Manual

DPC - Data Path Compiler

MicroMagic, Inc.

Version 4.3 - Public Domain Release

MicroMagic, Inc. Public Domain Release

.

Table of Contents

About This Document

Chapter 1 DPC: Data Path Compiler And Data Path Fundamentals 1

Chapter 2 Setting Up And Initializing DPC 5

Appendix A

DPC Installation Including Timing Analyzer Interfaces 33

About This Document

This chapter provides an overview of the *MicroMagic, Inc. DPC DataPath Compiler Manual*:

Objectives

This user guide is a intended to provide detailed information about the MicroMagic, Inc. DPC DataPath Compiler software program and its options. We recommend that you go through the MicroMagic DataPath Compiler Tutorial to familiarize yourself with the tool and its functions.

Audience - DPC

This document is designed for chip designers and architects. It assumes that you have a strong foundation in IC design and development, and schematic design. This document also assumes that you are familiar with the following programming languages and operating systems: Tcl/Tk, and UNIX or Linux. This manual also assumes that you are familiar with MicroMagic's SUE Design Manager.

Organization

This tutorial is divided into parts. Each part leads you through a different aspect of the MicroMagic DPC DataPath Compiler.

This document contains the following parts:

- Preface, "About This Document" (this chapter), provides a brief description of the content and organization of this document.
- Chapter 1, "Overview", gives an overview of what you can accomplish using DPC, and the requirements necessary for running the program.
- Chapter 2, "Setting Up Directories And Initializing DPC" covers what needs to be set up in order to run DPC, and the steps for running DPC.
- Chapter 3, "The DPC Placement Algorithm and Bit-Slice Design", provides instructions on how DPC does placement, using Bit slice vs. packed gates. It then details more advanced placement techniques, including Bit Pitch, spaces, and hierarchical placement.
- Chapter 4, "Timing The Design", covers the setup for running static timing analysis.

- Chapter 5, “DPC Menus And Commands”, goes through each command in the DPC Sim Menu. All other SUE Menus are covered in the SUE Manual.

- Appendix A, “DPC Installation Including Timing Analyzer Interfaces”, goes into greater detail on how to set up DPC, including the setup for static timing analyzers.

Chapter Organization

Each part contains descriptions of the commands and methodology necessary to create datapaths using DPC.

Documentation Conventions

General Conventions

This guide uses the following text conventions:

- Statements, commands, command output, filenames, directory names, and configurations are shown in a boldface, fixed-width font. The following example shows a full path name:

  `~/mmi_private/sue/.sue`

- In examples, text that you type literally is shown in bold. In the following example, you type the word *sue*:

  To launch sue, type **sue**

- Menus, menu options, and pop-up menus are generally shown in a boldface, sans serif font. For example:

  Select `Save As` from the `File` Menu.

Conventions for Software Commands and Statements

When describing the MicroMagic, Inc. Public Domain Release software, this guide uses the following type and presentation conventions:

- Statement or command names that you type literally are shown nonitalicized. In the following example, the statement name is *set MC(default_generator)*:

  ```
  set MC(default_generator) sram
  ```

- Variables for which you substitute appropriate values, are shown in itallics. Optional commands within a string of code are often enclosed within square brackets ([ ]). Variables are enclosed within angle brackets (<>). When you type the **`setMACRO`** statement, you substitute a value for *name*, *row1* and *row2*. *CELL* is an optional command.

  `set MACRO(<`*name*`>){ [`*CELL*`] <`*row1*`><`*row2*`>...}`

Conventions for Mousing

Throughout this document, we assume you are using a 3-button mouse. When we refer to using the mouse, the following settings apply:

- Mouse-Button-1, or Button-1, refers to the left-most button (index finger) when you are using a right-handed mouse.
- Mouse-Button-2, or Button-2, refers to the middle, or center button (middle finger).
- Mouse-Button-3, or Button-3, refers to the right-most button (ring finger) when using a right-handed mouse.

  Reverse these settings if you are using a left-handed mouse, using your mouse control program, so that you click Button-1 with your index finger, Button-2 with your middle finger, and Button-3 with your ring finger.
- "Click" means to depress the designated mouse button once.
- "Double-click" means to depress the designated mouse button twice, quickly.
- "Drag" means to depress and hold down the designated mouse button while moving the mouse.

Getting Set Up

In order to run DPC, you must also have SUE installed.

Refer to Chapter 2, and Appendix A, for details on DPC installation.

Chapter 1

DPC: Data Path Compiler And Data Path Fundamentals

Introduction

In deep submicron design, wire length is the dominant factor affecting critical path timing. Cell placement becomes a crucial step in chip performance. With traditional CAD tools, designers are at the mercy of automatic placement tools. DPC lets the designer control placement with immediate timing feedback. Multiple "what-if" experiments can be performed, allowing rapid identification and experimentation with solutions to timing problems that might otherwise be very difficult to identify, not to mention fix. For data paths in particular, this methodology typically yields a minimum 2x performance improvement over automatic place and route.

Much of the performance gain in data paths is realized by ordering icons into bit slices to minimize wire lengths. Maximal packing density, which is possible with automatic placement tools, will produce much longer average wire lengths and seriously increase routing congestion compared to bit-slice data path design. Furthermore, clock routing in these structured data paths can be predetermined, which allows for excellent clock balancing and skew reduction. Control logic tends to be less structured than data paths, is more time consuming to enter and provides less speedup. However, improved clock routing and placement control makes it a viable alternative to place and route for very high-performance designs.

Data Path Compiler Foundations

The DPC flow is standard-cell based, as opposed to full-custom data path flows which are transistor based. All timing is done at the cell level. While most data paths created with DPC approach the performance and packing density of full-custom data paths, further improvements are possible by adding custom cells to the library. These new cells need only to be sized and characterized to be used by DPC.

The Standard Cell Library And Timing Models

To fully utilize the placement and timing features of DPC, you need a standard cell library complete with timing models, and access to one of the following static timing analyzers:

1. Cadence *Pearl* (Version 4.0 or later), or
2. Synopsys *Pathmill* (Version 5.3 or later), or
3. Synopsys *PrimeTime* (Version 2.1 or later).

Pearl requires timing libraries in the TLF/CTLF format, while *Primetime* requires timing libraries in the Synopsys *db* (binary) format. Fortunately, both formats can be derived from the Synopsys *lib* (ASCII) format, using *syn2tlf/tlfc* from Cadence and *dc_shell* from Synopsys, respectively. *Pathmill* can be run using Synopsys timing models for cells (this method requires the SFX license) or at the transistor level. Note that you must have the **vlog2e license** to operate *Pathmill* from DPC.

For best results, the cell libraries might need to be tuned so that cells pack well inside of bit pitches.

Placement Files

The Data Path Compiler creates both a gate-level hierarchical Verilog netlist and a DEF placement file from a hierarchical *SUE Design Manager* schematic. Most commercial routers, such as Cadence's *Warp Router* (part of the Silicon Ensemble suite), or Avant!'s *Apollo,* read this DEF format. This DEF file, complete with the Verilog netlist and cell information such as LEF, enables these tools to fully route the design.

The placement information is derived from the location of the icons in the schematic, along with bit position information (if desired). Icons in the same column in the schematic are aligned vertically and ordered by placement, or according to bit number as determined from the icon's name or *dpc property* (see below for a complete description of the placement algorithm). Since **SUE** knows the placement and connectivity, it can estimate the wire capacitance, assuming manhattan routing and a user-defined overhead factor that accounts for routing inefficiencies. In practice, this capacitance estimate is usually accurate to within 10% for typical designs which do not have significant routing congestion.

Critical Path Timing

After netlisting and placement, the designer can view the cell placement in **SUE** and quickly iterate the design by changing the schematics until the desired placement is achieved. Next, **SUE** automatically times the design using the specified static timing analyzer, and back-annotates the timing results onto the schematic. **SUE** calls the desired static timing analyzer, passing it the Verilog netlist, the estimated capacitances, and any user-defined timing constraints, and parses the output. Any particularly slow nodes are reported in the **SUE** command window. All top-level nets are then back-annotated directly onto the schematic and the critical path is highlighted. Other paths can be selected from the timing window and highlighted through all levels of hierarchy. Any critical path timing can also be displayed directly on the **SUE** placement window, in addition to the schematic.

After the desired placement and timing are completed, the DEF files created can be used to drive a commercial router to route the design and extract either capacitances or RC data in the form of DSPF or RSPF files. This post-route data can then be substituted for estimated capacitance through SUE to time the final design and back-annotate the results onto the schematics.

The Data Path Flow

This section describes the general flow which should be used when designing a data path using DPC. This assumes that DPC has been properly installed as described in Appendix A.

Before using DPC, you should first go through the SUE Tutorial and then the DPC Tutorial.

1. Using your DPC icons (ones which are associated to actual layout cells), create a schematic for your datapath. The SUE schematic can be hierarchical and can contain custom blocks such as a register file. Refer to the DPC Tutorial and "The DPC Placement Algorithm" chapter of this manual for details on how to draw your SUE schematic to achieve the desired placement.

2. Once your schematic is complete, select **DPC Netlist** from the **Sim** menu. This will create a placement view of your datapath. View your resulting placement with the **Toggle Placement View** command (hotkey: **Ctrl-a**). Make changes to your schematic until you have a good placement.

3. Once the placement looks OK, run **DPC It**. This will:

   - Create the placement file (`<cell_name>.place.def`) and Verilog netlist (`<cell_name>.vg`). These files are also created when you run **DPC Netlist**.
   - Estimate the routing distance and generate an estimated parasitic file (`<cell_name>.est_dspf`).
   - Run static timing analysis (*Pearl*, *PrimeTime* or *PathMill*).
   - Load the resulting timing information into DPC.

4. You now look at the critical paths both in the schematic and placement view. Make the necessary changes to your schematic (changing placement, sizing gates, or even changing architecture) and then rerun **DPC It**. You iterate through this flow until you have the desired timing.

5. Once the timing is good, you need to look at the congestion to see if this design is routable. Go to the placement view for this datapath and select the **Congestion** command from the **DPC Sim** menu. Refer to the **Congestion...** (page 33) command for details on how to interpret the congestion information. Make changes to your schematic, including inserting space, until the congestion looks good. You can also check the congestion before timing.

6. Once both timing and congestion for your datapath good, you now send the design to a router. Most commercial routers take the placement file (`<cell_name>.place.def`), Verilog netlist (`<cell_name>.vg`) and the cell library information as input.

With DPC, you spend your timing iterating, fixing the timing and congestion, before you send the design to a router.

Chapter 2
Setting Up And Initializing DPC

Introduction

In order to run DPC, you must already have a version of MicroMagic's' *SUE Design Manager* installed at your site. DPC is initially launched through SUE, and runs on top of the SUE program, as well an making use of SUE schematics and library files.

In addition to SUE, you must have your design library installed and ready to access. This chapter will address the directory and file structure setup, switch and variable settings, and other configurations necessary to run DPC to maximimum efficiency.

Setting Up The Data Path Compiler

Setting Pointers And Variables

Before running DPC, a librarian should set up the appropriate pointers and variables in the **.suerc** file. These pointers and variables include, but are not limited to:

- The pathname to the standard cell library,
- The routing grid size,
- The default cell heights and widths of the standard cells,
- The pathname to a file specifying the widths of all standard cells and/or heights,
- The data path pitch.

For static timing, the following variables must also be set up:

- The selection of the static timing analyzer and the command needed to invoke it,
- The pathname of the timing files,
- The average capacitance per micron of wire,
- The clock period and clock waveforms,
- The default output loading,
- The default input arrival time of inputs and departure time of outputs, etc.

Setting Grid And Pitch Sizes

Setting Grid Size

Size input to DPC is in grids, also known as *minimum routing pitches*. DPC uses the global variable **DPC(SCALE)** to convert from microns to grids. Most routers, despite their claims, generally adhere to a wiring grid to work well. The wiring pitch is often chosen to be the minimum contacted wiring pitch allowed by the design rules for the process. High speed designs sometimes use larger pitches to reduce sidewall capacitances which tend to dominate, especially with millering. Minimum metal area rules, as well as the fet pitch in the cells, also play into determining the appropriate pitch to allow stacked vias.

Setting Pitch Size

Some processes, in order to further increase contacted pitch, allow one direction to have a tighter pitch than the other direction. For example, in a common 0.18 micron process, the minimum X pitch of metal layers M1-5 might be 0.56 microns, while the minimum Y pitch is 0.66 microns. They do this by reducing the metal overlap of contact/via requirement in one direction. DPC handles a non-square pitch by accepting a pair of values for the **DPC(SCALE)** variable, setting the X and Y pitch, respectively.

For example, in the case above, you would set the pitch by typing:

```
set DPC(SCALE) "0.56 0.66"
```

If you decided to use a square pitch, you would enter:

```
set DPC(SCALE) 0.66
```

Note that if you purchase a cell library, the assumed routing grid will be specified by the library's manufacturer.

Cell sizes must be specified as an integral number of grids. If grid sizes are not integers, they will be rounded by DPC, so you cannot simply set **DPC(SCALE)** to equal 1 and design in microns unless your grid happens to be 1 micron. Why must cell sizes be an integral number of grids? Because the port locations in the cells will be off-grid. Furthermore, filler cells will not fit, cell spacings will cause lots of well and implant gaps, and well/substrate ties won't overlap (assuming the common practice of sharing well/substrate ties with cells above and below).

Setting Standard Cell Height And Width

For all placement and timing, DPC only needs to know the width and heights of the standard cells. It does not need a layout or even an abstract of the cell (for example, a LEF file). However, if you have these, you can write a script to extract the widths and heights.

Furthermore, since the heights are typically all the same (except for a few special cells), only the widths need to be specified and the heights will default to the value specified in the variable **DPC(DEFAULT_ROW_HEIGHT)**. The pathnames to the file or files containing the cell sizes are specified in the variable **DPC(PATH)**.

For example, you might have two set of files set up as:

```
set DPC(PATH) "/home/cad/stdcell/dpc/stdcell.dpc \
```

```
/home/cad/stdcell/dpc/special.dpc"
```

These files have the format:

```
# Comment lines start with a "#"
# Blank lines are ignored

<cell_name>     <width>   [<height>]
<cell_name>     <width>   [<height>]
...
```

Where the width and (optional) height are specified in grids (routing tracks).

For example,

```
# My Standard Cell Library
INVA        2
INVB        4
NAND2A      4
FAD         10  12
...
```

In this example, all of the cells except for "FAD" would get the default height.

Loading Specialized Data Path Icons

You also need to insure that the special data path icons are loaded. Note: these are NOT icons like NAND and NOR but are internal icons, like "place", that DPC requires. This can be done by adding the following line to your **.suerc** file:

```
lappend AUTOLIB $SUE_DIR/schematics/dpc
```

Alternatively, copy these icons into the `devices Library` or some other library that will be loaded.

Running The Data Path Compiler

To run Data Path Compiler (DPC), you must have a valid SUE Design Manager license installed.

For your initial startup of DPC, first load SUE.

- From the UNIX command window, type:

  ```
  sue
  ```

- Once SUE has loaded, from within the main SUE window select `Change Simulation Mode` from the `Sim` menu, then set "`Type`" to `dpc` and the "`Properties`" to `verilog`.

  Alternatively, you can set the above variables directly by addint the following lines in the **.suerc** file before starting SUE:

  ```
  set NETLIST_TYPE dpc
  set NETLIST_PROPS verilog
  ```

You can also start in DPC mode by starting SUE with the following command:

```
sue -SET NETLIST_TYPE=dpc -SET NETLIST_PROPS=verilog
```

Basic DPC Features

Netlisting And Viewing

Execute `DPC Netlist` from the `Sim` menu to automatically netlist the design to Verilog from the current cell and build the corresponding placement file (**`<cell_name>.place.def`**).

Execute `toggle placement view` from the `Sim` menu to display the hierarchical placement.

To see the contents of hierarchical blocks, select the desired blocks and then execute `push into` from the `View` menu, just as if they were hierarchical cells in a SUE schematic. To return to the hierarchical view of a cell, select any of its contents and execute `pop out of` from the `View` menu.

The features above do not require a static timing analyzer or any standard-cell timing information or setup.

Timing A Design

To time the design, select `DPC It` in the `Sim` menu which will first run `DPC Netlist` and then run timing and back-annotate the results. Alternately, you can select `Time It` from the `Sim` menu. `DPC It` is just a shorthand for running `DPC Netlist` followed by `Time It`. However, `Time It` will also bring up a window listing various options and allow you to change any defaults, such as the capacitance file to use, before proceeding.

At the completion of the static timing analysis, all nets are back-annotated with timing values, and the worst critical path in the schematic is automatically selected. To view other critical paths, click on the radio button for the desired critical path and then click on `Display CP`. You can also push into sub cells in the critical path to view them, or go to the placement view and click on `Display CP`.

When you select `Display CP`, in addition to the timing, SUE annotates the rise or fall time on the critical path. You won't see the rise/fall times immediately after running `DPC It`, because you are seeing all the timing. To view the rise/fall times, click on `Display CP`.

Displaying A Specific Instance

It is often desirable to locate a specific instance from the schematic view in the placement view, or the reverse. Select the appropriate instance in either view and execute `Display Other View` from the `Sim` menu. This will toggle views and select the appropriate icon. If the icon is not visible in the placement view because it is in an unexpanded cell, a temporary icon will be displayed.

Displaying Connections

In the placement view, it can be very helpful to know all the connections to an instance before deciding to move that instance. Select the instance in the placement view, and then execute `Display Connections` from the `Sim` menu. This will show all the connections to that instance with flylines. If a hierarchical instance is selected, then all connections to any cell in that hierarchical instance are displayed.

Congestion Analysis

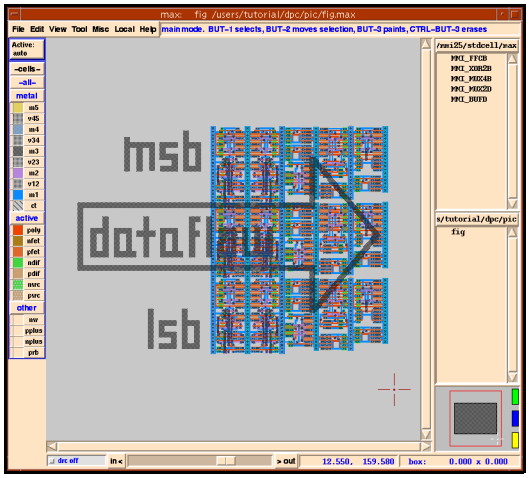
Before you send a design to a detailed router such as *Silicon Ensemble* or *Apollo*, you want to make sure the design is routable. After either `DPC It` or `Time It` have been run (both run the DPC global router), you can execute `Congestion...` from the `Sim` menu. Refer to `Congestion...` on page 33 for more details on the congestion analyzer. If you find areas which are congested, you can add space for routing as described in `Adding Spaces` on page 21.

Chapter 3
The DPC Placement Algorithm

Overview

The placement of icons on the schematic directs the relative placement in the placement file. DPC assumes that data flows horizontally and builds the placement from the top (See Figure 1), which is the most significant bit (MSB).

Figure 1: DPC Data Flow



Don't try to build data paths with data flowing from the top of the page to the bottom or with the LSB on the top. If you do, you will be greatly disappointed.

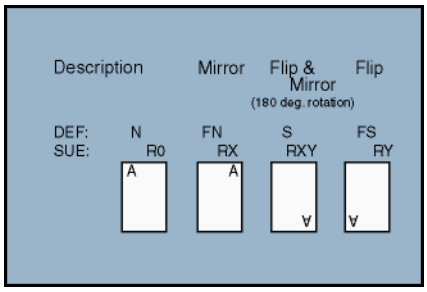
Most cell libraries are oriented so that the top of the cell has a horizontal VDD supply and the bottom has a horizontal GND supply. In this orientation, the cell heights are fixed but the cell widths vary depending on the function and power level of the cell. In the DPC flow, cells are ROTATED 90 degrees. Therefore schematics oriented by columns in DPC will end up as rows in the DEF placement file.

Note that special libraries can also be built with different height cells to better implement certain functions. For example, better packing efficiency can be achieved by building a special full-adder cell for a multiplier. However, you must only place cells of the same heights in the same columns. DPC does not check for this violation.

The Data Path Compiler automatically adjusts the rows of cells in the final placement file so that they alternate between flipped and not flipped. This enables the rows to be abutted so that the VDD supplies and the GND supplies overlap. Therefore, if you change the schematic so that you add or subtract a single column of cells, all subsequent cells will get automatically flipped in the placement from the previous run. You can see this by looking in the placement view (Select `toggle placement view` in the `Sim` menu). The small triangle in each placement icon signifies its orientation.

DEF files allow four possible orientations of cells: N, FN, S, FS as shown in Figure 2. 90 degree rotations are not allowed. N corresponds to the orientation of a cell without any flipping or mirroring. Cells flipped so that their power is at the bottom of the cell and their ground at the top are FS. This differs from rotating a cell 180 degrees, which gives you an S orientation. In DPC, we define flip as mirroring the cell around its X axis and mirror as mirroring the cell around its Y axis:

Figure 2: DEF Cell Orientations



The Basic Placement Algorithm

(Make Sure You Totally Understand This!)

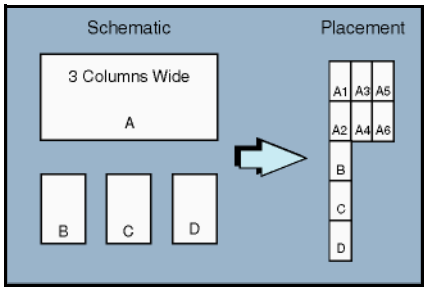
The first stage of DPC consists of breaking the entire design into columns. All icons whose boundaries overlap with other icons in the same column (even a little) are placed in the same column. It is wise to leave a definite space between columns so that there is no confusion. All text (including the text in the icons), wires, arcs, lines, or icons that don't affect placement (like I/O's or title bars) are ignored during this step.

Once the icons in a column are determined, DPC will place the icons beginning from the left most column and proceeding to the right. The icons in each column are placed from top (MSB) to bottom, sorted by the vertical location of their origins (the little "+" in the icon view of the cell).

The Data Path Compiler is fully hierarchical. If you place down an instance of a 16-bit adder, for example, all of the sub cells of that adder will be placed starting from the given position. This is not quite the same as if you had one multiple-column cell that was the adder. In that case, columns of the cell wouldn't be flipped automatically and thus you couldn't stack other cells above or below it (it would be a special cell with a different cell height).

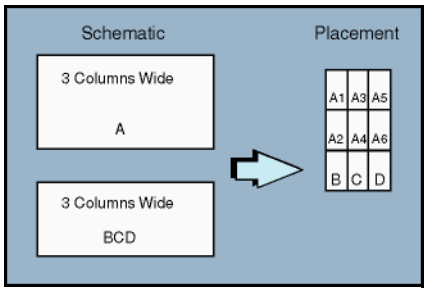
If you place a hierarchical cell that has 3 columns of cells in it, for example, and you want to place a cell under each of the columns, you can't simply do the following:

Figure 3: Hierarchical Placement - Example 1



Instead you must create a new hierarchical cell which contains the three cells and then stack the two different hierarchical cells:

Figure 4: Hierarchical Placement - Example 2



If you want to interleave two (or more) columns on the screen, place a "`row_spanner`" (think column spanner — it's a row in the layout because of the 90 Degree rotation) icon under the columns. This is just a simple empty generator whose width can be adjusted so that it will span multiple columns.

If none of the icons in a column have any **`bit specifiers`** (bus notation) or **`dpc properties`** (see below), then the icons will be packed flush against each other starting from the MSB (top). No bit slices will be preserved. This mode is useful for packing small bits of control logic either by themselves or in and around data paths.

Bit Slice Design

(You Better Understand This, Too)

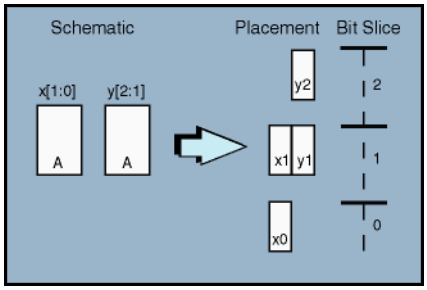
In most data paths, cells are placed in bit slices and do not overlap into adjacent bit slices. The data wires flow horizontally, confined to a single bit slice which keeps wires short and makes circuits fast. A notable exception is in logical or arithmetic functions which contain trees or other combining functions, like carry look-ahead adders. In these cases, cells must be placed in a more free-form arrangement, often crossing bit slices. In addition, the minimal control logic and buffering that drives rows of flip-flops, muxes, etc. is often bit slice independent.

Bused icons (icons with `[max:min]` or just `[num]` in their name) are expanded into the appropriate number of individual icons and automatically placed in the bit slice associated with their bit number. For example, an icon with the name `[3:2]` will be expanded into two icons, `[3]` and `[2],` and placed in bit slices 3 and 2. Thus, the icon's name serves two purposes:

- It specifies how many instances are bused together (1 or more) AND
- It specifies in which bit slices to place the icon.

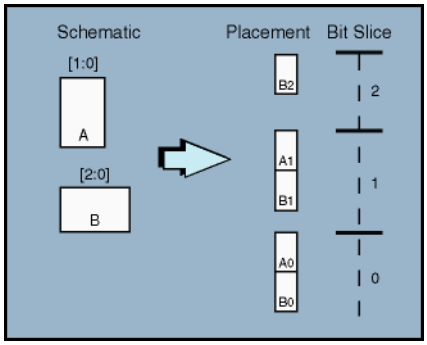
Note that the bit slice position can be overridden with the `"dpc" property` (see below). If any icon in a column has a bit specifier, then the bit slices of that column will be aligned to the same bit slices of all other columns. Therefore, icons with name [1] will be aligned with all icons in all columns with the name [1] on that schematic. Consider the following example:

Figure 5: Bussed Icons



If you place multiple bused icons in the column with overlapping bit slices, then the icons will be interleaved. Furthermore, the icon whose origin is closer to the top of the schematic will be placed first. For example:

Figure 6: Bussed Icon - Example 2

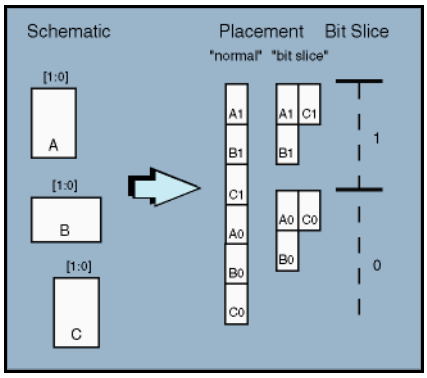


What happens if you interleave enough cells in the same bit slice that they don't fit? The Data Path Compiler has two ways of dealing with this situation:

- Let the cells overflow into the next bit slice, OR
- Force overflow cells into adjacent columns to the right in the same bit slice.

Case 1 is considered the "normal" mode and case 2 is called "bit_slice". These modes are determined globally by the variable **DPC(MODE)** and they can also be adjusted on a schematic by schematic basis by adding the "dpc_control" icon and specifying the "mode". Below is an example that shows the different placements that result under the different modes:

Figure 7: DPC Placement Modes



Mixed Bit Slice Design

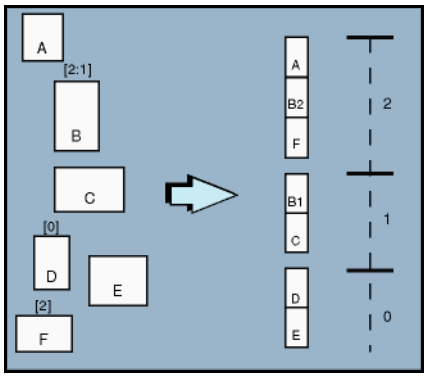
(And This One)

What happens when you mix bused icons with non-bused icons in the same column? This happens all the time. All icons without a bit specification will be placed in the "current" bit location. Remember that the icons are being placed on a column by column basis starting from the icon whose origin is closest to the top and ending with the one whose origin is closest to the bottom of the schematic. The "current" bit location is either the bit slice where the last icon was placed in the column or, if none, then the highest bit slice in the entire schematic (i.e. the top of the placement).

To override this rule, simply add a bit slice identifier to the icon with the desired bit location, such as **[3].** This can be either placed in the name field or in the dpc property field for people who don't like to clutter up the name. Note: a bit specifier in the dpc field overrides the name field.

The following example should make this crystal clear:

Figure 8: DPC Placement Example



Getting Tricky

Sometimes a bus with more bits than the rest of the data path is needed, for example, from the output of a multiplier. To get this bus into the width of the rest of the data path, you want to double up the bit slices. The "dpc" property on standard cell icons is set up for just that and other non-standard cases. To specify the algorithmic bit slice positioning of a bus, use the syntax:

```
p[max:min]=f($b)    or
p[bit]=f($b)        or more simply
p=f($b)             or more simply
f($b)
```

where "$b" will be substituted for the actual bit number. NOTE: spaces are NOT allowed in expressions. Also, you can use the most simplified last form only when the expression includes a "$b" (see below). For example, to fit a 64 bit bus ([63:0]) into bit slices 31 to 0, use the following expression:

```
$b/2
```

To flip the order of an icon with the size [15:0], use:

```
15-$b
```

to place all three inverters of [2:0] at bit slice position 5, use:

```
p=5
```

Note that in this example you would need to use the "**`p=f($b)`**" syntax because it doesn't include a "**`$b`**", i.e., just putting a "5" in the **`dpc property`** won't work.

If you want to specify some bits with a different algorithm than others, you can do that also, for example:

```
p[7:4]=7 p[3:0]=0
```

which will put the top half of the icons in the topmost bit position and the bottom half into the bottommost bit position. You could have specified the bits completely, also:

```
p[7]=7 p[6]=7 p[5]=7 p[4]=7 p[3]=0 p[2]=0 p[1]=0 p[0]=0
```

Note that if you don't specify any bits or ranges of bits, they will just get their normal default bit locations.

You can also specify what rows to place the icons of a bused icon in, using the following syntax:

```
r[max:min]=f($b)    or
r[bit]=f($b)        or more simply
r=f($b)
```

Thus, if you want to stack a bus in half the height but in two rows, you might use the following syntax:

```
p=$b/2 r=$b%2
```

where the "%" is the modulo operator in TCL (sometimes called the remainder operator). If you want to array an icon laterally instead of vertically, use this:

```
p=0 r=$b
```

Note that the row syntax is more restrictive than the pitch syntax: you can't specify negative rows and you can't skip rows.

The pitch or height of each bit position is predetermined by the designer, based on the technology and cell library. This pitch is specified in the **.suerc** file in terms of grids or routing tracks by the variable **`DPC(PITCH)`.** The bit pitch for a single schematic can be overridden by adding the `"dpc_control"` icon and specifying the **`"pitch"`.**

Adding Spaces

Sometimes, you want cells to be spaced apart from each other in ways different than the simple bit slice methodology. DPC provides this capability with "spacer" cells and by adding to the dpc properties of icons.

The spacer cell, which is in the `"dpc" Icon List` box and is a special cell, allows you to add arbitrary horizontal and/or vertical space to a design. Think of this icon as a cell whose size can be adjusted on the schematic (but is empty). If you wanted to added some space between the two rows of your design, place the spacer icon in the desired spot and set its "rows" line to, for example, "2r" or "2 rows". This will add two entire rows of space (or two columns in the placement view) which might be needed to improve the routability of a particularly congested area. If DPC doesn't see an "r" in the string, it will default to grids, which allows for finer control. Note that some routers, like Gambit's, do not handle rows spaced by anything other than complete rows.

Spacer cells can also be added in between icons in a given column to provide extra space between icons. Simply change the "col" value to the desired number of grids of space. If DPC sees a "p" in the value, as in "2p" or "2 pitches", DPC will add that many pitches of space.

A more handy way of adding space between icons in the same column is by adding "newbit" (shorthanded to "--") to the dpc property of an icon without a bit specifier. This adds as much space as needed (if any) to start the icon in the beginning of a bit pitch. This property is very useful in lining up the internals of adders and other functional blocks.

Finally, you can shift an icon up or down a specified number of grids by adding an offset to its dpc property like "+3" or "-2". Be warned that you can force DPC to overlap cells by using a negative offset. This can be useful if your standard cell library is set up to allow this.

Embedding MegaCells into Data Paths

Often, data paths are interrupted by large array functions (called MegaCells) like register files or fifos. While the data paths can be broken and these functions can be added at the top level of the chip, embedding them into data paths is often desirable. MegaCells are different than standard cells or even special cells for the following reasons:

- Because they are large, we can't approximate the ports as all being in the center of the cell and hope to get reasonable estimated capacitances.
- They can't be arbitrarily flipped because they don't look like standard cells and because their port locations will move drastically, affecting timing and routability.

For example, if a data path is interrupted by a fifo and the fifo is designed to take input from one side and create output on the other, then you definitely want to orient the fifo correctly in the data path.

Ports Files

DPC uses "ports files" to specify the actual port locations of a cell. Ports files also contain desired orientation and bounding box information. A ports file must have the same path name as the DPC cell but have the suffix "**`.ports`**" instead of "**`.sue`**". (These suffixes are the recommended defaults but can be changed in the **.suerc** file.) For example, if you had the fifo located in the file:

```
/home/proj/fast_chip/sue/fifo64.sue
```

then the ports file for this fifo would need to be in the file:

```
/home/proj/fast_chip/sue/fifo64.ports
```

Ports files will always override other data such as the bounding box size given in the files pointed to by the variable **`DPC(PATH)`**.

Ports files have the following format:

```
# Comment lines start with a "#"
# Blank lines are ignored

bbox width [height]

[orient N|FN|S|FS]
[flip 0|1]

port port_name x y
port port_name x y
...
```

The bbox line specifies the bounding box of the cell in grids and, the port lines specify the coordinates of each port (also in grids). If any ports are not included, they will default to the center of the cell.

If no "orient" line is given, then DPC will assume that this cell is like a standard cell and will alternate flipping it on a row by row basis. This is useful for overriding the bounding box size from DPC(PATH).

If the "flip" line is supplied, DPC will use the value to determine if the next row is flipped or not. The default is "1" which means that the next row after this cell has an orientation that is flipped from this row. If given a "0" then the next row will has the same orientation as this row. This is useful for double height rows, for example.

If the orientation of the cell is given in the ports file, that orientation will be passed, independent of the row, to the placement file. The user can, however, specify other orientations by adding the keywords **mirror** and/or **flip** to the dpc property of any instances of the cell.

Verifying that a MegaCell has the correct orientation is easy by looking in the placement file. Any selected cell that has specified port locations from ports files will display those ports when you do a `show sel. term names` command from the `View` menu.

The task of making a ports file depends on the state of the cell. If the cell is already layed out, an automated way of creating the ports file is desirable. MicroMagic has a simple script called **`lef2ports`** which parses the LEF description of a cell and creates a ports file from it. Otherwise, you need to write it by hand or translate it from some other format.

Chapter 4
Timing the Design

Much of the power of the Data Path Compiler involves timing a design for critical path optimization. How does this differ from taking the output of logic synthesis, or any other netlist, and running it through a static timing analyzer? One word: placement. DPC knows the placement of the cells and the interconnect. Therefore, it can estimate all wire capacitances very accurately. Otherwise, you need to either use a wire-load model, which is only a statistical approximation, or run place and route and extraction. The latter takes your iteration time from about once a minute to once a day.

DPC runs static timing analysis for you and back-annotates the results onto the schematic, which makes timing a design a breeze. Most people who use the DPC Data Path Compiler have previously run a static timing analyzer only infrequently if ever. Yet, since DPC handles the interface, you don't need a manual for most tasks.

Combinational vs. Sequential Circuits

If you are timing a purely combinational circuit, you tend to be interested in different things than if you are timing sequential circuits. Purely combinational circuits have no state (that is, no flip-flops or latches) and are typically functions like adders, shifters, etc. These are usually the building blocks of larger, sequential circuits.

In a purely combinational circuit, you care about the maximum delay from any input to any output, or the *critical path*. You usually don't worry about if a given input is earlier than another since this block may be used in multiple instances in the design.

In a sequential circuit, you are mostly interested in meeting a given cycle time from any register to any other register. Inputs or outputs must be treated specially (see next section). You need to specify what clocks are in the design.

For sequential circuits, DPC needs to know all of the clock names in the design, which are kept in the variable **`DPC_TIMING(clk_names)`** and the desired cycle time (minus clock skew/jitter) which is kept in the variable **`DPC_TIMING(clk_period)`**. You can also change these on the fly by running `Time it` from the `Sim` menu. Note that DPC sets the design up for "ideal clocks", meaning that the clock inputs to all flip-flops arrive at exactly the same time. You can override this with the "**`.constraint`**" file.

When timing a design, DPC looks at all of the primary inputs to the design. If none of them match the predetermined clock names, then DPC assumes you have a purely combinational block and sets up the timing accordingly.

Setting I/O Timing Constraints

An entire design can rarely be captured in a single static timing analysis and hence external I/O timing must be identified. In most cases, this involves specifying the input arrival times of external inputs and the output setup and hold times of external outputs.

Since no signal will likely arrive at time zero, but more likely, at the minimum, a clk-to-q delay from a register, the DPC variable **`DPC_TIMING(arrival_time)`** is used to specify the default arrival time for all inputs to sequential logic. So, for example, if you want all inputs to default to a 1ns arrival time:

```
set DPC_TIMING(arrival_time) 1ns
```

Similarly, outputs need to go somewhere, which, at a minimum, is usually into a flip-flop which has a setup time. The default departure time is specified by the variable **`DPC_TIMING(departure_time)`**. The value is with relation to the next edge of the clock. Thus if you assume that outputs from blocks can go anywhere on the chip, you might want to give them, say, 3ns to get there, which would mean setting:

```
set DPC_TIMING(departure_time) 3ns
```

Typically the arrival times and departure times should be set larger than these minimums to account for wire delay and loading. Furthermore, unless you use a design style that prohibits logic on wires between blocks, you might want to add additional suitable logic delays.

The .constraint files are used to set timing constraints on individual or groups of I/O signals, overriding the above defaults. A timing constraint file for a schematic should have the same name as the schematic, be in the same directory, and have the suffix "**`.constraint`**". This name can be overridden in the "`time it`" command in the "`Sim`" menu.

The timing constraint file is merely included into static timing analyzer during the timing analysis and thus must observe either *Pearl*, *Pathmill*, or *PrimeTime* syntax and semantics—depending on which one you are using. The syntax is different depending on whether you are timing a purely combinational block (no clocks) or a sequential block (with clocks).

When using *Pearl*, for each input constraint in a purely combinational block, add a line of the following form to the **`.constraint`** file:

```
input <signal> _input_ ^ <rise_min> <rise_max> <fall_min>
<fall_max>
```

For each input constraint in a sequential block, add a line of the following form to the **`.constraint`** file:

```
arrival <signal> wave ^ <rise_min> <rise_max> <fall_min> <fall_max>
```

and for each output constraint:

```
departure <signal> wave ^ <rise_hold> <rise_setup>
<fall_hold><fall_setup>
```

Note that "**`wave`**" is the default waveform as specified by the **`DPC_TIMING(wave_name)`** variable in the **.suerc** file

For example (sequential):

```
arrival cin wave ^ 1n 1n 1n 1n
departure bus* wave ^ 0n 2n 0n 2n
```

Default units are specified in the technology file and thus it is a good idea to put them in (for example, **p** or **n**).

Wildcarding "*" is permitted in signal names.

The "**arrival**" command can also be used to trick *Pearl* into ignoring certain multicycle paths, such as ones from asynchronous resets. For example, by setting the following:

```
arrival reset_L v -2n -2n -2n -2n
```

Pearl will assume that the signal "**reset_L**" arrives very early and hence does not create any critical paths in the design. Note that *Pearl* also has specific commands for identifying multicycle paths.

Consult the *Pearl* manual for more information and options. Alternatively consult the *Pathmill* or *PrimeTime* manual for similar commands and syntax relating to it.

Input Drivers and Output Loading

In addition to specifying the input and output delays, we must specify how to drive the inputs and what loads to place on the outputs. DPC has two ways of specifying the input driver: fixed transition time, or driver cell.

To drive all primary inputs with a specific driver cell, use the variable **DPC_TIMING(driver_cell)**. The format of this variable is a list of the driver cell name and the output port, for example:

```
set DPC_TIMING(driver_cell) "INVC out"
```

would set the input drive strength to be the output of the **INVC** cell. If no driver cell is specified, DPC defaults to using a fixed input transition given by the variable **DPC_TIMING(input_transition)**, for example:

```
set DPC_TIMING(input_transition) 200ps
```

Note that if you use a fixed input transition you can sometimes underestimate timing because you have loaded your inputs too heavily. Also note that *Pathmill* does not allow a driver cell and thus DPC defaults to using a fixed input transition.

Primary outputs automatically get annotated with a default capacitance given by the variable **DPC_TIMING(out_cap)**. For example, to place 20fF of capacitance on all primary outputs, use the line:

```
set DPC_TIMING(out_cap) 20fF
```

Both the input drivers and output capacitances can also be specified on a net by net basis with the .constraint file.

Chapter 5
DPC Menus And Commands

DPC Menus

This section describes each menu item for the DPC Data Path Compiler, and covers only the DPC `Sim Menu` items. For all other menus and commands, refer to the SUE User Guide.

A brief description of each command is displayed in the SUE `Message Area` when the cursor is over the menu item. Keep in mind that DPC runs on top of the SUE Design Manager, so the main window will display SUE properties and labels. You know DPC is running when you see the `(dpc)` label at the end of the title bar in the main window.

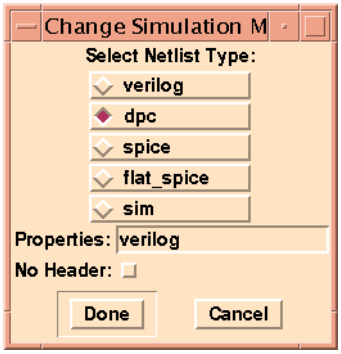
Sim Menu

The `Sim Menu` controls all the commands for Netlisting, Simulating, and Plotting of nets, as well as creating and manipulating Verilog properties, and cross-probing with the MAX Layout Editor.

The `Sim Menu` changes depending on which *Simulation Mode* you have selected. This section addresses the `DPC Simulation Mode` menu and commands.

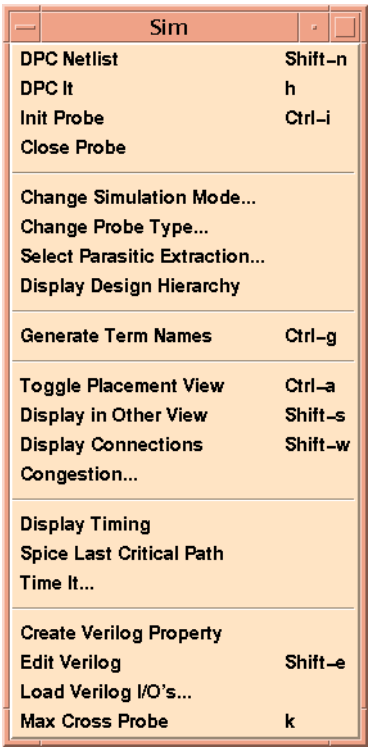
Change Simulation Mode: "dpc"

Figure 9: "dpc" Netlist Type Selected in Change Simulation Mode Pop-up



From the `Change Simulation Mode` pop-up, selecting the option "`dpc`" accesses the `Sim` menu shown in Figure 10.

Figure 10: The "DPC" Sim Menu



`DPC Netlist` *Hotkey:* ***Shift-n***

Write out a Verilog netlist (`.vg`) and a DEF placement file (`.place.def`). You can now look at the placement view of the cell.

`DPC It` *Hotkey:* ***h***

This command does the following things.

- Writes out a Verilog netlist (`.vg`) and a DEF placement file (`.place.def`).
- Runs a Steiner route to compute wire lengths, calculates parasitic information for the wires and outputs an estimated parasitic file (`.est_dspf`). DPC also creates a control file (`.timing_in`) for the static timing analyzer. What's in this file depends on which timing analyzer you are using.
- Next, static timing analysis (*Pearl*, *PrimeTime*, or *PathMill*) is run on this design. Two output files are generated: the output timing file (`.timing_out`) which contains the information on critical paths, and the slow nodes file (`.slow_nodes`).
- Finally, the results of timing analysis are loaded into DPC and the `DPC Timing` window is brought up.

`Init Probe` *Hotkey:* ***Ctrl-i***

Does not do anything in DPC mode.

`Close Probe` *Hotkey:* ***None***

Does not do anything in DPC mode.

`Change Simulation Mode`

Hotkey: ***None***

Brings up a pop-up where you can change the simulation parameters for `Netlist Type` and `Properties`, as shown in Figure 9.

`Change Probe Type...`

Hotkey: ***None***

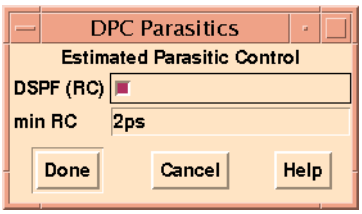
There are no simulation probes in DPC mode.

`Select Parasitic Extraction`

Hotkey: ***None***

Selecting this will open a `DPC Parasitics` pop-up menu, as illustrated in Figure 11.

Figure 11: DPC Parasitics Setup Pop-up



`DSPF (RC)`: If `DSPF` is not selected, DPC creates a capacitance wire load estimate for each net; otherwise it creates an RC network. NOTE: You must use at least the 2000.11 release of *PrimeTime*.

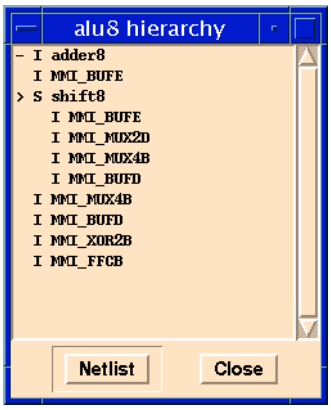
`min RC`: If `DSPF` is selected and "min RC" is greater than zero (0), only nets with RC time constants greater than the minimum RC value given will get complete RC networks. The others will have capacitances only, since the R's are unimportant. This feature simply reduces file sizes and run times.

`Display Design Hierarchy`

Hotkey: ***None***

This command displays the design hierarchy of the currently selected cell in a small information window, as shown in Figure 12. This window not only displays the design hierarchy, but more importantly, allows you to control where behavioral models should be used. If there is no underlying schematic for an icon, SUE automatically looks for the behavioral model, which can either be included in the icon, or in a separate text file. If the window is blank, click on `Netlist` to netlist the schematic, and display the hierarchy information.

Figure 12: Display of Design Hierarchy



By default, SUE stops and uses a non-SUE generated model (such as a behavioral model) when it finds the Verilog property (**`-type auto -name verilog...`**) in the icon for the cell. This can be changed in two ways.

- First, in the `Display Design Hierarchy` pop-up, you can have SUE look into a lower level schematic for netlisting. If you see "**`-I`**" in front of a cell name, this means that the icon view is being used for netlisting (therefore, the behavioral model) but that there is an underlying schematic for this cell.

  In the example in Figure 12, **`adder8`** is such a cell. If you click with Button-1 on the cell in the pop-up and then click on `Netlist`, SUE will use the schematic for that cell when netlisting. In Figure 12, **`shift8`** is a cell where the schematic for the cell is used for netlisting, instead of the behavioral model.

- The second way to control netlisting is by adding the following property into the icon for a cell.

  ```
  -type fixed -name netlist_level -text 2000
  ```

  This will ignore the Verilog property on this icon for netlisting, and use the underlying schematic instead.

> **Tip** All instances of a cell must be netlisted the same way.

`Toggle Placement View`

Hotkey: ***Ctrl-a***

This feature toggles between the DPC placement view and the SUE schematic view. You must first run `DPC Netlist` before being able to utilize this option.

`Display Other View`

Hotkey: ***Shift-s***

If in schematic view, DPC will switch to the placement view and highlight in the placement view the cell which was selected in the schematic. If in placement view, the selected cell will be highlighted in the schematic view.

Display Connections

Hotkey: ***Shift-w***

Draws fly-lines showing where the selected cell is connected to other cells in the schematic.

Congestion...

Hotkey: ***None***

The DPC congestion analyzer runs a global router on the DPC design and displays the results over the DPC placement view. The global router routes on a global grid whose size you can define. Typically, you want to choose the grid size to be near the bit pitch of the datapath by some number of row heights (both in tracks). The global router will not work well if you make the global grid too small (for example, 1x1). Also, the global router doesn't do layer or track assignments within the global grid.

You must start the global router from a DPC placement view.

You also need to specify the number of routing resources per direction per track. For example, if M2 and M4 are horizontal (DPC direction) and you want to allow the global router to use both metals, you would specify 2 horizontal routing resources per track. If you also had M6 but it was at twice the pitch, you would specify 2.5. If your power grid consumed 20% of M2/M4/M6, you would specify 2.5 * 0.8 = 2.

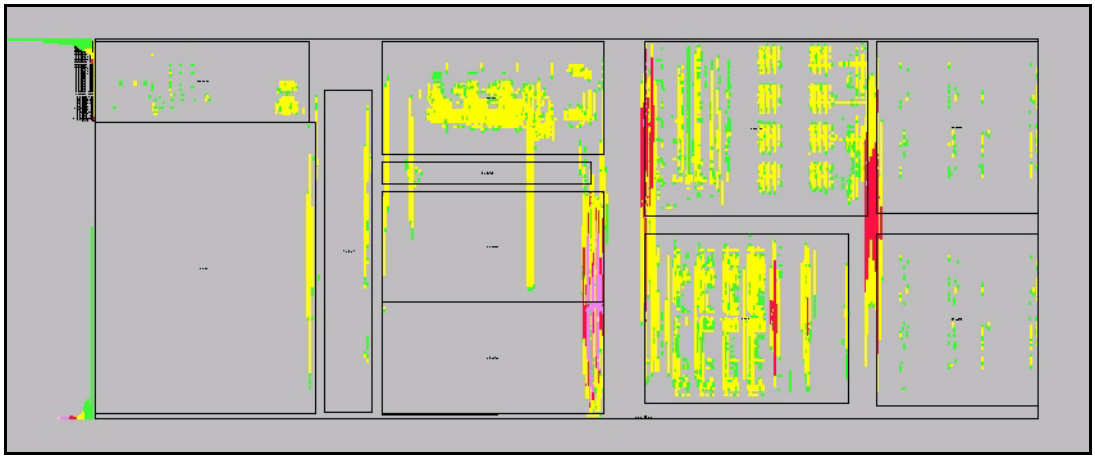
When displaying the congestion, you need to specify the threshold fraction. Any global grids with congestion below this threshold will not be displayed, which makes the displaying faster and less cluttered. If you specify the threshold as 1.0, then only the global grids that are over 100% congestion will be displayed.

The congestion display uses colors and line widths to signify congestion. Routes that are <50% congested are shown in green; between 50% and 100% in yellow; between 100% and 150% in red; 150% to 200% in violet; 200% to 250% in pink; and over 250% in white.

Even if a design has areas that are over-congested, it may still be routable. However, the router will have to bypass the congested routes which will likely significantly increase wire lengths over the ideal Steiner lengths.

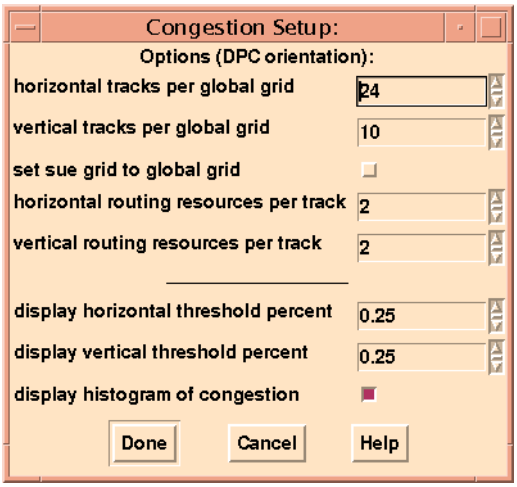
The example shown in Figure 13 is from a design with 106K placeable instances. The global router took 12 seconds to compute the route and another few seconds to display the congestion. You can see three main areas of high congestion. Since the areas around these congested areas have very low congestion, a router can still route this block, albeit with longer wire lengths in these areas.

Figure 13: Congestion Analysis Example



When you are ready to run congestion analysis, click on `Congestion...`, and the pop-up menu shown in Figure 14 will appear, with the following variable settings:

Figure 14: Congestion Setup Pop-up



`horizontal tracks per global grid:` Size of horizontal global grid (rotated from DEF/layout orientation).

`vertical tracks per global grid`: Size of vertical global grid (rotated from DEF/layout orientation).

Remember that the DPC orientation is rotated 90 degrees from the DEF placement file. So, for example, if you will be routing M2 and M4 vertically in your router using the DEF placement, M2 and M4 will be shown horizontally in the DPC view.

`set sue grid to global grid`: Sets the SUE grid to be the same as the global grid to aid in display.

`horizontal routing resources per track`: Metal tracks available for routing in the horizontal direction (rotated from DEF/layout orientation). Typically, the number of vertical metal layers minus overhead for power, clock, etc.

`vertical routing resources per track`: Metal tracks available for routing in the vertical direction (rotated from DEF/layout orientation). Typically, the number horizontal metal layers minus overhead for power, clock, etc.

`display horizontal threshold percent`: When displaying horizontal congestion, don't display anything if the congestion is below this threshold. The threshold in the percentage (/100) of the resources. Making this number greater than 0, makes the display much faster and, sometimes, makes the output more readable.

`display vertical threshold percent`: When displaying vertical congestion, don't display anything if the congestion is below this threshold. The threshold in the percentage (/100) of the resources. Making this number greater than 0, makes the display much faster and, sometimes, makes the output more readable.

`display histogram of congestion`: Displays a histogram of horizontal congestion and vertical congestion to the left of the design. The histogram is another method for visually evaluating congestion problems. There are five bars in the histogram for each color (in the congestion range), with the height of each bar representing the number of routing grids in that range. For example, the five red bars represent routing grids that are 100-110%, 110-120%, 120-130%, 130-140%, and 140-150% congested.

`Display Timing`

Hotkey: ***None***

A pop-up will open which contains a list of critical paths and selection buttons for back-annotating.

`Spice Last Critical Path`

Hotkey: ***None***

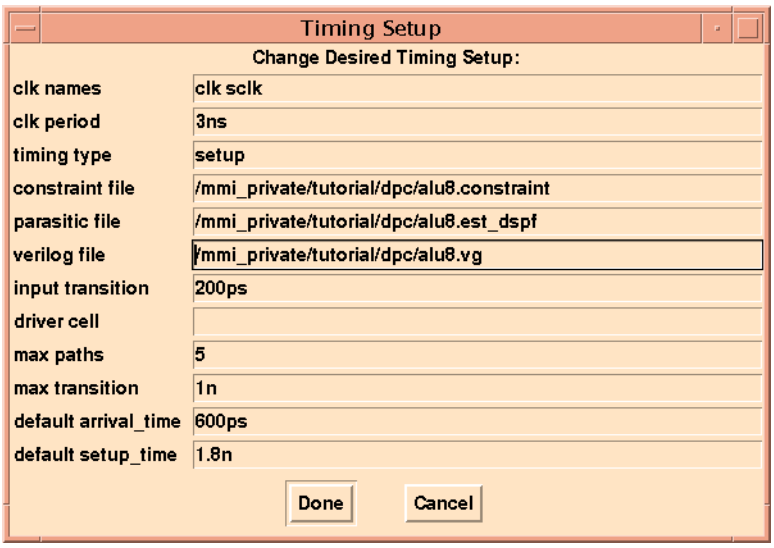
This option will run a SPICE simulation on the last critical path highlighted, and report the results in a small information window.

`Time It...`

Hotkey: ***None***

A pop-up window will appear, as shown in Figure 15, with several `Timing Setup` options.

Figure 15: Timing Setup Form



When these options are set, click on `Done` and DPC will do the following:

- Run a Steiner route to compute wire lengths, calculate parasitic information for the wires, and output an estimated parasitic file (**`.est_dspf`**). DPC also creates a control file (**`.timing_in`**) for the static timing analyzer. What's in this file depends on which timing analyzer you are using.
- Next, run static timing analysis (*Pearl*, *PrimeTime*, or *PathMill*) on this design. Two output files are generated: the *output timing file* (**`.timing_out`**) which contains the information on critical paths, and the *slow nodes file* (**`.slow_nodes`**).
- Finally, the results of timing analysis are loaded into DPC and displayed in the DPC Timing window.

`Create Verilog Property`

Hotkey: ***None***

This option automatically creates a "verilog-netlist" text string for an icon. You can modify the text by clicking on it with Button-1, using the Backspace or Delete keys to erase text, and then type in your changes. Click Button-2 over the text to bring up a selection box for text *size*. Click Button-3 to bring up a selection box for text *orientation*.

`Edit Verilog`

Hotkey: ***Shift-e***

This brings up a window with the Verilog **`.vb`** file for the selected instance. If nothing is selected, then it displays the **`.vb`** file for the current cell. If there is no Verilog file, an error message will appear.

`Load Verilog I/O's...`

Hotkey: ***None***

A file window will open where you can select the name of a textual Verilog file. Click `Done`, and the Verilog I/O's will be placed into the current SUE schematic.

Appendix A
DPC Installation Including Timing Analyzer Interfaces

Basic DPC Installation

Below is a suggestion for the directory structure for the files needed to run DPC. You can set up the files any way you want. The important thing is that you have specified all the needed variables and files for DPC. An example setup is part of the DPC Tutorial. Please note that tnot all of the files listed below are required, only those that apply to the static timing analyzer you are using.

The **.suerc** file is the same one used by the SUE Design Manager, with the addition that it sources **$MMI_TOOLS/../mmi_local/mmi25/library/dpc/dpc.suerc**. The **dpc.suerc** file is what sets up all the things needed to run DPC and timing analysis.

```
mmi_local
sue
    .suerc
    default_spice_header.h
    default_verilog_header.h
    mmi25
        spice (for PathMill only)
            default_spice_header.h
            mmi25.l (spice models)
        stdcell
            spi
                include_params
                MMI_BUFB.sp
                ...
            sue
                MMI_BUFB.sue
                ...
        library
            dpc
                dpc.suerc
                mmi25.dpc (not needed if you have LEF for all cells)
            lef
                mmi25.lef
            pathmill
                include_params
            pearl (for Pearl only)
                pearl.tech
                mmi25.ctlf
            synopsys (for PrimeTIme only)
                mmi25.db
                mmi25.lib
            verilog
                mmi25.v
```

There is an example of a **dpc.suerc** file as part of the DPC Tutorial located in:

```
tutorials/dpc/mmi25/library/dpc/dpc.suerc
```

The directory structure of the DPC Tutorial is set up to be a self-contained environment which doesn't require any of this setup.

The DPC setup files should generally be located in the **mmi_local** directory. The **dpc.suerc** file contains all of the SUE startup data required to run DPC. Look at the **mmi_local.sample** directory for an example of the directory structure, and the example **dpc.suerc** file. Look at the **dpc.suerc** file. If you want to run *PrimeTime*, instead of *Pearl*, you simply uncomment the following line:

```
#set DPC_TIMING(simulator) "primetime"
```

You also need to comment out the *Pearl* line. We'll talk more about the timing analyzer setup later. The **dpc.suerc** file must be sourced when you start up SUE. This can be done in a number of ways. One way is to add the following lines to the **.suerc** file in **$MMI_LOCAL/sue**, or to your home or current directory **.suerc** files.

```
set tech <dir>
source $tech/library/dpc/dpc.suerc
```

Alternatively, you can add the following lines to **$MMI_LOCAL/sue/.suerc** or another **.suerc** file:

```
switch [use_first PROJECT env(PROJECT)] {
  mmi25 {
    set tech <dir>
    source $tech/library/dpc/dpc.suerc
    puts "Setting up for DPC with mmi25 library ..."
    }
  }
```

With this, you can load the DPC example, and invoke SUE by typing:

```
sue -SET PROJECT=mmi25
```

or by setting the UNIX environment variable **PROJECT** to **mmi25**, for example:

```
setenv PROJECT mmi25
```

and then simply invoking SUE by typing:

```
sue
```

Standard Cell Library Info For DPC

One of the things you need in order to run DPC is the size of the standard cells. This can now be automatically commputer from a LEF file. Simply point to the LEF for your cells in the **dpc.suerc** file.

If you don't want the LEF files accessible by DPC users, you can create a DPC cell size file from the LEF.

If you run the following command on your **LEF** file (in this case **mmi25.lef**), you'll get the information needed for DPC:

```
egrep 'MACRO|SIZE' mmi25.lef
```

Take this information, convert it to grids, and put it in the following format in the **mmi_local/sue/mmi25/library/dpc/mmi25.dpc** file.

```
MMI_AND2B   4.0
MMI_AND2C   5.0
MMI_AO21B   6.0
MMI_AO22B   7.0
```

The numbers in the **mmi25.dpc** file are in terms of grids and not microns. You need to know the grid size used in the **LEF** file. The values in the **mmi25.dpc** file are in terms of grids.

Look in the **dpc.suerc** file for the following lines which set up the grid for this library:

```
# grid size
set DPC(SCALE) 0.74
```

Pearl Interface

Edit the following lines in the **dpc.suerc** file to set up the *Pearl* timing analyzer.

```
# pearl timing setup
set DPC_TIMING(pearl,command) pearl
# set DPC_TIMING(pearl,command) "rsh mmics1 cd [pwd]\\; \
/users/tools/pearl/bin/pearl"
set DPC_TIMING(tech_file) $tech/library/pearl/pearl.tech
set DPC_TIMING(ctlf_file) $tech/library/pearl/$techtype.ctlf
set DPC_TIMING(filter) {-same_path -similar_path -node_suffix [#]}
```

The second **DPC_TIMING(pearl,command)** statement is how you would set up *Pearl* to run remotely.

One of the things required to run *Pearl* is a *technology file*. Below is an example of what needs to be in the technology file.

```
technology mmi25
power_node_name vdd 2.25
gnd_node_name gnd
logic_threshold  1.125
rise_time 20-80 200PS
SYNOPSYS_UNITS 1N 1P 1K
TEMPERATURE 105
PROCESS 1.0
LIBRARY_CORNER TYP
units time 4.0ps 1e-12
units capacitance 4.0fF 1e-15
end_technology
```

The **ctlf_file** is the *timing library file* required by *Pearl*. This file can be created from the Synopsys **.lib** file by first running **syn2tlf** and then compiling it with the **ctlf** compiler. If the customer is already running *Pearl*, they should already have the **ctlf** file generated.

In order to run **spice last critical path** with *Pearl*, there are a number of things which need to be set up. If you need to do this, contact MicroMagic, Inc. at support@micromagic.com.

PathMill Interface

The following lines in the **dpc.suerc** are edited to set up DPC to run *PathMill*. There are two modes in which you can run *PathMill*. The typical way to run *PathMill* is to run it down to the transistor level. This is how the *PathMill* interface is set up for DPC.

```
#set DPC_TIMING(simulator) "pathmill"
set DPC_TIMING(pathmill,command) pathmill
set DPC_TIMING(pathmill,tech_file) ""

# Points to spice netlists of leaf cells
set DPC_TIMING(pathmill,library) ". $tech/stdcell/spi"
# Header file
set DPC_TIMING(pathmill,spice_header) \
"$tech/stdcell/spi/include_params"

# For spice last critical path only
set DPC_TIMING(pathmill,spice_command) read_spice_deck
set DPC_TIMING(pathmill,spice_include) \
{{.lib '$tech/spice/mmi25.l' TT}}

# optional pathmill configuration commands
set DPC_TIMING(pathmill,cfg_options) \
{{pwl *} {xfer_all_pair} {log_on all} {search_structure mux}}
```

PathMill can directly read **BSIM3** models. The SPICE header file (**include_params** in this example) specifies where the SPICE models are located, as well as the temperature and voltage. *PathMill* also requires the SPICE netlist for each of the leaf cells. The **DPC_TIMING(pathmill,library)** variable points to the SPICE netlists. The **cfg_options** are some of the standard options used when running *PathMill*.

In order to run **spice last critical path** with the *PathMill* interface, add the **spice_include** variable, which is the same as the **.lib** statement, at the top of the **include_params** file. You also need to have the **mmi_local/sue/spice** script working, which should have been set up for the SUE installation.

PrimeTime Interface

To set up the *PrimeTime* interface, edit the following lines in the **dpc.suerc** file:

```
# if you want to use primetime, comment out next line
#set DPC_TIMING(simulator) "primetime"
set DPC_TIMING(primetime,command) pt_shell
set DPC_TIMING(db_file) $tech/library/synopsys/$techtype.db
set DPC_TIMING(db_time_units) 1ns(must correspond to your db file)
set DPC_TIMING(db_cap_units) 1pF(must correspond to your db file)
```

The **db_file** is the format of the library timing file used by *PrimeTime*. This file can be generated from a Synopsys **.lib** file. To do this, type the following at the UNIX prompt:

```
dc_shell
read_lib <tech>.lib; write_lib
```

You may already have a **.db** file for the library. The **cap** and **time** units are the units which are inside the **.db** file.

Index

Symbols

A

B

C

D

E

F

R

S

T

V

W