed. Try setting the delay goal to a realistic value. Also, try changing the type of equalization and when it is employed. Generally, starting optimization without equalization and ending optimization with equalization is the best strategy, but starting with equalization can sometimes be better.

- **Check for FFs with a clock input not connected to CLK**. The *CLK* signal is treated specially, and some improvements cannot be found when *CLK* is not used as the clock input.

- **Use user-defined critical paths and the I/O summary to gather more timing data** to ensure that the actual behavior matches that which is expected. Make sure that inputs and outputs expected to be noncritical are in fact so.

Pipelining Problems and Excessive Flip-Flop Usage

Improper use of pipelining can lead to extreme results. The principle problems involve automatic latency deskewing, excessive loading and delay goals set too low.

- **Hide latency at the inputs to loops**. When the design contains loops, you must be careful to hide the latency of all signals entering the loop, using *hidelat*(). Otherwise, pipelines are inserted into the loop and are detected as an error condition.

- **Hide latency when connecting CLK to an instance**. Another potential problem occurs when foreign cells have a connection to *CLK*. If deskewing is required at the instance inputs, an attempt to pipeline *CLK* is made. This is flagged as an error condition.

- **Be careful with latency differences and high fanout structures**. When signals with large latency differences interact inside of a structure with large fanouts (a multiplier, for example), a large number of FFs can be used. You should either use *hidelat*() to prevent deskewing or manually equalize the delays before the fanout occurs.

- **Choose a reasonable delay goal**. If the delay goal is set very small, a very large number of FFs can be used. You can generally reduce the area by resynthesizing the circuit with a delay goal equal to the final delay achieved. In particular, optimization for **speed** should not be selected when pipelining.

- **Don't overload operands when pipelining**. Because the pipelines are inserted during synthesis using estimated loading values (not just estimated wire loading, but estimated gate loading also), you should be careful not to load critical operands beyond the estimated loading. When the estimated load is exceeded, the delay estimate is optimistic and the delay goal may not be met. This problem is best solved with the proper use of buffering and isolation. The use of the pipeline slack parameter can be used to conveniently provide additional margin when pipelining, but this margin is applied globally which results in less efficient use of area.

Carrysave Problems

Carrysave operands can be used in a very restricted way and the behavioral model produces inexact results for these operands (the final results are correct, however). As explained previously, carrysave operands can be used in only a few operations.

- The maximum number of bits allowed per column (bit position) is 2. If the process of converting carrysave operands using the *convert*() function produces the "Too many bits in column" message when it fails, use the **convert** option for the **carrysave** attribute to ensure that the output has at most 2 bits per column,

Rule Violations

If the design has overloaded net violations, it is generally for one of the following reasons:

- Logic optimization is disabled locally or globally
- An impossible constraint was given (check the value of the **outload** attribute at the outputs)

Data Format Problems

The data format of the operands is used extensively during synthesis. You should check the operand summary in the Design Report file to ensure that each operand has the intended format.

Ridiculous Outputs

If you notice that the design contains extreme structures, it may be because logic optimization has been disabled. Certain synthesis routines create structures that are inefficient. The logic optimizer can improve such structures significantly. The following are examples of structures that rely on logic optimization:

- *sreg*() with pipeline loaning
- *hidelat*()
- Incrementors, comparators
- Many structures with constant or partially constant inputs

Poor Utilization

Poor utilization can be caused by:

- Excessive Wallace tree depth. Try using **maxtreedepth** with a value between 32 and 64.

- The compute/drive ratio is very different from the desired value for a CBA library. Make sure optimization is enabled. The Design Report file contains a “Max Possible Utilization” entry that indicates the maximum utilization possible. If this value is lower than expected for a balanced design, you may need to relax the delay goal.

Excessive Runtime and Memory Usage

MC is normally very efficient in its use of run time and memory. There are some conditions that can result in abnormally high use of runtime and memory considering the size of the circuit being synthesized. Generally, you should try to avoid using many very small operands in the computation of a very large operand. In such a case, the extension of the small operands results in a great deal of memory allocation and de-allocation (which also takes time). In addition, the object lists increase in length, resulting in greater search time for the objects. When the number of constructs in the input description is large (regardless of the complexity of the final circuit), the runtime and memory efficiency may suffer. For most reasonable cases where the operand width is less than 100 bits, this should not be a problem.

Optimization

Logic optimization in MC has several steps. MC has an effective default strategy for controlling these steps. This strategy consists of two parts, one controlled by MC and one user-controlled. As you become more expert, you will probably want to fine-tune the strategy to improve the results.

MC Strategy

The strategy controlled by MC is designed to provide good overall results and is reflected in the ordering of optimization steps; you can select which steps are executed, but not the ordering. The ordering is shown in Table 9-9. Note that in general, strict improvement optimizations are performed first, followed by rule optimizations, then reversible, and finally nonreversible optimizations. Nonreversible optimizations are those which cannot be undone by a later optimization step.

Table 9-9 The Module Compiler Optimization Strategy

Optimization Step	Type
Gate Eater	strict improvement
Rules	irreversible
LogicMin1	irreversible
LogicMin5	irreversible
Reorder	reversible
Timing	reversible
Reorder	reversible
Area	reversible
LogicMin2	irreversible
LogicMin3	irreversible
LogicMin4	irreversible
Reorder	reversible
Timing	reversible

There are several reasons for this strategy:

- Strict improvement optimizations should be done first as there is no reason not to wait.
- Rule checks should be done as early as possible because illegal circuits have no value and there is no sense trying to improve timing or area for an illegal design.
- Irreversible optimizations should be done as late as possible to ensure that swaps are not made in an area that appears to be noncritical but which later becomes critical. The irreversible nature of the optimization makes it virtually impossible to undo.
- Pin reordering helps between the major reversible optimizations to prevent getting stuck in a local minimum.

Design Strategy

You, as the designer, can control the logic optimization process in several ways: choosing which steps are performed, the number of local iterations, the number of global iterations, when delay equalization is used and which instances are optimized.

Each optimization step described can be enabled or disabled from the GUI. You can also do this using the **-opt** <*value*> command line option.

The logic optimization performed during synthesis and final optimization should not be disabled for normal operation, because many synthesis routines have been written with the expectation that logic minimization will improve many special cases. Greatly inferior results with minimal improvements in execution time can occur.

You can also control the maximum number of times (local iterations) each optimization step is performed. Each optimization step is repeated locally the specified number of times or until no progress can be made. Generally, a value of 3 to 4 is a good choice. Smaller values can be useful for speeding up the execution time for large circuits. Larger values can be chosen if a given step is terminated while still making progress.

You can also specify the number of global iterations, either through the GUI or at the command-line interface. This number of iterations is always the same as that of the specified value, even if no apparent progress has been made. Generally, 2 or 3 global iterations produces good results, but for certain structures, more iterations may be beneficial.

You must choose the number of global iterations to perform with delay equalization and, if delay equalization is used, whether it should be global or local. The command line option, **-ep** <*value*> indicates that the last <*value*> global iterations uses delay equalization. To select global delay equalization (equalize over timing groups), use the option **-eg** +. For local equalization (over groups) use **-eg** -.

The use of equalization can have dramatic affects on complex circuits. If all global iterations utilize equalization, swaps may be made early that reduce area in an apparently timing noncritical portion. Further optimization may turn the noncritical area into the critical area, due to improvements in the previously critical paths. If the swaps are irreversible, the circuit performance suffers. If no equalization is used, critical paths may not be improved because lesser critical paths which have also not met the delay goal will not tolerate any slowdown. A good compromise is to perform one or two global iterations without equalization, followed by one or two with equalization. Normally, you choose local equalization when you want to see how close each group has come to achieving the delay goal. Global equalization can cause some groups within the same timing group to slow down (saving area) to the delay of the slowest group in the timing group.

Optimization Example

The following is the optimization log for a complex design with many groups. This design includes pipelining, loops, RAMs, shift registers with pipeline loaning and latency equalization. It was optimized for a delay of 11.75 ns with two global passes, one of which employed local equalization. The number of local iterations was set to four and all optimization steps were enabled.

Note the following items of interest in the example:

- The critical path moves between groups during the optimization. Although this design employs groups with different delay goals, all of the critical groups had the same delay goal. When the critical path changes between groups with different delay groups, be sure to look at the slack rather than the delay numbers to monitor progress. In the case, the optimizer was successful in driving the slack to zero.
- Timing optimization results in increases in the area and power of the circuit, while decreasing the critical path delay. Note that negative number in the “net changes” section indicates a growth in either instances or sections. In the second pass, when the delay goal has been met, the timing optimization step is skipped.
- Overloaded nets were repaired without an increase in critical path length. When the critical path length increases during this step, you should try to buffer or isolate the affected nets.
- This is a CBA library with an intrinsic compute-to-drive ratio of 3.0.
- The area measure includes the compute/drive ratio and hence the area optimization drives this ratio to 3.0. In some cases, it can be seen that the total number of sections was increased during some logic minimization steps. This is allowed to improve the compute/drive ratio.
- The optimizer makes a large number of swaps. The design ends up with 3771 instances after making 11953 swaps; each instance was swapped about three times. During these swaps 983 instances and 4556 sections were removed. The timing improved by 1.72 nanoseconds.

Example 9-7 Optimization Log for a Complex Example

```
Beginning Timing Optimization ...
TIMING (ns)  POWER        AREA        OPTIMIZATION     NET CHANGES    crit
delay  slack  (W)    inst   sect   c/d      Step      swaps  inst sect  group
--------------------------------------------------------------------------------
13.471  -1.72  0.995   4754  23375   2.5   Gate Eater     319   319  703  FB_Filter
13.145  -1.40  0.970   4435  22672   2.5   Gate Eater       0     0    0  FF_Update
13.145  -1.40  0.970   4435  22672   2.5        Rules      17   -24  -52  FF_Update
13.145  -1.40  0.971   4459  22724   2.5        Rules       0     0    0  FF_Update

Pass 1, Not Equalizing Delays ...
TIMING (ns)  POWER        AREA        OPTIMIZATION     NET CHANGES    crit
delay  slack  (W)   insts  sects   c/d      Step      swaps  inst sect  group
--------------------------------------------------------------------------------
13.145  -1.40  0.971   4459  22724   2.5   LogicMin 1      20    -8   47  FF_Update
13.145  -1.40  0.970   4467  22677   2.5   LogicMin 1       0     0    0  FF_Update
13.145  -1.40  0.970   4467  22677   2.5      Reorder     465     0    0  FF_Update
13.145  -1.40  0.970   4467  22677   2.5       Timing     742     0 -647  FF_Update
12.672  -0.92  0.983   4467  23324   2.2       Timing     154     0  -75  FF_Update
12.649  -0.90  0.985   4467  23399   2.2       Timing      34     0  -29  FF_Update
12.649  -0.90  0.986   4467  23428   2.2       Timing       8     0   -4  FF_Update
12.649  -0.90  0.986   4467  23432   2.2      Reorder     182     0    0  FF_Update
12.649  -0.90  0.986   4467  23432   2.2   Area/Power    3246     0 2905  FF_Update
12.322  -0.57  0.942   4467  20527   3.0   Area/Power     666     0  205  FF_Update
12.319  -0.57  0.942   4467  20322   3.0   Area/Power     569     0   17  FF_Update
12.319  -0.57  0.942   4467  20305   3.0   Area/Power     608     0    8  FF_Update
12.319  -0.57  0.942   4467  20297   3.0   LogicMin 2     497   497  905  FF_Update
11.824  -0.07  0.926   3970  19392   3.2   LogicMin 2       1     1    2  FB_Filter
11.824  -0.07  0.926   3969  19390   3.2   LogicMin 2       0     0    0  FB_Filter
11.824  -0.07  0.926   3969  19390   3.2   LogicMin 3     128   128  388  FB_Filter
11.784  -0.03  0.921   3841  19002   3.2   LogicMin 3       8     8   32  PLL
11.784  -0.03  0.920   3833  18970   3.2   LogicMin 3       0     0    0  PLL
11.784  -0.03  0.920   3833  18970   3.2   LogicMin 4      47    40   49  PLL
11.784  -0.03  0.919   3793  18921   3.2   LogicMin 4       4     3    5  PLL
11.784  -0.03  0.919   3790  18916   3.2   LogicMin 4       0     0    0  PLL
11.784  -0.03  0.919   3790  18916   3.2      Reorder     375     0    0  PLL
11.744   0.01  0.919   3790  18916   3.2       Timing     116     0    0  PLL
11.744   0.01  0.919   3790  18916   3.2       Timing      10     0    0  PLL
11.744   0.01  0.919   3790  18916   3.2       Timing       5     0    0  PLL
11.744   0.01  0.919   3790  18916   3.2       Timing       4     0    0  PLL

Pass 2, Equalizing Delays Locally ...
TIMING (ns)  POWER        AREA        OPTIMIZATION     NET CHANGES    crit
delay  slack  (W)   insts  sects   c/d      Step      swaps  inst sect  group
--------------------------------------------------------------------------------
11.744   0.01  0.920   3790  18916   3.2   LogicMin 1       0     0    0  PLL
11.744   0.01  0.920   3790  18916   3.2      Reorder     170     0    0  PLL
11.716   0.03  0.920   3790  18916   3.2   Area/Power     718     0   72  PLL
11.749   0.00  0.922   3790  18844   3.0   Area/Power     803     0    4  FB_Filter
11.749   0.00  0.922   3790  18840   3.0   Area/Power     845     0    2  FB_Filter
11.749   0.00  0.922   3790  18838   3.0   Area/Power     878     0    0  FB_Filter
11.749   0.00  0.922   3790  18838   3.0   LogicMin 2       5     5    7  FB_Filter
11.749   0.00  0.922   3785  18831   3.0   LogicMin 2       1     1   -1  FB_Filter
11.749   0.00  0.922   3784  18832   3.0   LogicMin 2       1     1   -1  FB_Filter
11.749   0.00  0.922   3783  18833   3.0   LogicMin 2       1     1   -1  FB_Filter
11.749   0.00  0.922   3782  18834   3.0   LogicMin 3      11    11   15  FB_Filter
11.749   0.00  0.921   3771  18819   3.0   LogicMin 3       0     0    0  FB_Filter
11.749   0.00  0.921   3771  18819   3.0   LogicMin 4       1     0    0  FB_Filter
11.749   0.00  0.921   3771  18819   3.0   LogicMin 4       1     0    0  FB_Filter
11.749   0.00  0.921   3771  18819   3.0   LogicMin 4       1     0    0  FB_Filter
11.749   0.00  0.921   3771  18819   3.0   LogicMin 4       1     0    0  FB_Filter
11.749   0.00  0.921   3771  18819   3.0      Reorder     228     0    0  FB_Filter
11.741   0.01  0.921   3771  18819   3.0       Timing      60     0    0  FF_Filter
```

```
11.741   0.01  0.922   3771  18819   3.0        Timing        1      0     0  FF_Filter
11.741   0.01  0.922   3771  18819   3.0        Timing        1      0     0  FF_Filter
11.741   0.01  0.922   3771  18819   3.0        Timing        1      0     0  FF_Filter
```

V

W

X