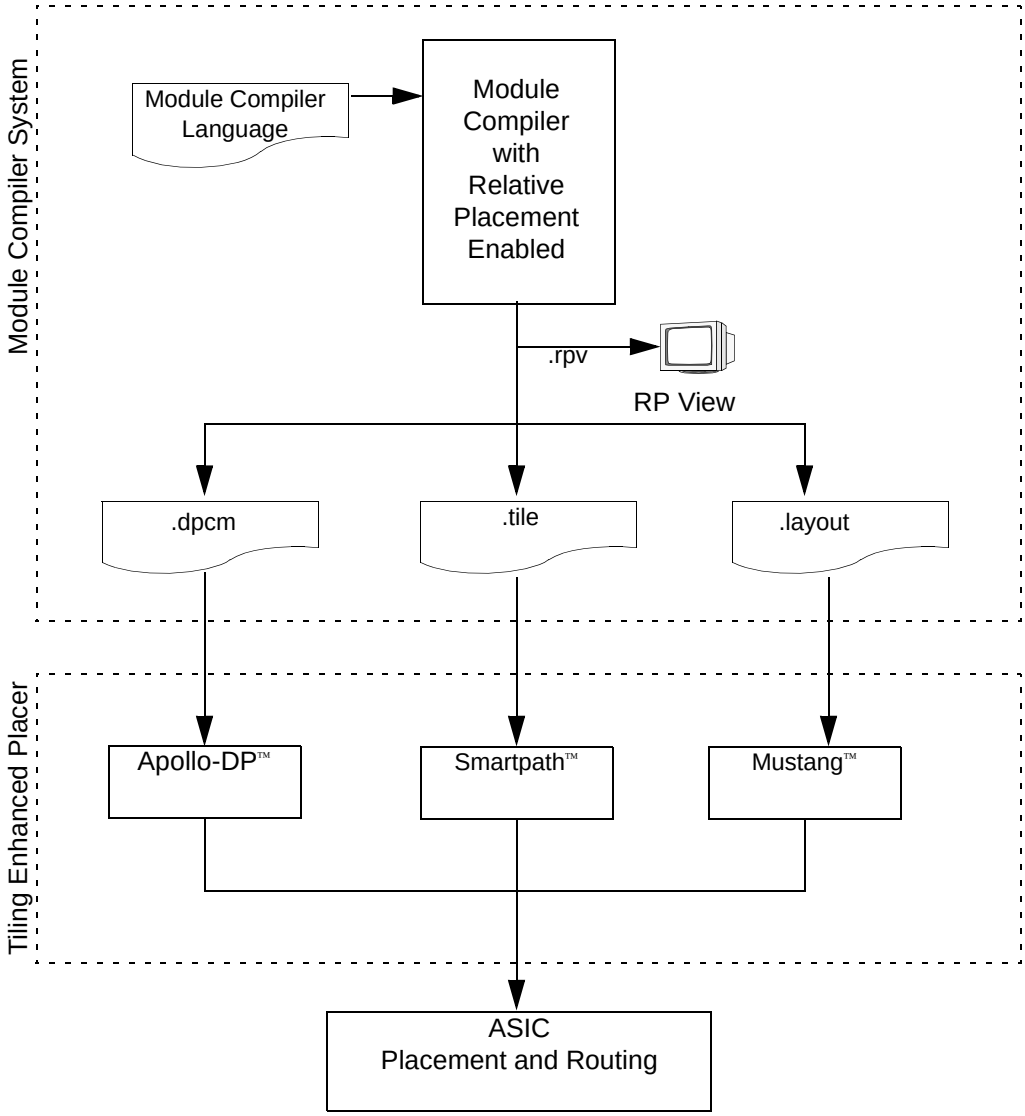
Module Compiler Relative Placement Flow

Figure 10-1 Module Compiler Relative Placement Flow



In Figure 10-1, Module Compiler RB Beta outputs two files:

- A relative placement information file (.layout, .dpcm, or .tile)
- A .rpv file for viewing in RP Viewer

Module Compiler Relative Placement Checklist

- Load 1999.05 Async., see Chapter 2 “Installation and Setup” of the *Module Compiler User Guide* for details.
- Read over this document and its appendix.
- Determine a directory to run Module Compiler RP in.
- Make sure the synthesis libraries are set for use with Module Compiler.
- Make sure that necessary mcenv variables are set before invoking Module Compiler Relative Placement, see “Enabling Module Compiler Relative Placement” on page 10-5.
- Enable Module Compiler RP (see next section).
- Select “RP Grouping” check box in GUI (see Figure 10-5).
- Select Optimization Controls in GUI (see “RP Optimization Control” on page 10-16).
  - Select “RP Optimizations” check box.
  - Determine multiplier RP.
  - Specify column removal rate.
  - Calculate logical height and enter value.
  - Determine I/O Locations (Left-to-Right recommended).
  - Specify level of column merging optimization effort.
- Specify Output Format (see Figure 10-21).
  - Specify output format (in addition to .layout).

- Specify output filename.

- View RP using Module Compiler View (see “Module Compiler Relative Placement Viewer” on page 10-32.

Enabling Module Compiler Relative Placement

- The Module Compiler Relative Placement application is built into the Module Compiler application. You do not need any additional software to run the relative placement capability. Relative placement is enabled by default and therefore does not need to be enabled. To disable the Module Compiler Relative Placement capability, you will need to modify the mc.env file.

- To disable the Module Compiler Relative Placement capability, you will need to modify the mc.env file.

  - Using a text editor such as vi or EMACS, add the following line to the beginning of the mc.env file:

    ```
    dp_physical -
    dp_layout_out -
    ```

  - Alternatively you can type at the UNIX prompt:

    ```
    % mcenv dp_physical -
    % mcenv dp_layout_out -
    ```

    These commands will enter the variables and their settings into the mc.env file. If mc.env does not exist, Module Compiler will created it for you.

You must set both variables to enable Module Compiler Relative Placement capability. Setting these variables will modify the Module compiler GUI. The new GUI features will be covered later in this document.

Starting Module Compiler Relative Placement

Once you have enabled Module Compiler Relative Placement capability by editing the mc.env file, you can run the application.by typing at the UNIX prompt:

```
% mc -tech <my_library>
```

This will bring up the Module Compiler GUI. Clicking on the “Optimization” menu will show a screen similar to that in Figure 10-2.

Figure 10-2 RP Layout Options Optimization Menu Selection

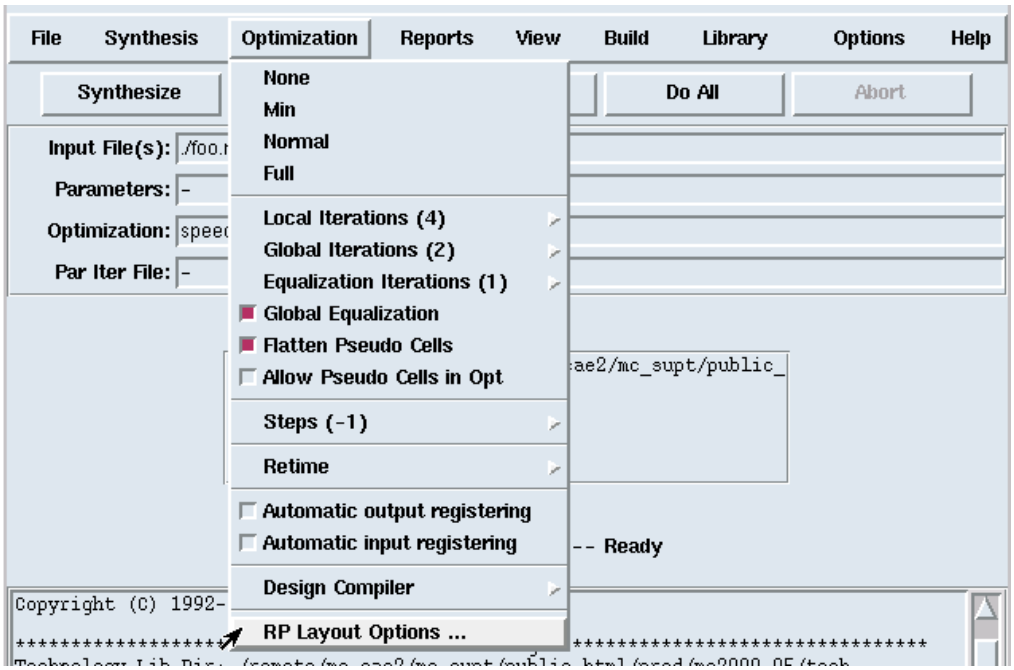


Figure 10-2 show that Module Compiler Relative Placement has added an additional submenu item “Physical Layout Options...” to the “Optimization” menu in the graphical user interface (GUI).

Relative Placement Overview

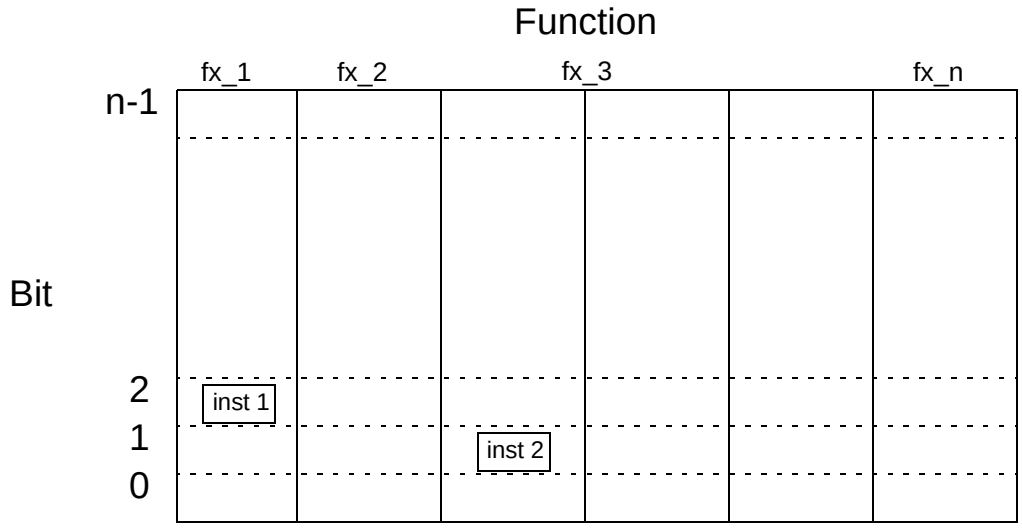
Relative placement is also called tiling or datapath structured placement. You use RP information to control the placement of instances in your design.

Module Compiler Relative Placement assigns an instance to a row and column position. Each row corresponds to a single bit of a bus, and each column corresponds to a function or an operation. The instances making up a function span multiple bits (rows) and are arranged vertically in a column. A function can be composed of one or more columns.

Module Compiler Relative Placement optimization provides column and row optimizations using a cost function to improve wire length and/or utilization.

Your layout system might require that the bit slices be vertical rather than horizontal, but this change does not impact the general operations discussed in this section.

Figure 10-3 Module Compiler Relative Placement Illustration



In Figure 10-3, instance 1 is located in the first column (fx_1) and 3rd row (bit 2). Instance 2 is located in the 3rd column (fx_3) and 2nd row (bit 1). Also columns 3 and 4 make up function 3 (fx_3).

Relative placement does not specify absolute (X,Y) locations for instances as would full ASIC place and route. The benefit of working with relative placement rather than absolute placement is efficiency and speed. By working at a higher level of abstraction, you can quickly explore different placement approaches.

In order to view the results in RP View, Module Compiler Relative Placement writes out a file (.rpv) and a viewer (MC View).

Module Compiler provides the RP information in a file that provides an efficient starting point for placement and routing. Module compiler supports several third party tile formats as well as providing its own format, .layout.

Module Compiler provides both physical grouping, gate-level tiling, and relative placement optimization:

- Grouping and gate-level tiling use directives in Module Compiler Language to specify relative placement (row and column positions) for instantiated gates.

  Gate-level tiling produces RP blocks using the `physicalfunction` directive. These blocks can placed in a physical group using the `group` directive. For more information see Figure 10-5.

- Module Compiler Relative Placement optimization provides column optimizations using a cost function to improve wire length and utilization. For more information see “Display Contrast Menu” on page 10-49.

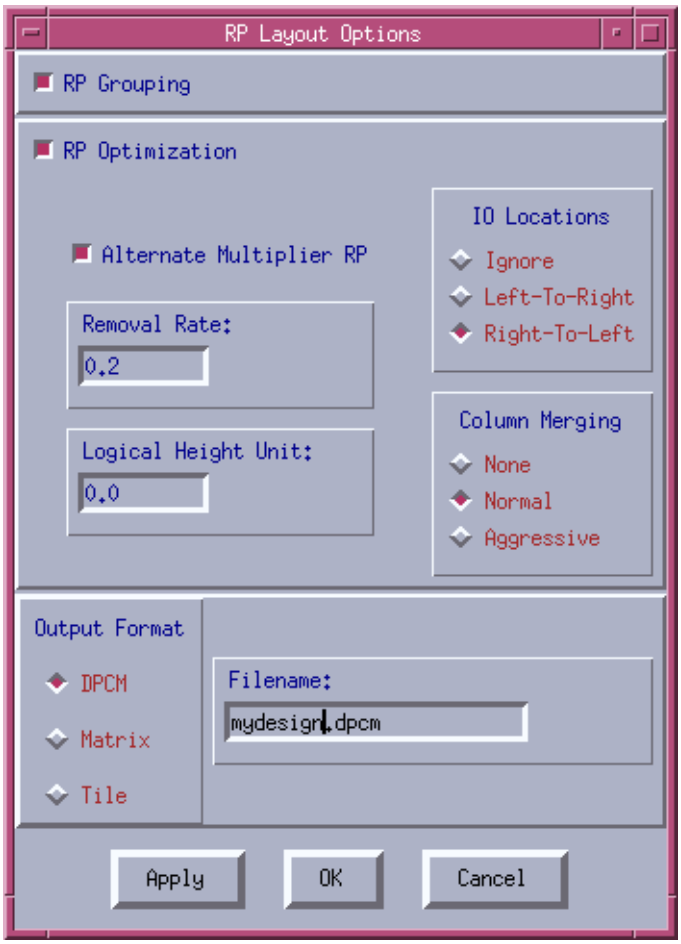
Module Compiler can automatically perform the following RP optimizations:

- Reorder columns
- Remove columns
- Merge columns
- Use alternative Wallace tree multiplier RP

You can choose the default optimization values, or control the RP optimization by setting your own values in the GUI.

With Module Compiler, the default for relative placement optimization is on. Module Compiler will perform relative placement optimizations. To disable RP optimization, turn RP optimization off in the “RP Layout Options...” submenu of the Module Compiler GUI. By disabling RP optimizations, Module Compiler will not perform RP optimizations or output a RP file.

Figure 10-4 RP Layout Options Window



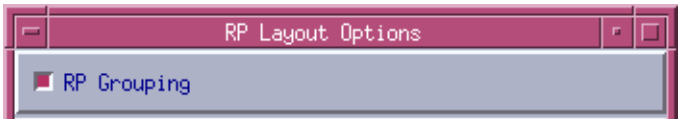
Use the “RP Layout Options” window to control the RP optimization. Clicking on the “RP Layout Options...” menu item in Figure 10-2 brings up the window in Figure 10-4. When you open this window for the first time, it displays the default settings.

This window has three buttons on the bottom: Apply, OK, and Cancel. When you click apply, Module Compiler enters the settings but keeps the window open. When you click OK, Module Compiler enters the

settings and quits the window. Clicking Cancel closes the window and keeps any previous settings unchanged. The rest of the controls on this window will be covered in detail.

Enabling Relative Placement Grouping

Figure 10-5 RP Grouping Control



Select this check box (Figure 10-5) to enable physical grouping in Module Compiler. Grouping is always supported in Module Compiler. But when you select this check box, Module Compiler will treat each logical group as a physical group.

If RP grouping is selected, and you do not specify a logical group, Module Compiler will create one physical group for the entire design and name the group “misc”.

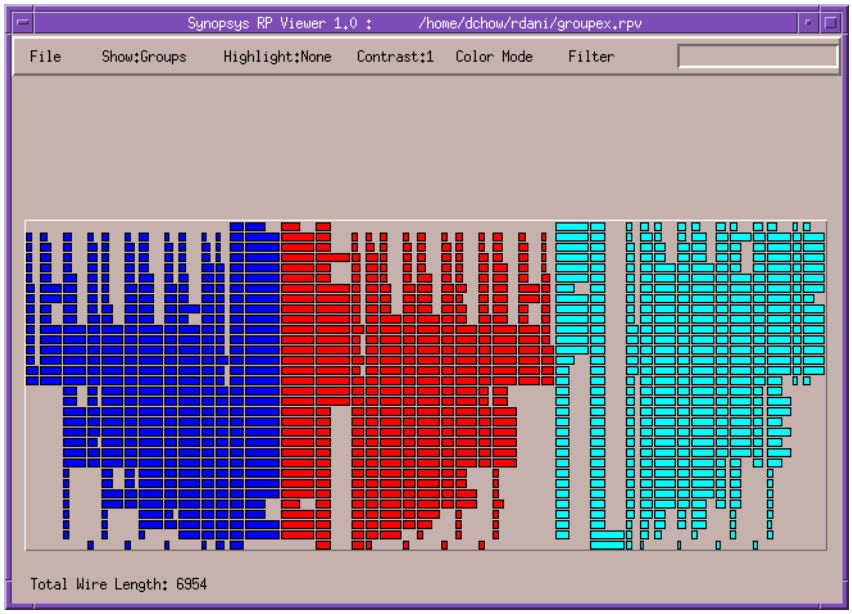
You use the Module Compiler Language `group` attribute to define a group and provide it with a name.

Example 10-1 Using the Group Attribute

```
module groupex (Z,A,B,C,n);
integer n=32;
input [n-1:0] A,B,C;
output [n:0] Z;
wire [n]tmp;
wire [n]tmp2;
tmp2=A+B;

directive(group="first");
tmp=C+B;

directive(group="second");
Z=tmp+tmp2;

endmodule
```

Example 10-1 gives an example of using `group` in Module Compiler Language. With RP Grouping enabled, Module Compiler creates the RP shown in Figure 10-6.

Figure 10-6 View of groupex Example RP



In Figure 10-6, you can see the creation of three groups (misc, first, and second) from the Module Compiler Language example (Example 10-1).

Gate-Level Tiling

Along with physical grouping, you can control gate-level tiling by specifying directives in the Module Compiler Language. Gate-level tiling allows you to create an RP block (physicalfunction) and specify cell row and column positions in that block.

This RP block can be associated with a physical group if you have enabled RP Grouping. A physical group can have one or more RP blocks created with the physicalfunction directive. A RP block can belong to one and only one group.

You can use gate-level tiling to create reusable RP blocks, such as an array multiplier, a ripple adder, and so on. The following discusses gate-level tiling directives and gives a Module Compiler Language example of gate-level tiling usage.

Gate-Level Tiling Directives

These are the directives in Module Compiler Language to support Gate Level Tiling:

- `physical`

  This directive can be set to “`on`” or “`off`”. If set to “`on`” then it will enable gate-level tiling

  ```
  directive(physical="on")
  ```

- `physicalfunction`

  This directive is used to name the bit-sliced RP block in Module Compiler Language.

  ```
  directive(physicalfunction="rippleadder")
  ```

- `row`

  This directive specifies the row position of a instance in its physical function. The default value for this directive is 0.

  ```
  directive(row=3)
  ```

- `col`

  This directive specifies the column position of a instance in its physical function. The default value for this directive is 0.

  ```
  directive(col=4)
  ```

Gate-Level Directive Example

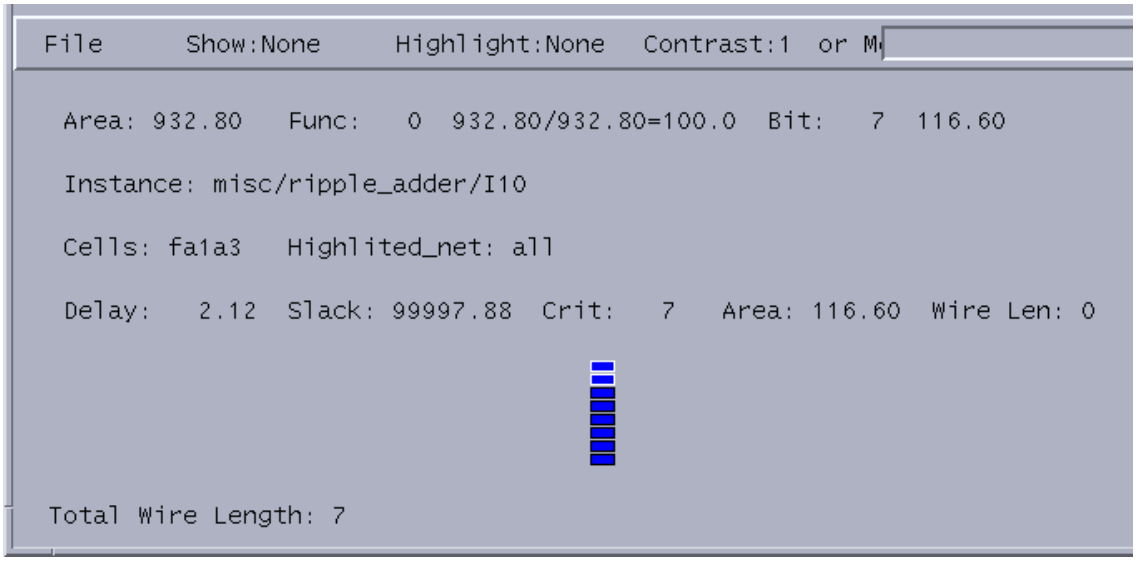
This example implements a ripple adder using the gate-level directives discussed above.

Example 10-2 Gate-Level Directive

```
module ripple(Z, CO, A, B, CI, n);
directive(physical="on");
directive(physicalfunction="ripple_adder");

integer n = 8;
input [1] CI;
input [n] A, B;
output [1] CO;
wire [1] repl(i,n) {CARRY_IN_{i}, }  CARRY_IN_{n};
wire [1] repl(i,n,",") {Z{i}};
output [n] Z=cat(repl(i,n,",") {Z{n-1-i}});

CARRY_IN_0 = CI;

repl(i, n) {
    directive(row={i},col=0);
    fa1a I10{i} (CARRY_IN_{i+1}, Z{i}, A[{i}], B[{i}],
CARRY_IN_{i});
}
CO = CARRY_IN_{n};

directive(physical="off");

endmodule
```

Figure 10-7 shows the result of the example above.

Figure 10-7 Result of Gate-Level Directive Example

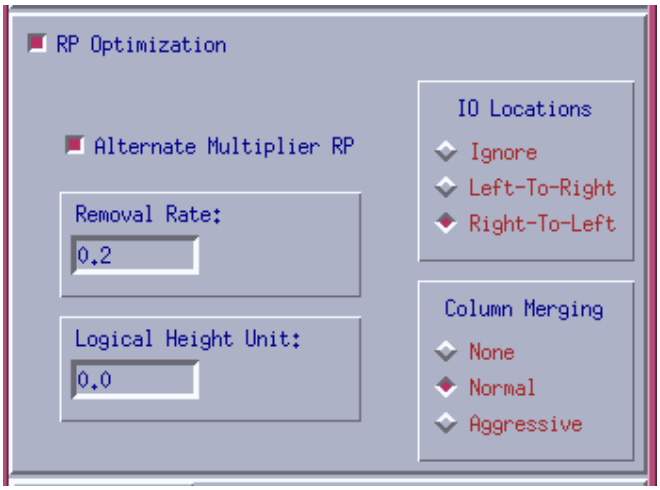


Viewing Example 10-2, in Module Compiler RP Viewer shows the creation of eight adders in the physical block named ripple_adder. RP View is covered in more detail in the section “Module Compiler Relative Placement Viewer” on page 10-32.

RP Optimization Control

The control section of the RP Layout Options Window is show below in Figure 10-8.

Figure 10-8 RP Optimization Control Section



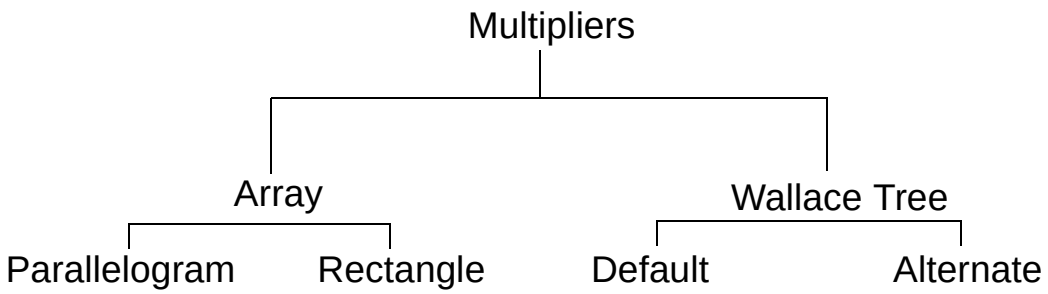
Select the PR Optimization check box to enable RP optimization in Module Compiler. This is the master switch to turn on all RP optimizations.

Alternate Multiplier RP Control

Select this check box to use an alternative relative placement multiplier.

In Module Compiler Relative Placement Beta, there are four Multipliers that you can choose from that are show in Figure 10-9.

Figure 10-9 RP Multiplier Choices

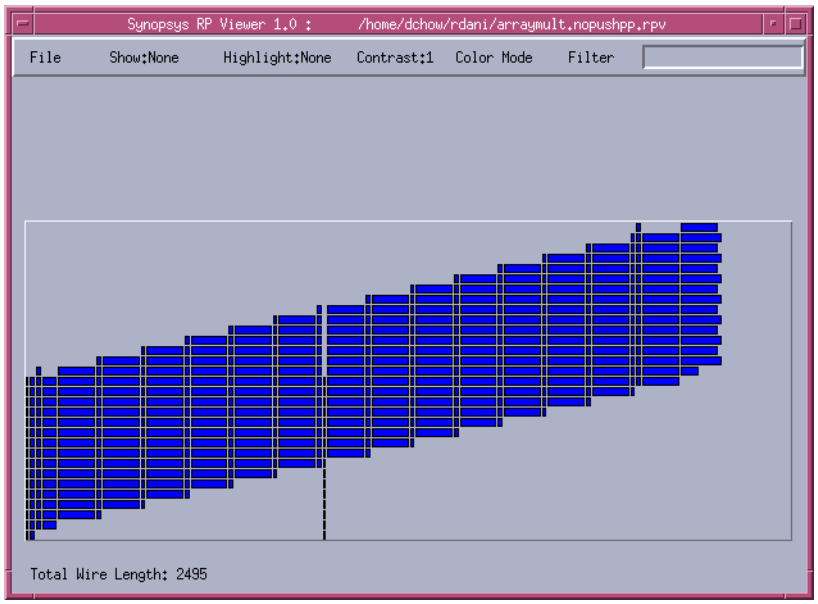


There are four multiplier choices for the Module Compiler Relative Placement Beta, which are shown above (Figure 10-9). The use of these multipliers depends your design goals. The array multiplier is serial and very regular, and can be easily placed in a bit-sliced fashion. It gives better utilization and wiring compared to a Wallace tree multiplier. The Wallace tree does not have the regularity of the array multiplier, but has better performance.

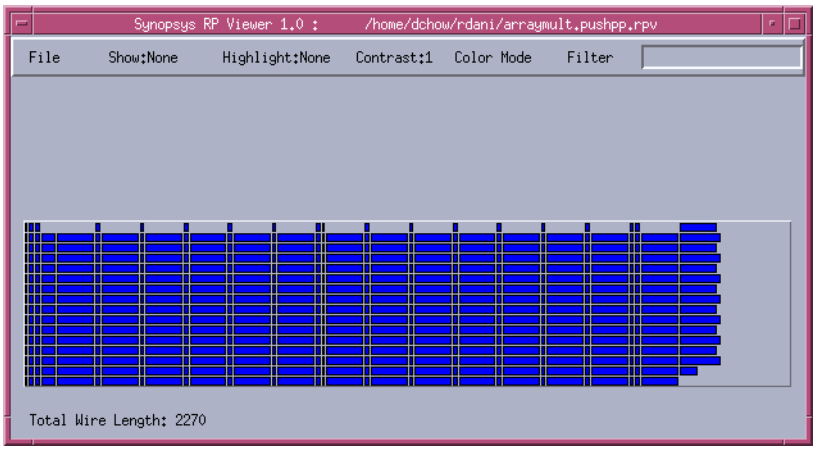
The two types of RP for an Array Multiplier are shown in Figure 10-10.

Figure 10-10 Array Multipliers

Parallelogram



Rectangle



Create an Array multipliers

These are the steps to create an array multiplier show in Figure 10-10.

Set maxtreedepth to 2 and multtype to nonbooth to create an array multiplier. This occurs even if you select the Alternative Multiplier check box. To create an array multiplier, see Example 10-3 on page 10-21 for details.

- Set the mcenv variable `rp_pushpp 1` to create a rectangular-shaped array multiplier RP.
- Set the mcenv variable `rp_pushpp 0` to create a parallelogram-shaped array multiplier RP. Zero is also the default value.

You can set rp_pushpp to 1 or Zero by typing one of the following commands at the UNIX prompt:

```
% mcenv rp_pushpp 1
```

or

```
% mcenv rp_pushpp 0
```

Alternatively you can use a text editor to add one of the following entries to mc.env:

```
rp_pushpp 1
```

or

```
rp_pushpp 0
```

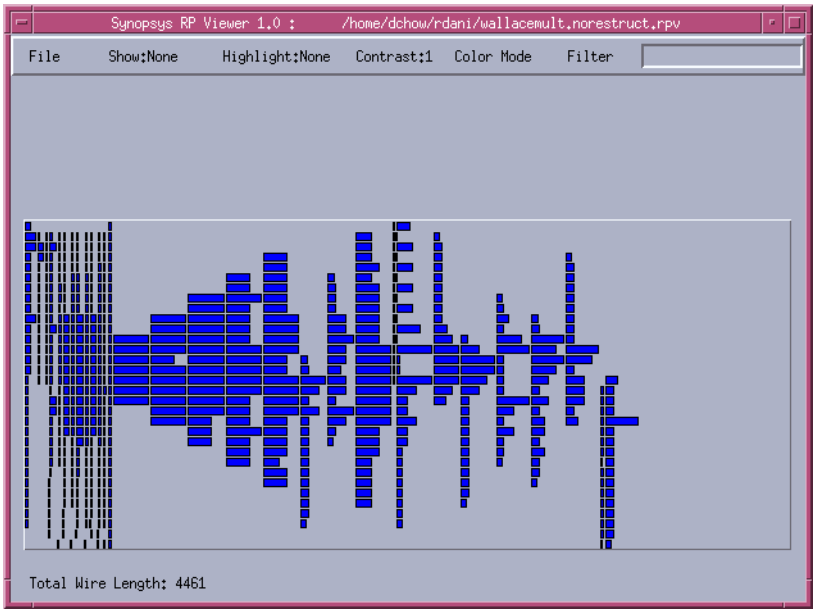
Example 10-3 Creating an Array Multiplier

```
//generate an array multiplier:

directive(maxtreedepth=2);
directive(multtype="nonbooth");

Z = A*B;

//A, B can be signed/unsigned numbers
```

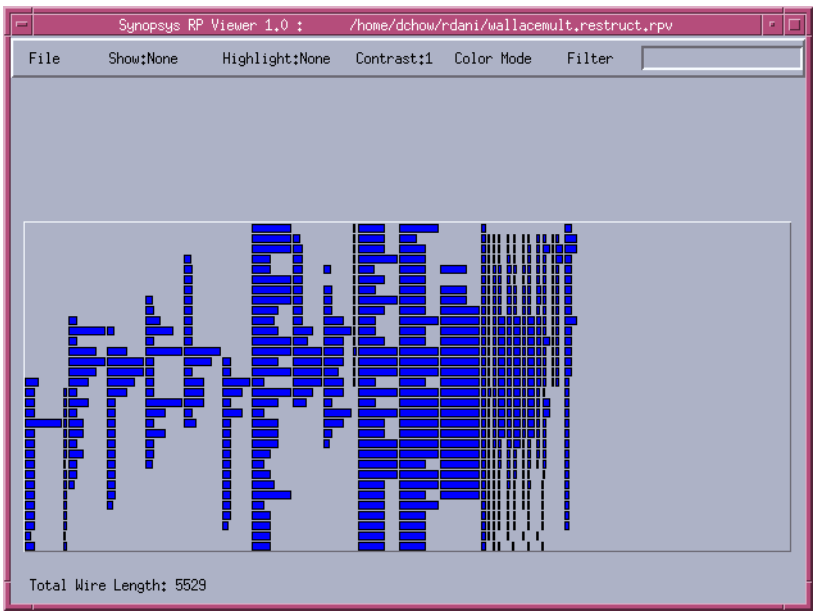
An example of the two types of RP for an Array Multiplier are shown in Figure 10-10. This is only an example for comparison purposes. Your result will depend on your design and optimization settings.

Figure 10-11 Wallace Tree Multipliers

Default



Alternate



Create a Wallace tree multiplier

You can specify an alternate relative placement for the Wallace tree multiplier. The default is off (unchecked) that gives the standard Module Compiler Wallace tree multiplier. Turn this feature on (checked) to use a alternate Wallace tree multiplier RP.
You can also control the Wallace tree multiplier by:

- Setting maxtreedepth equal to a high value such as 9999 to create a Wallace tree multiplier, which has lower regularity.
- Setting the value to between 2 and a high number (9999) to create an intermediate multiplier (between an array and a Wallace tree multiplier).

Specifying Signal IO Direction

Figure 10-12 IO Locations Control

This control in Figure 10-12, allows you to specify the signal IO locations. You can specify the signal direction or specify none (ignore).

Module Compiler Relative Placement optimization provides rudimentary support for I/O locations. There are three port locations: Ignore, Left-to-Right, and Right-to-Left.