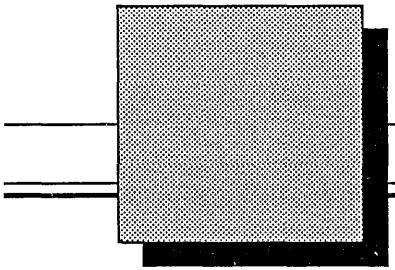


Library Development Guide

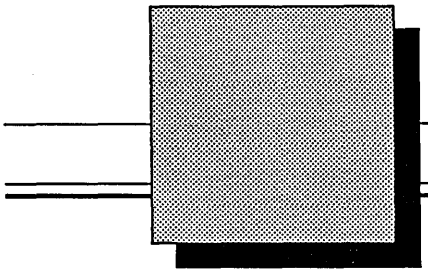
October 1, 1989

P/N: 900-00690 Rev. A

Valid Logic Systems, Incorporated
2820 Orchard Parkway
San Jose, California 95134
(408) 432-9400 Telex 371 9004
FAX (408) 432-9430

MANUAL REVISION HISTORY

Rev	Date	Software Release	Reason for Change
A	10–1–89	Library Release 9.0	Initial release.



Preface

The *Library Development Guide* describes creating and maintaining libraries. This guide is intended for use by the System Librarian and/or Library Developer. The Library Developer is responsible for maintaining and modifying all libraries and creating new libraries. Valid recommends that the Library Developer have the following qualifications:

- Knowledge of a system text editor
- Understanding of SCALD language concepts
- Operating knowledge of any Valid application necessary to create required component models and the ability to interpret the application results:

  GED
  ValidPACKAGER
  ValidCOMPILER
  ValidSIM
  ValidTIME
  Transcribe

Valid recommends that you give some consideration to library development decisions that are pertinent to your site requirements. Some of the issues you should resolve:

- Are you developing mil-spec or standard components?
- What body standards should you follow?
- What are the minimum test procedures for a completed component?
- What is the minimum size for a body or the text included with a body?
- Are you using ANSI or commercial components?
- What tools are you currently using? What tools will you be using in the future: Timing Verifier, Simulator? Should you save time by creating models now for tools you might use in the future?
- Will users be developing their own components or will the Librarian be the only developer? Will the Librarian test user-developed components?

The standards used in this manual are for commercial components. If you are building ANSI library components, you should follow the standards designed for ANSI components.

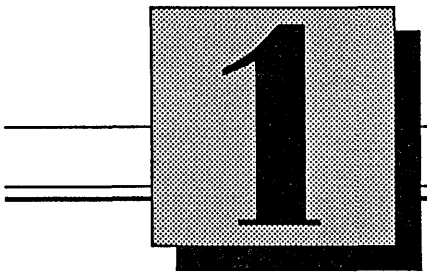
Several Valid manuals are helpful during the library development process. Valid recommends that the Library Developer have access to the following documentation:

- Library Development Guide
- Library Reference Manual
- ValidCOMPILER Reference Manual
- ValidPACKAGER Reference Manual
- ValidSIM Reference Manual
- ValidTIME Reference Manual
- SCALD Language Reference Manual
- Tutorial I: Logic Design
- Tutorial II: Using your Validation Designer

The *Library Development Manual* covers the following topics:

Section 1:	Library conventions and syntax issues; library organization; library maintenance.
Section 2:	Creating library components.
Section 3:	Creating the physical model.
Section 4:	Creating the simulation model.
Section 5:	Creating the timing model.
Section 6:	Creating support components.
Section 7:	Testing a new library.
Appendix A:	Text file method of creating a library drawing.
Appendix B:	Changes in the library drawing method.

Table of Contents



Library Fundamentals

This section discusses:

- Library development process
- Library conventions
- Signal name syntax
- The library directory
- The master library file
- Library components
- Component versions
- Operating system considerations
- Maintaining libraries on foreign hosts

The Library Development Process

There are several steps involved in creating a new library and new components. The section of this manual that contains further information on a procedure is noted beside each step.

1. Build a body drawing. *(Section 2)*
2. Create a physical model. *(Section 3)*
3. Create a simulation model (if required). *(Section 4)*
4. Create a timing model (if required). *(Section 5)*
5. Test the components. *(Section 7)*
6. Update the master library file. *(Section 1)*

Section 6 discusses creating support components.

When designing components, some decisions must be made about how to assign values that are not specified in the data sheets. The librarian must decide what values are to be used and then must maintain consistency for all components in the library. Such decisions should be documented in a file placed in the directory so that other users of the library can read them.

As a general rule, permissions on component models (body drawings, simulation models, and timing models) are set so that only the librarian or root has permission to change the models.

Library Conventions and Syntax

Library conventions and syntax affect how model descriptions are entered for all libraries. The Valid libraries conform to the following conventions and syntax.

Conventions

Conventions govern, to a large extent, the shape of bodies in the libraries and how signals are named. Conventions have far-reaching scope and affect all libraries and all designs made with those libraries. Global conventions must be decided on prior to purchase or installation. Many conventions are determined by corporate policy; others are determined by the user.

Signal Name Syntax

Signal names contain information about the signal condensed into a very short space. Library models use signal names to convey information and to correctly model parts. As a consequence, the designer must follow the same syntax used in the libraries.

Valid supports five different signal syntaxes. These are referred to as Library Formats 1 through 5. Only one syntax or library format can be used at a site.

Library Format 1 is the Valid standard library format. A signal name in Format 1 consists of five fields in the following order:

negation name subscript assertion general_properties

The five library formats vary in four respects:

- The order of the five fields
- The bit ordering convention (left–to–right or right–to–left)
- The characters used to indicate low and high assertion
- The character or characters used to indicate a bit subrange

For more information on signal syntax and a description of each Library Format, see the SCALD Language Reference Manual.

It is important to decide on a Library Format before on–site installation. The signal syntax in use at any given site is defined in the *config.dat* file. Table 1–1 gives the location of this file on the various platforms.

Table 1–1. Config.dat File Locations

Platform	File Location
DECstation	/usr/valid/lib/config.dat
PC AT	/u0/scald/config.dat
SCALDsystem	/u0/scald/config.dat
Sun	/usr/valid/lib/config.dat
VAX	SCALD$ROOT:[LANGUAGE]CONFIG.DAT

If the analysis programs are to run on a VAX mainframe, then an identical config.dat *file must reside on the host.*

Signal syntax is often set by predefined company standards. If your company has no standard, we suggest you use the Valid standard library format (Library Format 1). Examples in the documentation assume Library Format 1 unless specified otherwise.

Library Organization

A library consists of a main directory with subdirectories for each component in the library. All of the Valid-maintained libraries on your system have a similar structure. Any library you create will also have a similar structure.

The Library Directory

All of the Valid-supplied libraries are installed in a single library directory. Table 1-2 defines the location of this directory on each of the Valid-supported systems.

Table 1-2. Library Directory Locations

Host	Directory Location
DECstation	/usr/valid/lib
PC AT	/u0/lib
SCALDsystem	/u0/lib
Sun	/usr/valid/lib
VAX	SCALD$ROOT:[LIBRARIES]

Figure 1-1 shows the structure of a typical Valid-supplied library directory.

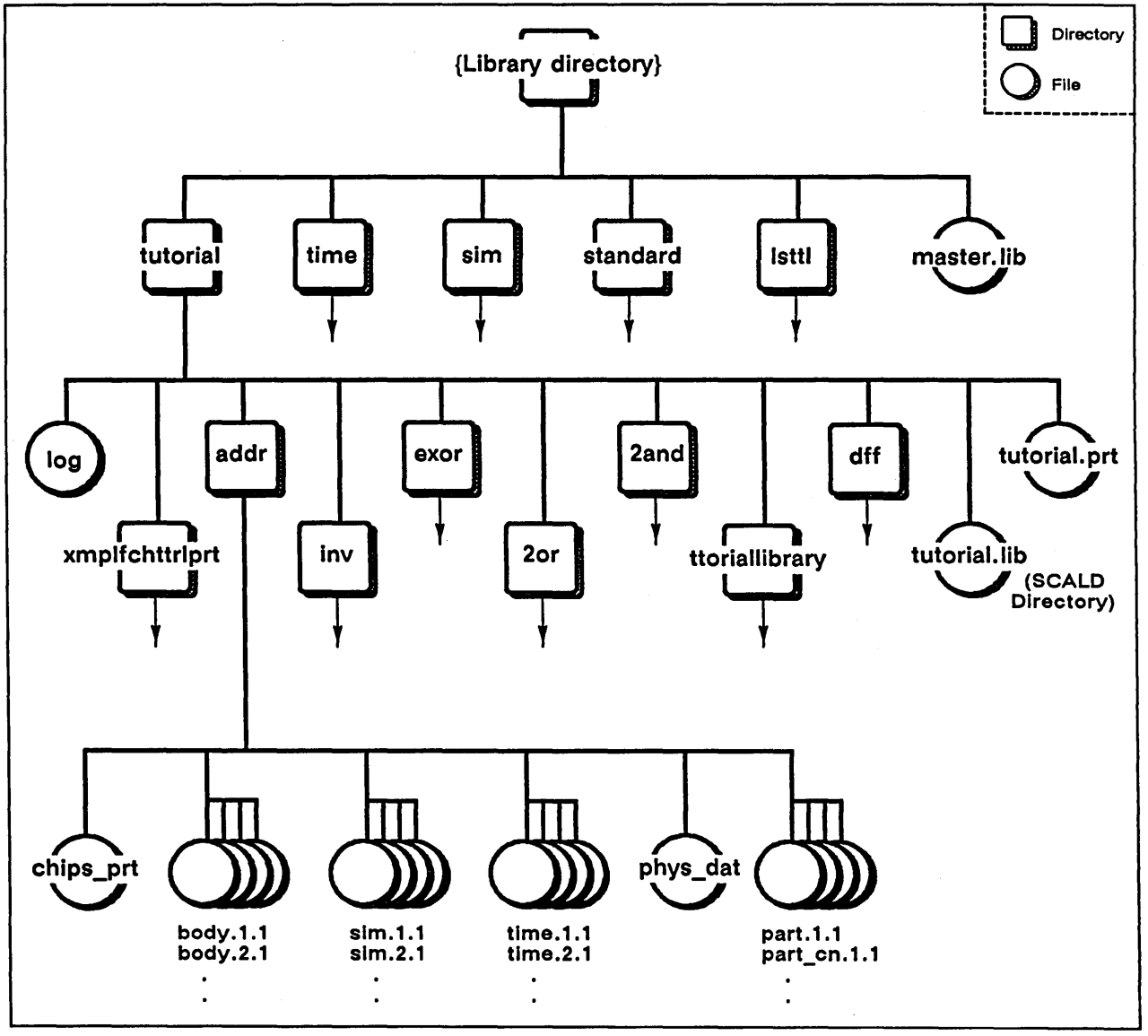
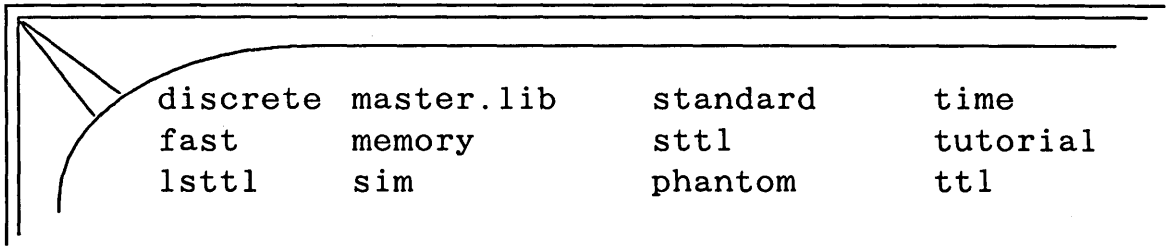


Figure 1–1. Contents of a Typical Library Directory

When you list the contents of the library directory, you see the name of each Valid library installed on your system and a few other files and directories. A sample listing of the contents of the library directory is shown in Figure 1–2.



```
discrete  master.lib    standard    time
fast      memory        sttl        tutorial
lsttl     sim           phantom     ttl
```

Figure 1–2. Contents of Sample Library Directory

Individual Libraries

Each individual library is stored in a directory bearing its name. For example, the tutorial library resides in the directory:

DECstation/Sun: /usr/valid/lib/tutorial

SCALDsystem: /u0/lib/tutorial

VAX: SCALD$ROOT:[LIBRARIES]TUTORIAL.DIR

In addition to the subdirectories containing the component drawings, a library directory may contain these additional directories and files:

- Two reference drawing directories. The drawing called "EXAMPLE OF EACH ..." includes an example of *every* version of every part in the library. It is primarily for documentation purposes. The drawing is also useful for testing the models for the library since, when used in

a compilation, it invokes all versions of all of the parts.

These drawing names are abbreviated within the operating system.

The other reference drawing includes the *first* version of every part in the library. This drawing usually has the same name as the library (for example, the "TUTORIAL LIBRARY" drawing). It can be used for library development.

- A SCALD directory. The extension *.lib* is reserved for library SCALD directories (for example, *lsttl.lib)*. This file contains the mapping between GED drawing names and their corresponding operating system directory names.
- A physical information file. The optional *.prt* file contains physical information about each part in the library. It may be used by the Packager and by the interface programs.
- A log file. Each library has a log file of all updates made to the library since its initial release. The log for Valid libraries is maintained by Valid personnel and is in reverse chronological order.

Figure 1–3 shows a listing of the contents of the tutorial library directory on the UNIX operating system.

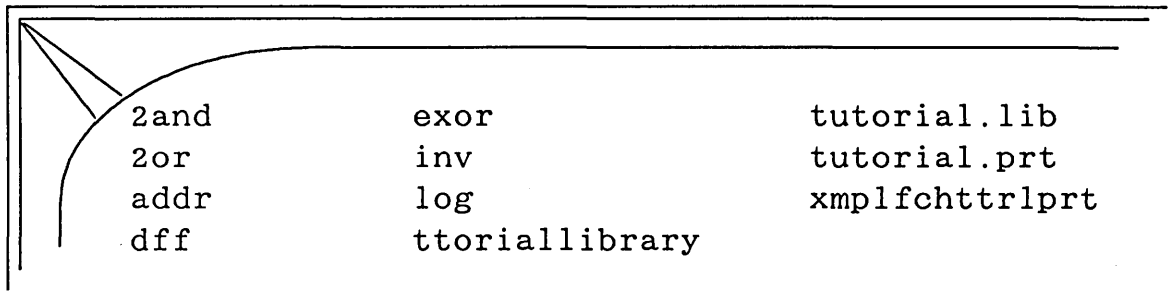


Figure 1–3. Contents of the Tutorial Library (UNIX)

The names of the drawing files are not exactly the same as the GED drawing names because of operating system limitations. The algorithm used to create file names acceptable to the operating system:

- Removes any spaces or special characters
- Shortens the drawing name to a maximum of 14 characters by selectively eliminating characters (beginning with vowels)
- Makes each name unique

The GED name–to–operating system name mapping is done automatically by the SCALD directory file.

The Master Library File

The master library file *(master.lib)* resides in the top library directory. It contains the name of each library on your system and the full path to that library. Adding a library entry to the *master.lib* file allows users to access that library.

The GED command **library** looks in the master library file for the full path to the specified library.

For instance, listing the LSTTL library in the *master.lib* file allows you to enter the command:

```
library lsttl
```

If there is no entry, you would have to enter the entire command line each time:

```
lib /usr/valid/lib/lsttl/lsttl.lib
```

A library can reside anywhere on the system as long as it is listed correctly in *master.lib*. When you have problems accessing a library, check the master library file and see if the library is listed.

It is not necessary to update the master.lib *file if you only change an existing library.*

The last thing you must do after you create and test a new library is to update the *master.lib* file. Use a text editor to access the file, and add an entry for the new library that lists the abbreviation for the library and the full path to the library.

Figure 1–4 shows a sample *master.lib* file on the Sun or DECstation system. The last entry adds the "userparts" library to the list of libraries that can be accessed on this system.

```
file_type = master_library;
'time' '/usr/valid/lib/time/time.lib';
'sim' '/usr/valid/lib/sim/sim.lib';
'standard' '/usr/valid/lib/standard/standard.lib';
'tutorial' '/usr/valid/lib/tutorial/tutorial.lib';
'lsttl' '/usr/valid/lib/lsttl/lsttl.lib';
'userparts' '/usr/valid/lib/userparts/userparts.lib';
end.
```

Figure 1–4. Sample Master.lib File

Library Components

A *library component* consists of a collection of drawings that together define the part. Within GED, each drawing has the same name and a different extension. The drawing names in GED are:

component.**BODY**

This drawing defines the shape, pins, and general properties of the library component; it is the symbolic representation of the part. When you add a library part to a drawing, the .BODY drawing appears on the screen. The .BODY drawing may represent an actual physical component, or it may represent a block of logic.

component.**PART**

The .PART drawing tells the Compiler that this component (body) is a low-level drawing that is added to logic drawings. This drawing contains a DRAWING body and its attached properties that define the physical component name for the Packager. When the Compiler compiles for "logic," it includes the global part information from the .PART drawing for each body on your drawing. This information is then passed on to the Packager.

component.**SIM**

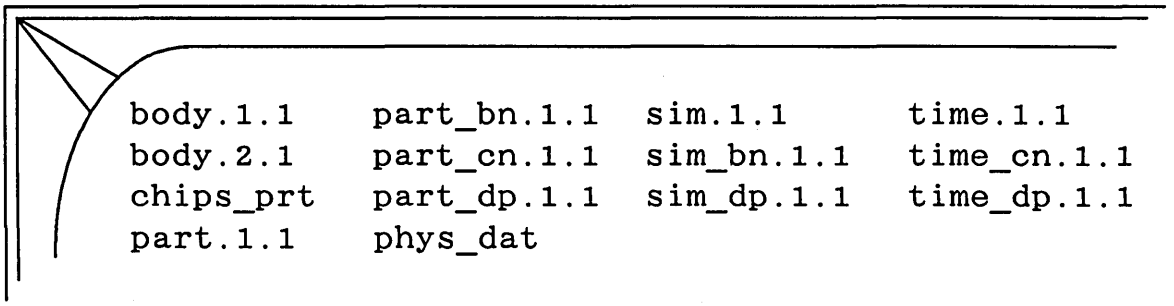
This drawing defines the simulation model for the library component. When the Compiler compiles for "sim," it includes the corresponding simulation model contained in the .SIM drawing for each body on your drawing. This information is passed to the Logic Simulator.

component.**TIME**

This drawing defines the timing model for the library part. When the Compiler compiles for "time," it includes the corresponding timing model contained in the .TIME drawing for each body on

your drawing. This information is passed to the Timing Verifier to check the timing behavior of the entire design.

When you list the contents of a component directory, the drawing files appear as shown in Figure 1–5.



```
body.1.1    part_bn.1.1  sim.1.1      time.1.1
body.2.1    part_cn.1.1  sim_bn.1.1   time_cn.1.1
chips_prt   part_dp.1.1  sim_dp.1.1   time_dp.1.1
part.1.1    phys_dat
```

Figure 1–5. Contents of Sample Component Directory

Additional Files

Besides the four component drawings, a library component directory contains these additional files:

- A *chips_prt* file. This file contains all the physical information for a component. The *chips_prt* file is used by the Compiler and by the **section** and **pinswap** commands in GED. (It may also be used by the Packager.) It allows you to preassign pin numbers to a component during schematic creation.

- A *phys_dat* file. This is an optional file used as input to a special program which can be used to create the *chips_prt* files. This file appears in some Valid libraries; these libraries may use a different procedure for modify-

ing physical information than do libraries without *phys_dat* files. See Appendix A for additional information on *phys_dat* files.

If disk space at your site is at a premium, the binary files can be deleted. GED can recreate any drawing from the ASCII version of the files.

- The binary files (*part_bn.1.1, sim_bn.1.1, time_bn.1.1*). These files contain the same information as the ASCII version of the drawings, stored in a slightly different format.

- The connectivity files (*part_cn.1.1, sim_cn.1.1, time_cn.1.1*). These files contain the logical net lists for the Packager, Simulator, and Timing Verifier.

- The dependency files (*part_dp.1.1, sim_dp.1.1, time_dp.1.1*). These files list each part used in the drawing and its library directory. This file is used by the GED **update** facility to ensure that the parts in the drawing are current.

There are some exceptions that require a logic drawing in a component directory.

Notice that there is no .LOGIC drawing in the component directory. Library components do not normally have logic drawings because components are the lowest-level drawings that are added to logic drawings. In flat designs, logic drawings are made up entirely of library components connected together. In hierarchical designs, logic drawings can be made up of library components and symbols representing other logic drawings connected together.

Component Versions

If a version is not specified, drawing version 1 is used by default.

The version number for a logic drawing should always be "1." The page number for a .BODY drawing should always be "1."

Many library parts have more than one body drawing to represent a part. Each of these body drawings is called a *version*. The *version number* is defined by the first digit following the drawing name extension (the third field in the name). The **version** command in GED selects which version of a part is used.

For example, there are two versions of the LS377 part in the LSTTL library: LS377.BODY.1.1 and LS377.BODY.2.1. Figure 1-6 shows the two versions of the LS377 component.

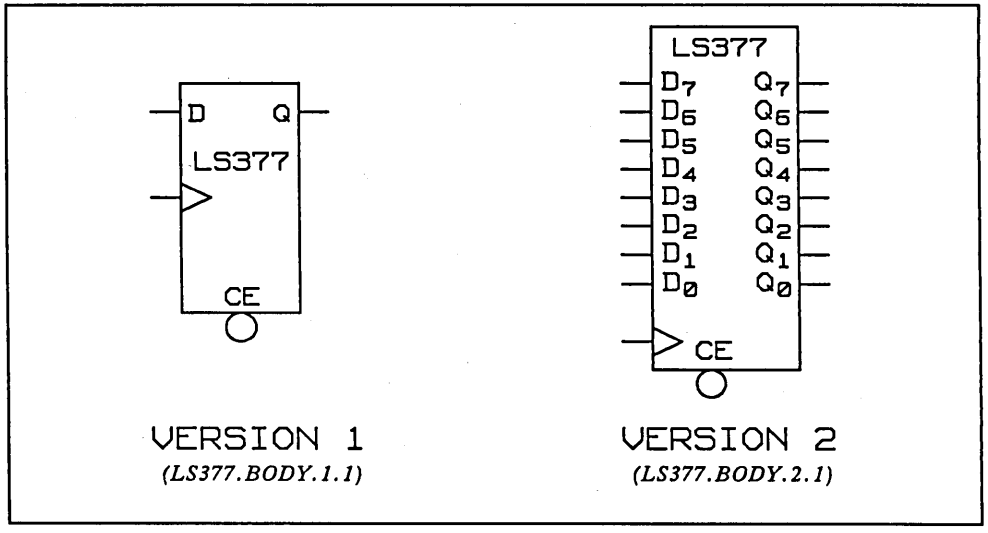


Figure 1-6. LS377 Versions 1 and 2

This is Valid's convention for defining body versions. You are not restricted to these conventions.

The Version 1 body drawing usually shows just one representative section of a package. The Version 2 body drawing typically shows all of the sections. (In the case of a simple gate, the second version usually shows the DeMorgan equivalent of the gate.)

The Version 1 Drawing

Because all sections of the LS377 are identical to each other, the Version 1 body can be used to represent:

- One section of a package
- Many sections of one or several LS377 packages

For more information on the SIZE property, see Section 2.

The Version 1 drawing of the LS377 is called a *sizeable* body. The drawing can be used to represent multiple sections by using vectored signal names and attaching the SIZE property to the drawing (after it has been added to a GED logic schematic).

The Version 2 Drawing

The Version 2 drawing of the LS377 more closely resembles the physical package of an LS377. The LS377 package contains eight identical sections, and the Version 2 drawing shows eight input pins and eight output pins. The Version 2 drawing is used for flat designs.

See the PIN_NUMBER *portion of Section 3 for more information on asymmetrical components.*

In most cases, the two body versions must have equivalent pin names. An exception to this rule occurs in parts with asymmetrical sections. In this case the versions of the part that represent the different sections must have no identical pin names, so that the different sections can be distinguished. Additionally, there must be a property attached to each section identifying the section. The Valid convention is to name this property "section" and to give the property a value that identifies the section number of the part to which the body corresponds.

Some simple logic gates have versions (the DeMorgan equivalents) that represent the two different

logical functions performed by the gate depending on the polarity of the input signal. An LS08, for example, performs an AND of high-asserted signals or an OR of low-asserted signals. The versions of the LS08 allow the designer to add either form of the gate to a drawing.

If a part has sections that are not interchangeable (such as the LS51), then there are additional versions that describe the additional sections. Figure 1-7 shows the different sections of the LS51 component.

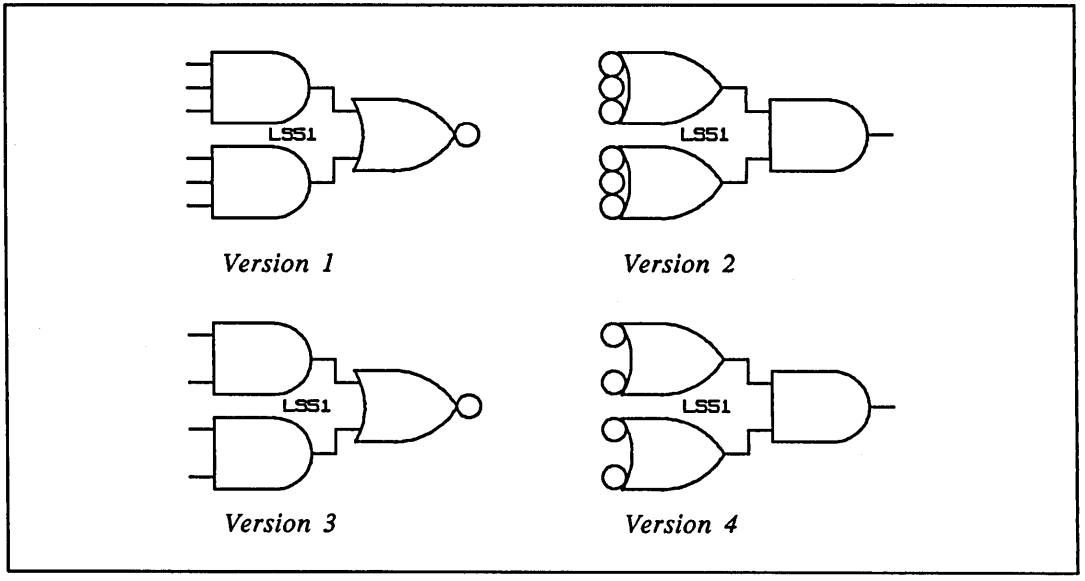


Figure 1-7. The LS51 Asymmetrical Component

Library Maintenance

The System Librarian must be very familiar with the SCALD system, the SCALD Language, and logic design. The librarian also needs to be reasonably conversant with the host's operating system and text editor. Maintaining libraries requires considerable caution since an error in a library affects many users and many designs (including completed designs).

Operating System Considerations

There are two important issues to consider for libraries: disk space and file protection.

Disk Space

Table 1–2 on page 1–5 lists the library directory for each system.

All libraries are stored in the library directory. There must be enough space in the directory for the libraries plus enough space left over for the users. For Valid-supplied libraries, the library space requirements are indicated in the individual library descriptions.

To determine the amount of space required for a user-created library:

1. Change directories to the library in question.

2. Determine how much directory space is used:

   UNIX/ULTRIX:
   ```
   du -s
   ```
   VMS:
   ```
   DIRECTORY/SIZE/GRAND_TOTAL [...]
   ```

3. Determine the amount of free space on the disk:

   UNIX/ULTRIX:
   ```
   df
   ```
   VMS:
   ```
   SHOW DEVICE DUA0
   ```

The amount of disk space required varies depending on the size and complexity of a user's designs.

The disk usage commands show the number of free blocks on */u0* or */usr* and *DUA0*, respectively. Leave enough disk space for users to work with after the libraries are installed. If installing a given library results in fewer than 1000 free blocks, you should either remove some files from */u0*, */usr*, or *DUA0*, not install the library, or acquire more disk space.

Protection

The libraries and their directories should be write-protected for everyone except the librarian. The files in the library should be owned by lib or system. Check library ownership by typing:

UNIX/ULTRIX:
```
ls -l
```

VMS:
```
DIRECTORY/OWNER/PROTECTION
```

If there are any files not owned by lib or system, you can fix this by logging on as root or system, changing your directory to the library in question, and typing:

UNIX/ULTRIX:
```
find . -exec chown lib  {} \;
```

VMS:
```
SET DIRECTORY/OWNER=SYSTEM
```

In VMS, respond to the "Directory" prompt with:

```
[...]
```

The *ls -l* and *DIRECTORY/OWNER/PROTECTION* commands also show the permissions of all of the subdirectories and files in the library directory.

With UNIX/ULTRIX systems, the correct permissions should be:

Files: **-rw-r--r--**

Subdirectories: **drwxr-xr-x**

This protection allows the user (which should be lib) to read and write the files, but allows everyone else only to read the files (UNIX/ULTRIX directories must have execute permission set in order to look inside the directory). If either the "group" or "other" write permissions or both are set (for example, -rw-rw-r-- or -rw-r--rw-), write permission should be removed by logging in as lib (since "lib" owns all the files), changing your directory to the library in question, and typing:

```
find .  -exec chmod go-w {} \\^;
```

With VMS workstations, the correct permissions should be:

Files: **RWED,RWED,R,R**

Subdirectories: **RWED,RWED,RE,RE**

With these protections, only the system and owner can read and write files; everyone else only can read the files. If either the "group" or "world" write or delete permissions are set (for example, RWED,RWED,RWE,RW), change the permission by logging in as system and typing:

```
SET PROTECTION=(G:R,W:G) *.DIR/LOG
```

Maintaining Libraries on a Foreign Host

If you work on a foreign host such as a VAX mainframe, keep a copy of the libraries on this machine. Otherwise each user must keep a private version of the libraries, which wastes disk space. These libraries should be kept in a read-only directory.

See the filecopy *utility in the* System Utilities Reference Manual *for details.*

To copy libraries to or from the foreign host, use the *filecopy* utility. You should run *filecopy* without a *transfer.log* file when copying libraries to ensure that all libraries are on the host. The transfer takes several minutes.

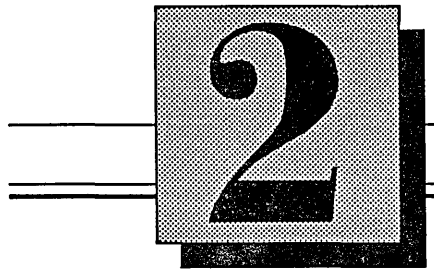
On the VAX mainframe, each library is stored in a directory under *[SCALD.LIBRARIES]*. Before you transfer the libraries, log in to the VAX and create a directory for each library you want to transfer.

For example, for a library named *biblio*, create the directory *[SCALD.LIBRARIES.BIBLIO]* on the VAX. The *filecopy.cmd* file would include the following directives:

```
report_files on;
copy_file 'biblio.prt';
directory 'biblio.lib';
host_kind VMS;
host_destination '/dev/vms/scald/libraries/biblio';
end.
```

Figure 1-8. Sample Filecopy.cmd File

Once the command file is set up, run the filecopy utility to copy all necessary information to the VAX.



Component Creation

This section discusses:

- Creating a library
- Creating a new component drawing
- Creating the body shape
- Drawing pins
- Adding pin names
- Annotating pins
- Attaching properties to the body
- Building other body versions
- Borrowing library parts

Basic Procedure (Checklist)

Creating a body involves the following steps:

1. Create a new library.
2. Create the new component drawing.
3. Create the body shape.
4. Add pins to the body.
5. Attach pin names to the pins.
6. Annotate the pins (using the GED **note** command).
7. Attach properties to the body.
8. Build other body versions.

Each of these steps is detailed in this section.

Throughout these procedures, there are creation standards included for your information. These are the standards used to create all of the Valid libraries; they are the suggested standards for creating your own libraries.

Creating A Library

The first step in developing a new library is to create a directory for the new library where you can store any parts you create. Since any change made to a Valid library is overwritten when that library is updated, you should also create a library of your own if you must modify an existing library part. Keeping new parts in a special library means your Valid libraries can be updated without losing data.

To create a new library on a UNIX or ULTRIX system, log in as user lib. By default the librarian's working directory is */u0/lib* on the SCALDsystem and PC AT or */usr/valid/lib* on the DECstation and Sun workstation.

To prevent access to new components until they are ready for distribution, create a permanent test directory under the *lib* directory. Use this directory to and test all new components. Do not enter the test directory name in the master library file. When the components are complete, copy the component directory tree, place the component in the correct library, and enter the new library name in the master library file.

To create new libraries on a VMS system, log in under your own user name. A library can be created under any user account and later moved to the *SCALD$ROOT*:[*LIBRARIES*] directory (although the file permissions and ownership will have to be changed to user lib).

Throughout this manual, a UNIX library named *newparts* (which is located under user lib) is used as a sample library.

Follow these steps to create the *newparts* library:

1. Log in as user `lib`.

2. From the */usr/valid/lib* directory, create the new library subdirectory:

   ```
   mkdir newparts
   ```

3. Copy the default GED files (**.cmd* and *startup.ged)* from a user directory into the new directory.

4. Move to the new directory:

   ```
   cd newparts
   ```

5. Edit the *startup.ged* file:

   ```
   vi startup.ged
   ```

6. Change the "use" line to read:

   ```
   use newparts.wrk
   ```

7. Delete the "masterlibrary" line.

8 Add a library command line for each existing library you might need to create the new library:

```
library lsttl
library ttl
```

9 Save the *startup.ged* file and exit the text editor.

10 Use the text editor to change the name of your SCALD directory in the *.cmd* files to the new directory name:

```
directory 'newparts.wrk';
```

Creating a New Component Drawing

Because the TTL 293 binary counter component is not included in the Valid library release, it is used as an example throughout this manual to demonstrate the process of adding a new component to your library.

To create a new component drawing, you must:

- Edit the .BODY drawing
- Move the body name away from the origin

Editing the .BODY Drawing

To create a new component called *293*, use GED to edit a new drawing called *293.body.2.1*. Make sure you add the *.body.2.1* extension; if you do not, the default drawing type is *.logic* and the default version and page number reference is *.1.1*.

Version 2 of a body is the "flat" version; that is, it has all pins explicitly marked. Version 1 of a body is *vectored,* with multiple-bit input and output pins. By building Version 2 first, you can more easily see the relationship between the component logic and the body representation.

When you edit *293.body.2.1*, the screen shows a body grid setting, the body name, and a small X (the body origin) in the center of the screen.

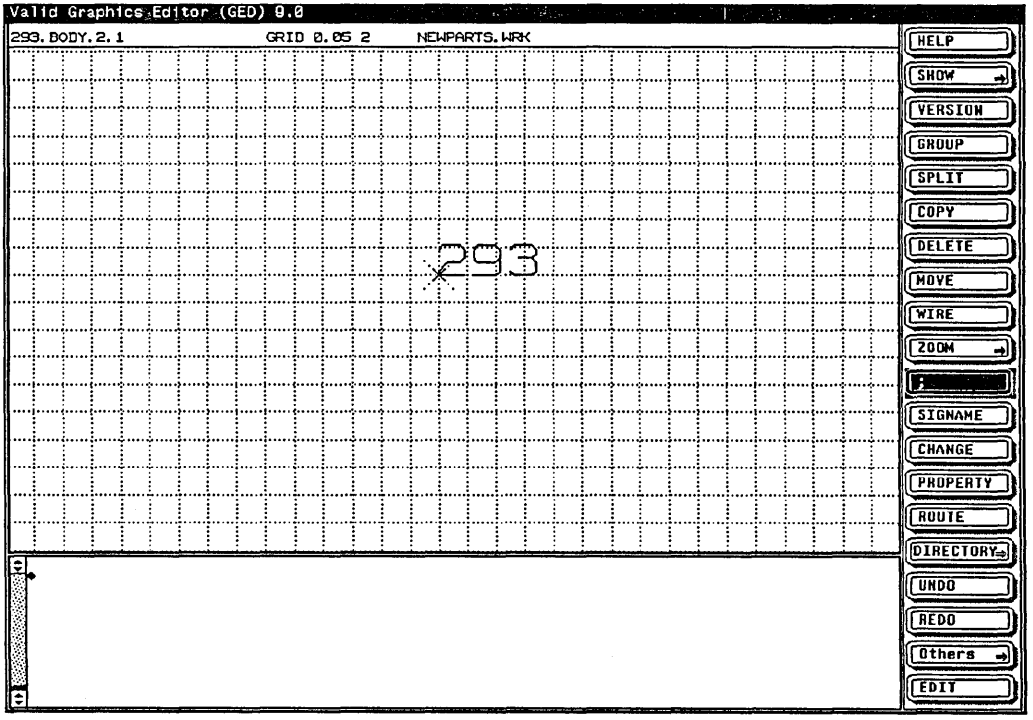


Figure 2–1. Editing The TTL Body

Grid Standards

- When you edit a .BODY drawing, the grid is set automatically to 0.05 2. This means there is one–tenth of an inch (0.1 inch) between displayed grid lines. Always use this grid setting when creating the body shape.
- Notes and connections to slanted lines (such as occur on select inputs of multiplexers, for example) can be placed on a grid of 0.01 10 if necessary. Do not use any other grid settings when creating bodies.

Moving the Body Name

The body name, 293, is a note that you can use to label the component body. Be sure to include the component name on the body. The **split** command lets you move the body name note away from the body origin (the small X that appears at 0,0).

To separate the body name and the body origin:

1. Select the **split** command from the GED menu.
2. Use the right mouse button to select the object to move.

   Do not move the body origin; if you do, the editor produces an error message when the body is written and moves the origin back to the center. If your first button click makes the origin move, just click again to select the body name.
3. Move the note to the top of the screen and place it down.

✓ The origin body is used to specify the origin of the body. All body properties are attached to the origin body, and the body should be symmetrical about the origin (the origin should be at the center of the body).

Shortening a Body Name

If you change the size of the note, it should still conforms to the suggested Valid standards.

In some cases, you may have a body name that does not easily "fit" into the shape of the body. If you wish to shorten the body name on a drawing, you can use the GED command **change** to shorten the body name note or the **display** command to change the size of the body name note to fit within the body shape.

Creating the Body Shape

Using GED, you can make a body any size and shape you want: round, oval, trapezoidal, rectangular. For consistency, the body shape should match the body function. Follow the component representation in the appropriate data book or use another Valid library component to model the shape of the body. In order for a body to integrate well with existing libraries, it should be approximately the same size as other library parts. When bodies are made a standard size and labeled in a consistent manner, schematics are easier to draw and maintain.

Note that the TTL 293 is not centered horizontally around the origin but that the body falls directly on the major grid intervals.

To create the shape of the TTL 293 body, use the **wire** command to draw a rectangle 10 squares long by 5 squares wide. Center the rectangle around the body origin as much as possible while still keeping the body shap on-grid. The basic shape of the TTL 293 component is shown in Figure 2–2.

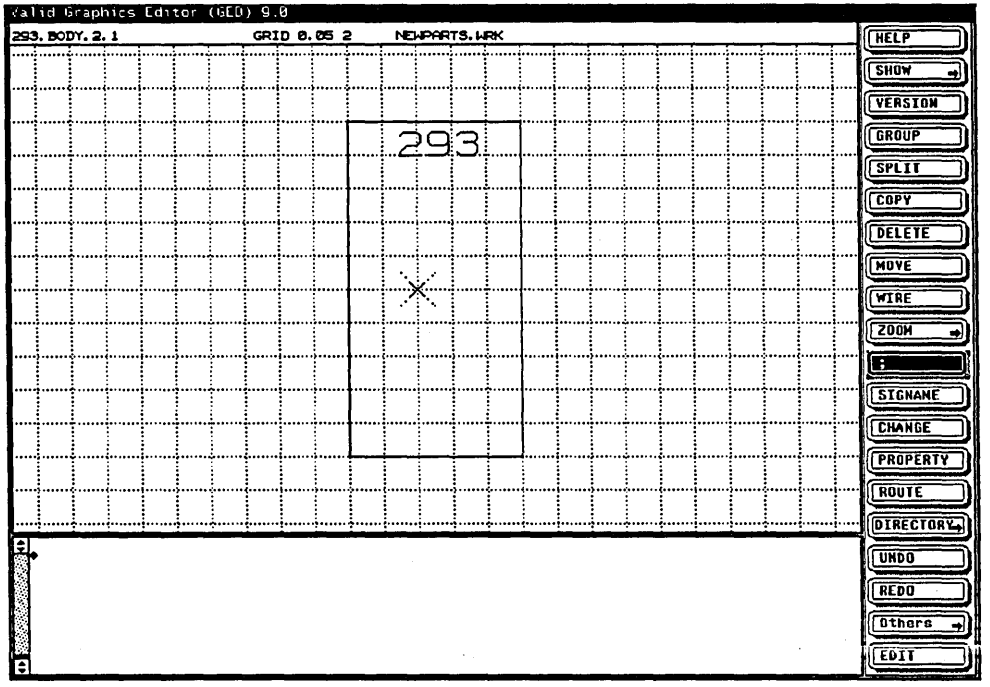


Figure 2–2. The TTL 293 Body Shape

Shape Standards

- The shape of the body should reflect (wherever possible) the function of the body.
- The size and shape of the body depend on the number of input and output pins required on the body drawing.
- The body should be symmetrical about the body origin.
- Bodies should be made as small as possible but not crowded. Follow the sizes of existing bodies in other libraries. Make flip–flops 0.4 by 0.8 inches. Make gates 0.3 by 0.6 inches.

Drawing Pins

The following pin conventions are used in all Valid-supplied libraries:

- High-asserted pins are shown with a wire stub.
- Low-asserted pins are shown with a circle.

Follow these steps to add pins to the TTL 293 body:

The circle has a 0.1-inch diameter.

1 Use the **circle** command to add two low-asserted clock pins to the left side of the body. Place the first point of the circle halfway between the first grid interval and the body edge. Place the second point of the circle on the body edge.

You can also use the **copy** *command to create multiple wire stubs.*

2 Use the **wire** command to add four high-asserted pins to the right side of the body. Start each wire at the body edge and extend it one grid interval out from the body.

3 Use the **arc** and **wire** commands to draw an AND gate at the bottom of the body. Place the first arc point one grid interval below the body and halfway between the second and third vertical grid intersections. Place the second arc point along the same horizontal grid interval and halfway between the third and fourth grid intersections. Place the third grid point at the bottom edge of the body halfway between the first and second grid intersections. Use the **wire** command to complete the gate and to add wire stubs to the bottom of the gate.

Dots can be open or filled. Use **set dots_open** *or* **set dots_filled** *to change the default.*

The pass-through clock pins are along the right side of the body. The pass-through clear pins are across the top of the body

4 Use the **dot** command to add a dot at the end of each circle and wire stub. This dot tells GED that this is a pin and that wires can be attached to it and connections made. The dot does not show up on the body when you add the component to a logic drawing in GED.

Remember to add the connection dots to the pass-through clock and clear pins.

5 Use the **wire** command and diagonal wires to add the clock wedges to the body.

6 Write the drawing to save it.

The TTL 293 body now looks like the one in Figure 2-3.

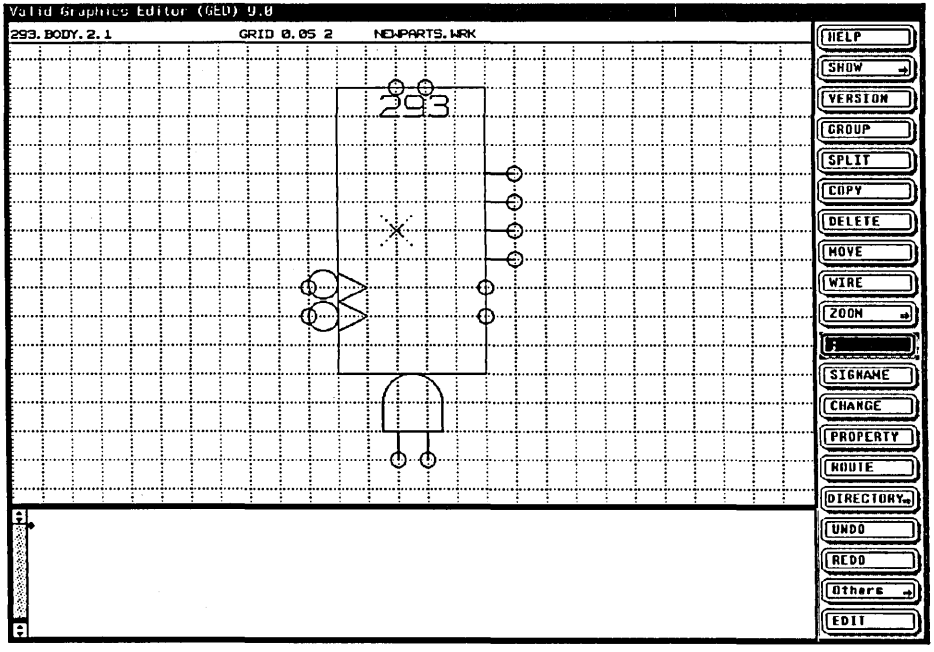
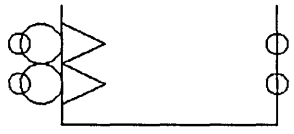


Figure 2-3. The TTL 293 Body and Pins

Pin Standards

- Place input pins on the left and output pins on the right. Place enable and select pins on the bottom. Place bidirectional pins on either side, usually on the right.
- Connect all pins to the body with either a 0.1-inch stub (made with a wire), or a bubble (0.1-inch circle), or, if the pin is bubbleable, with both a pin and a bubble (see page 2-15).
- Mark edge-triggered clock pins with a clock wedge 0.1-inch wide and 0.1-inch long. Use the white (center) button to draw the diagonal lines.
- Add pass-through pins wherever possible, especially on clocks, enable, and select lines. Do not add stubs or bubbles to pass-through pins.

Pass-through Pins



Pass-through clock pins

Pass-through (or feed-through or bus-through) pins are special "shortcut" pins placed on a body to make it easier to wire a group of bodies together. Many of the components in the Valid libraries are provided with pass-through pins on their clock signals. The pass-through pin is exactly opposite the clock signal. It lets you easily wire the clock signals of several library parts together.

Pass-through pins always appear on the .BODY drawing of the library part, although they are invisible when you add the body to a GED drawing.

For example, if you want to wire the clock signals of two LS374 components together, the pass-through pins allow you to wire in a direct line between com-

ponents. Without the pass-through pins, the wire would have to jog around the first component to connect to the second component. Figure 2-4 shows two LS374 components wired using the pass-through clock pins and two LS374 components wired without using the pass-through pins.

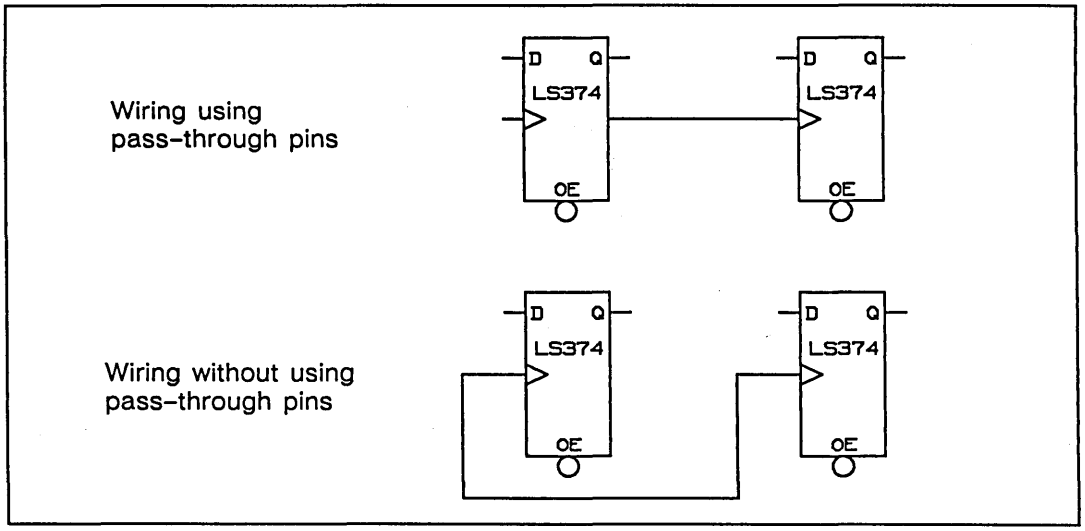


Figure 2-4. Using Pass-through Pins

Pass-through Standards

- Pass-through pins are always exactly opposite the visible pin to which they are logically connected.
- The pin names of the visible and the invisible pin must be the same. Identical pin names allow GED to interpret a wire connected to the pass-through pin as also being connected to the clock input pin.

- A pass-through pin never has a bubble, even if the pin it is associated with has a bubble.

  GED knows that pins of the same name are the same pin, and if one of them is bubbled, the other must be. This guarantees that correct bubble checking is performed even for pass-through connections.

- When defining a pass-through pin, make sure that the pin it is associated with is obvious. In Valid libraries, this is done strictly by its position on the body, not by the use of a note.

Bubbled Pins

The GED **bubble** command replaces the wire stub of a pin on a body with a bubble that represents a low-asserted signal. You must decide which pins can be bubbled.

A bubbleable pin

Since a circle indicates a low-asserted pin and a stub indicates a high-asserted pin, a bubbleable pin has both a stub and a circle. The length of the stub is the diameter of the circle. The connection point (marked by a dot) is at the far end of the stub, where the stub and circle meet.

To use bubbled pins in GED, you must attach the BUBBLED property to the body. For more information on the BUBBLED property, see page 2-29.

Adding Pin Names

You can use the text justification commands (SET LEFT, SET RIGHT, SET CENTER) to align the pin numbers.

The following conventions are used for pin names of parts in the Valid-supplied libraries:

- All pin names are based on the names in the relevant data books.
- On parts having multiple sections, the pin names carry a numerical suffix to distinguish among the sections (the bit subscript).
- Low-asserted pin names end with an asterisk character (*).

Use the **signame** command to add pin names to the TTL 293 drawing. Add the names as shown in Figure 2-5.

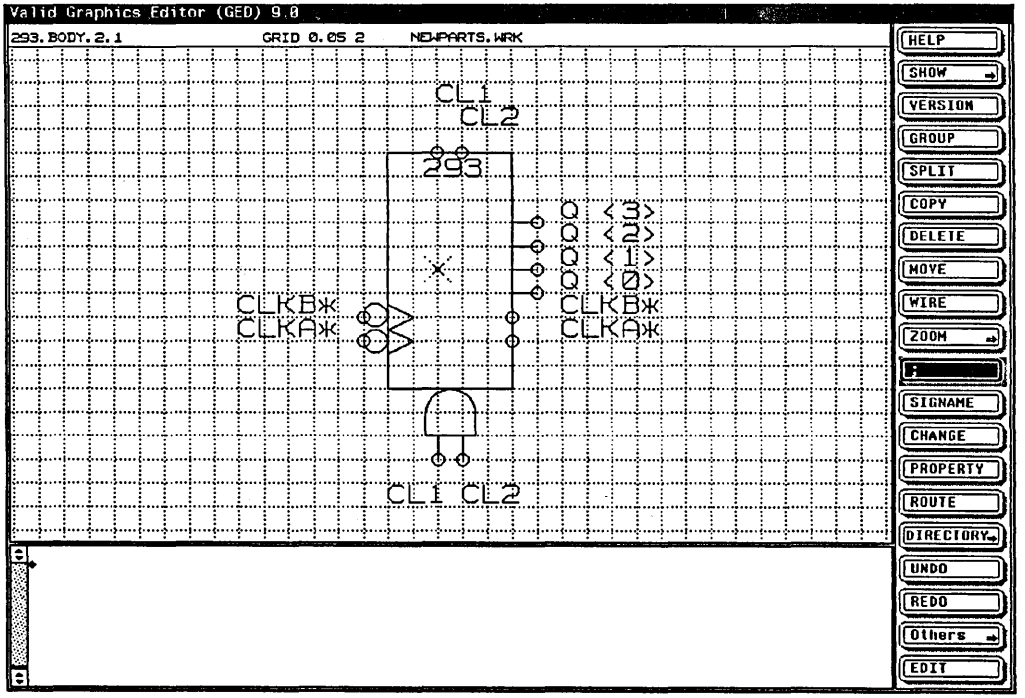


Figure 2-5. The TTL 293 Pin Names

Use the **show attachments** command to ensure that all pin names are properly attached.

- The pin names on a body must correspond exactly to those on the logic, timing, and simulation drawings associated with the body.
- Use the **signame** command to give the pass-through pin the same name as the parent pin.
- For gates, letter the input pins as found in the data books, or alphabetically. Letter the output pins as found in the data books, or use Y.
- The pipe character (|) is not allowed in pin names. This is to facilitate the syntax of the BUBBLE_GROUP property.

Adding Properties to Pin Names

There are two general properties that you can attach to pins when you enter the pin names:

- \NAC
- \NWC

These properties are only used on body drawings. They are not required in the corresponding timing and simulation models or logic drawings.

\NAC Property

The backslash is the general property prefix character.

\NAC stands for "no assertion check." \NAC allows a signal of either assertion to be connected to a pin. The Compiler assigns the assertion of the first signal connected to the pin as the pin's assertion. This forces any other signals connected to that pin to have the same assertion as the first signal. The \NAC property is used when the assertion level of signals is not important, but all the signals must be compatible.

\NWC Property

\NWC stands for "no width check." This property is used when the width (in bits) of a pin is not known or when it is desired that signals of any width may be connected to the pin. The Compiler determines the actual bit width from the context in which the \NWC property is used.

Figure 2–6 shows a sample merge body that carries both the \NAC and \NWC properties.

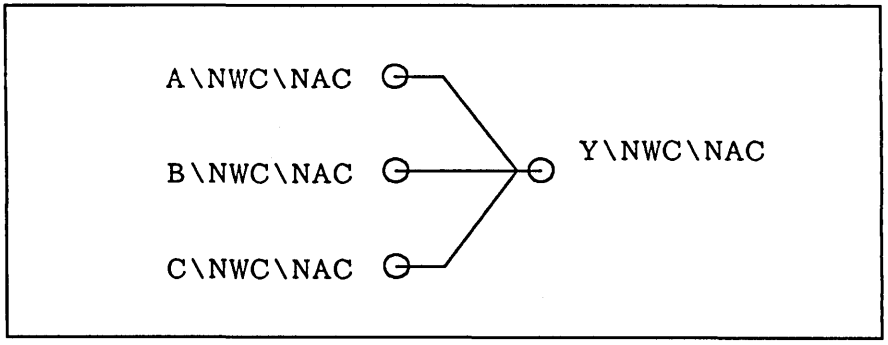


Figure 2–6. The \NAC and \NWC Properties

Annotating Bodies

A body should be annotated with:

- The component name
- The pin names
- Any other important information

Notes on pins are important since pin names do not appear when bodies are added to logic drawings.

Annotation makes the function of the body and each pin clear. The notes should be easily readable and should not be crowded. Follow these steps to annotate the TTL 293 body:

1. Use the **note** command and the notes shown in Figure 2–7. Center the notes on the pins *inside* the body and as close to the edge of the body as possible.

2. After placing the notes, make them smaller to minimize crowding. Use the **display** command to resize the notes as follows:

   display .8: All Q pins
   The CL pin
   Pin A
   Pin B

   display .6: Bit subscripts 0, 1, 2, 3
   The words BINARY CTR

3. Once the pins are the correct size, use the **move** command to realign the notes.

You can use a grid setting of **.01 10** *if necessary for better placement. Be sure to set the grid spacing back to* **.05 2** *after placing notes.*

The TTL 293 body now looks like the one in Figure 2–7.

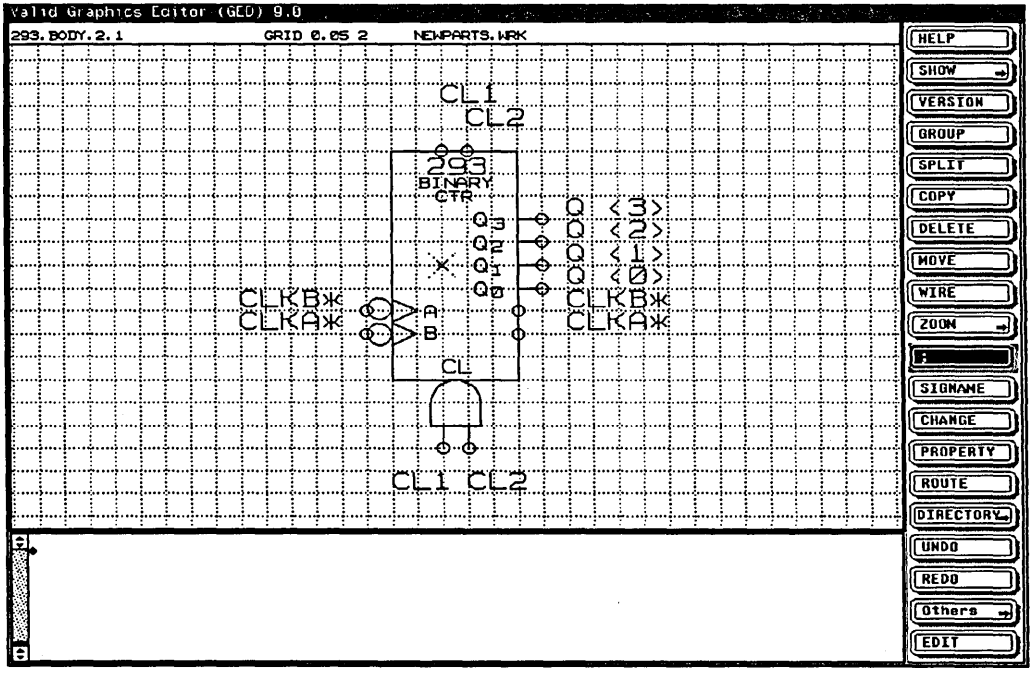


Figure 2–7. The Annotated TTL 293 Body

Annotation Standards

- Make the body name note default size for counters and shift registers, 0.8 for multiplexers and decoders, 0.75 for smaller gates, and 2.0 for VLSI chips.
- Mark open collector and open emitter pins with OC or OE placed immediately above the bubble or wire stub for the pin. Display the note 0.6 of the default size.
- Do not mark tri-state pins with a note.
- Clocks do not normally require a note; only DC clocks require a note.

Attaching Properties to the Body

A *property* is a name and value pair that conveys information about your design to the analysis tools. The information represented by the properties in a drawing is interpreted by the Compiler and then passed on to the other programs.

The standard body properties that can be attached to body drawings to affect component sizing and pin assertion level are:

- NEEDS_NO_SIZE
- HAS_FIXED_SIZE
- BUBBLED
- BUBBLE_GROUP

Use the same procedure to attach any body properties to the body origin.

Follow these steps to attach the NEEDS_NO_SIZE property to the origin of the TTL 293 component:

1. Select the **property** command from the GED menu.
2. Use the yellow button to select the body origin.
3. Type the name and value of the property:

   ```
   NEEDS_NO_SIZE=TRUE
   ```

   The word TRUE appears on the screen.
4. Use the yellow button to place the property value one-half grid interval above the body origin.

5 Type **display invisible** and point to the word TRUE. The property name and property value disappear.

Figure 2–8 shows the TTL 293 component with the NEEDS_NO_SIZE property attached and visible.

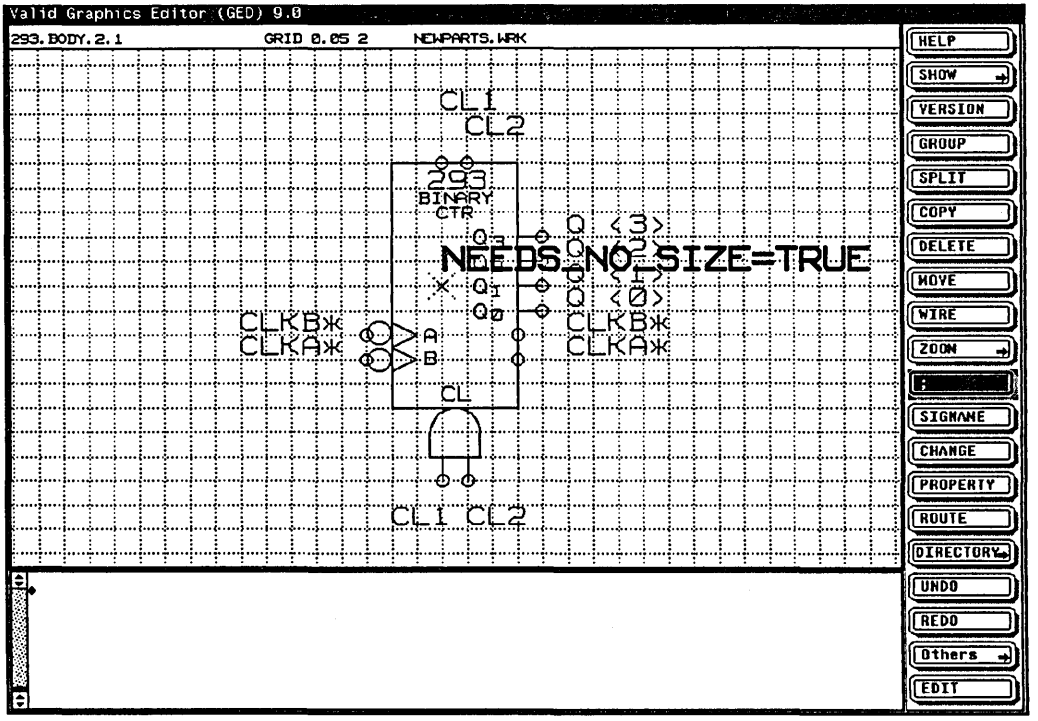


Figure 2–8. The TTL 293 Body with NEEDS_NO_SIZE Property

Once you have added the NEEDS_NO_SIZE property to the TTL 293 component, the basic drawing of Version 2 is complete. For information on building other body versions, see page 2–32.

NEEDS_NO_SIZE Property

The NEEDS_NO_SIZE property is attached to all versions of a component when there are no sizeable versions of the component.

- Version 1 of the body supports multiple-bit pins having a fixed number of bits (for example, PIN_NAME=Q <2..0>).
- Version 2 of the body shows each pin explicitly.

For more information on sizeable components and the SIZE property, see page 2-34.

Multiple-bit pins that support a fixed number of bits (vectored pins) are not affected by the SIZE property attached to the body when it is added to a logic drawing; therefore, the component "needs no SIZE." Attaching the SIZE property to a NEEDS_NO_SIZE component produces a Compiler error.

Figure 2-9 shows two versions of the LS138 component. Both versions have the NEEDS_NO_SIZE property attached; neither version is sizeable. Version 1 has a vectored input pin with a fixed number of bits; Version 2 has explicit input pins.

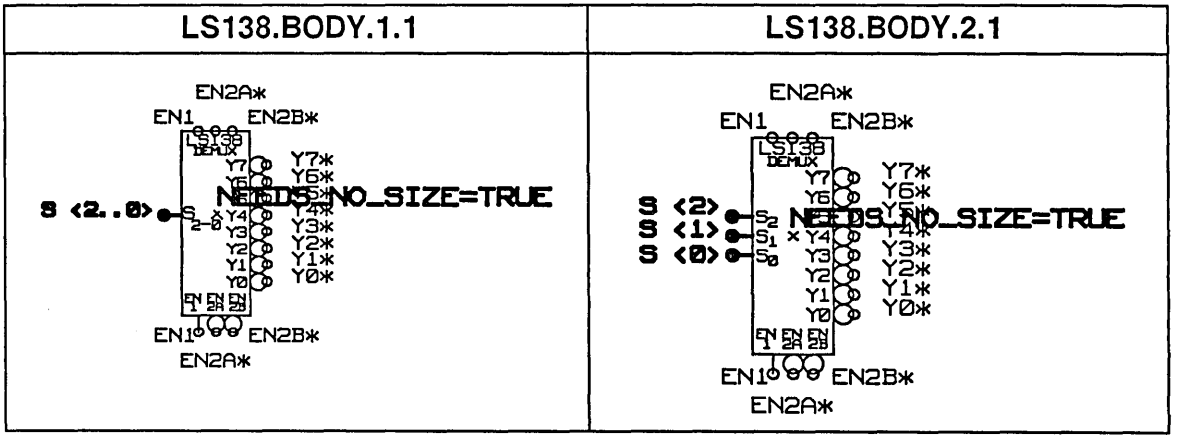


Figure 2-9. NEEDS_NO_SIZE Property

HAS_FIXED_SIZE Property

The other body property that affects component size is the HAS_FIXED_SIZE property. This property is attached to the non-vectored (explicit) version of a component when one version of the component is sizeable. HAS_FIXED_SIZE passes size information through to the simulation primitives.

- Version 1 of the component supports multiple-bit pins that include the SIZE parameter (for example, PIN_NAME=Q <SIZE-1..0>).

  SIZE specifies the number of multiple bits the pin represents. The value of SIZE-1 depends on the SIZE property attached to the body when it is added to a logic drawing. No SIZE property is attached to the body drawing during creation, only when the body is used in a schematic.

- Version 2 of the body shows each pin explicitly. This is the version that carries the HAS_FIXED_SIZE property.

  During body creation, you attach a value to the HAS_FIXED_SIZE property to define the final size of the component; therefore, the component "has a fixed SIZE." Attaching the SIZE property to a HAS_FIXED_SIZE component produces a Compiler error.

Figure 2-10 shows two versions of the LS367 component. Version 1 is sizeable. There is no size associated with the body drawing itself; the SIZE property is attached to the body when it is added to

a logic drawing. Version 2 has a fixed size of four bits (4B). Version 1 has vectored input and output pins; Version 2 has explicit input and output pins.

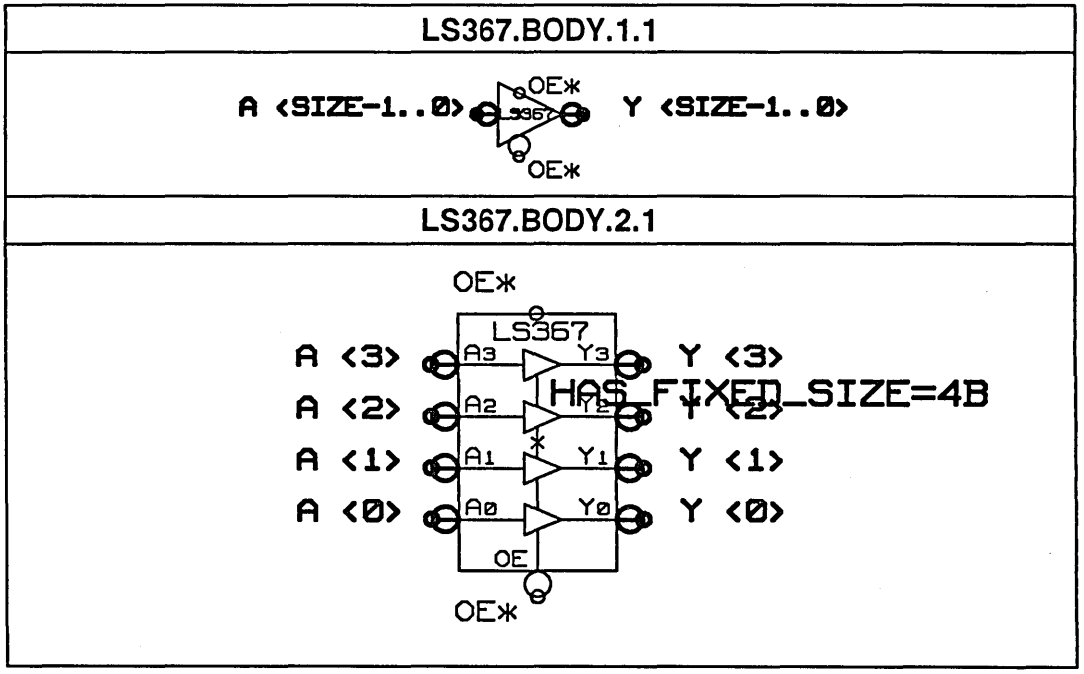


Figure 2–10. HAS_FIXED_SIZE Property

Invisible Properties

Some properties such as NEEDS_NO_SIZE and HAS_FIXED_SIZE are required properties that should not be changed by the user. These required properties, therefore, do not need to appear on your body drawing. You can make these properties invisible.

Position any invisible property 0.05 inches from the origin so that it may be easily located and not share the same coordinates as a visible body property.

- All body properties are attached to the body origin.
- The only invisible properties on a body should be section identifiers on asymmetrical bodies and the NEEDS_NO_SIZE and HAS_FIXED_SIZE properties. All other properties should be visible.
- An invisible property must not be in the same location as a visible object.
- When invisible properties are attached, locate them 0.05 inches from the origin.

BUBBLE_GROUP Property

Some library parts have pins that are logically related to each other, so that if you bubble one of these pins, you must also, by definition, bubble the other pin. An LS367 component permits the A input pin and the Y output pin to be bubbled. In fact, if one of these pins *is* bubbled, the other pin must also be bubbled. To prevent wiring errors, pins can be assigned to *bubble groups* so that when one pin is bubbled, other pins are also bubbled. For example, if pins A, B, and C are in one bubble group, when any one of these pins is bubbled, they are all bubbled. The BUBBLE_GROUP property is used on the body drawing to indicate which pins must bubble simultaneously.

Use the procedure on page 2-21 to add the BUBBLE_GROUP property to a body drawing.

The BUBBLE_GROUP property is attached to the origin of the body. Each BUBBLE_GROUP property defines one bubble group.

SYNTAX

BUBBLE_GROUP [*group_name*]=(*pin*|*pin*[|*pin*|*pin*] ...)

The *group_name* is an optional uppercase or lowercase single letter name. The *group_name* is not normally used; it is not necessary to name bubble groups except to define a bubble group that does not fit on a single line in the editor. (The limit is 80 characters.) All BUBBLE_GROUP properties of the same name define one large bubble group.

pin is the name of a pin to include in the bubble group. You can abbreviate pin names, but make sure you include enough of the pin name to make the name unique. For example, the minimum abbreviation for the pin names A<2>, A<1>, and A<0> is A<2, A<1, and A<0. Only the closing angle bracket can be omitted. The pin names Q <SIZE-1..0> and Q <SIZE-1..0>* must be typed explicitly since only the last characters of the pin names differ.

EXAMPLES

```
BUBBLE_GROUP=(A:Y)
BUBBLE_GROUP=(A <1:Y <1)
BUBBLE_GROUP=(I0:Y <SIZE-1..0>:Y <SIZE-1..0>*)
```

Figure 2-11 shows several BUBBLE_GROUP entries attached to the origin of the LS367 component. (The TTL 293 component does not require any bubble groups.)

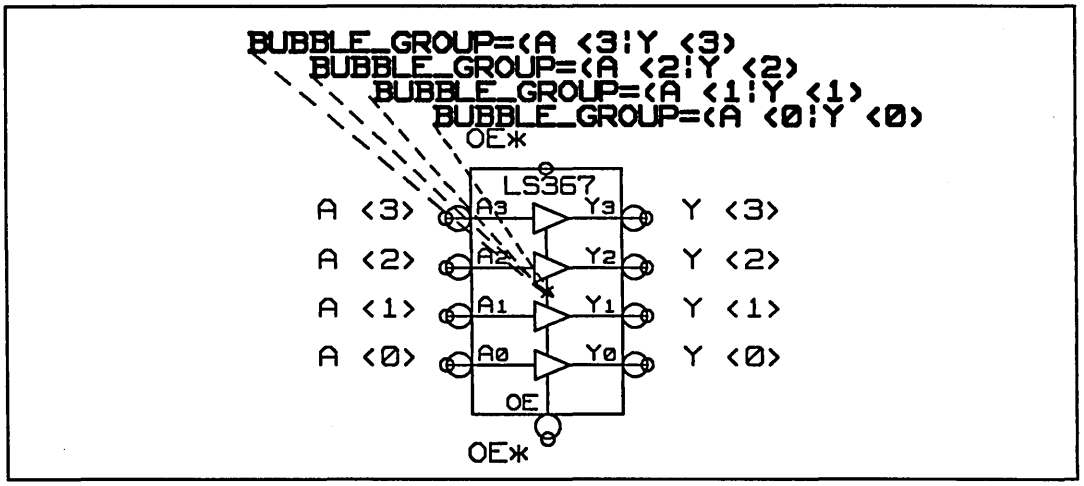


Figure 2-11. BUBBLE_GROUP Property

Asymmetrical Bubble Groups

A small number of bodies (such as XOR gates and parity generators) require an irregular arrangement of bubbled pins in a group. Asymmetrical bubble groups allow you to bubble some pins in the group while excluding other pins in the group. For example, if pins A and B are members of an asymmetrical bubble group, then bubbling pin A bubbles pin B, but bubbling pin B has no effect on pin A.

If an XOR gate has inputs A and B and output Y, the bubble behavior should be as follows:

If a bubble is added for:	There should be a bubble for:
A	Y
B	Y
Y	A (but not B)

There is no way to express this using conventional bubble groups, but it can be expressed as follows:

(A^Y)
(B^Y)
(Y^A)

An asymmetrical bubble group has the syntax:

SYNTAX

BUBBLE_GROUP=(*pin1^pin2|pin3|...*)

This means that if *pin1* is bubbled, all the other pins are bubbled, but if any of the other pins are bubbled, there is no effect on *pin1*.

Figure 2–12 shows asymmetrical BUBBLE_GROUP entries attached to the origin of the LS86 component.

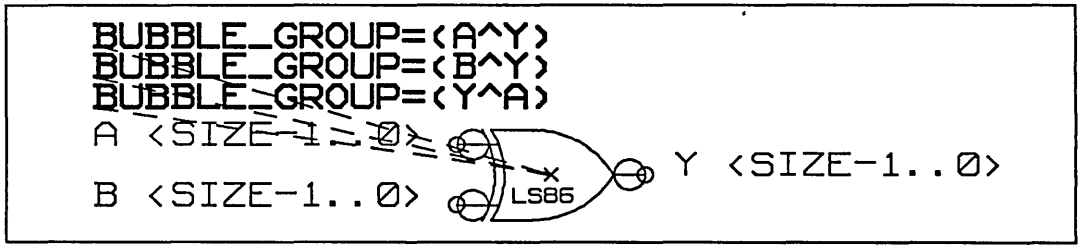


Figure 2–12. Asymmetrical Bubble Groups

BUBBLED Property

Once the bubble groups for a body are defined, the next step is to tell which pins start in the bubbled state and which start in the non-bubbled state. The BUBBLED property, also attached to the origin of the body, contains this information.

SYNTAX

BUBBLED=(*pin*[|*pin*|*pin*]...)

pin refers to the pins that are bubbled by default.

When you attach the BUBBLED property to a pin and add the component to a logic drawing in GED, the pin appears in the bubbled state, and the circle is displayed. If you enter the **bubble** command in GED and point to the bubbled pin, the pin toggles to the non-bubbled state and the line is displayed.

Figure 2-13 shows the LS04 component with both the BUBBLE_GROUP and the BUBBLED property attached to the body origin.

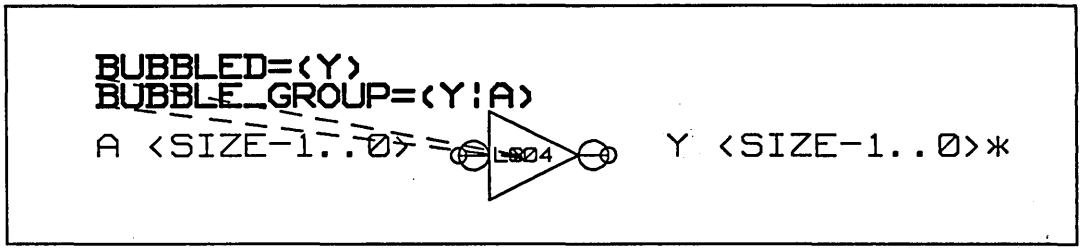


Figure 2-13. The BUBBLED Property

When you add the LS04 component to a logic drawing, the Y output pin is bubbled; the A input pin is not bubbled. When you enter the **bubble** command in GED and point to either pin, pin A is bubbled and pin Y is unbubbled.

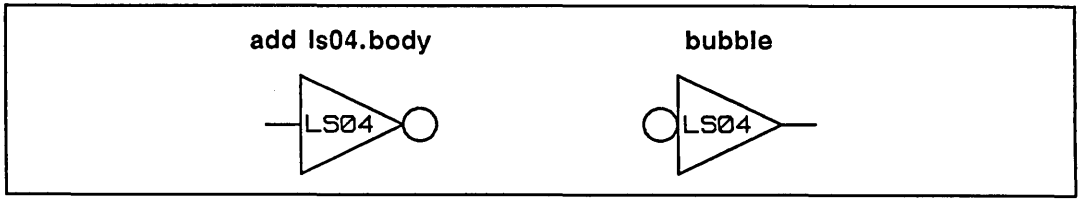


Figure 2-14. The Bubbled Pins in GED

Completing the Body Drawing

Once all the required properties are attached to the body drawing, follow these steps to complete the drawing:

1. Enter the **show attachments** command to verify that all property attachments are correct. Use the **reattach** command to correct any inaccurate attachments.

2. Enter the **write** command to check and save the drawing.

Building Other Body Versions

Once you complete the explicit version of a component, you can build other body versions of the same component. The most common variations are a vectored version and a sizeable version.

A *vectored* component has multiple-bit pins with a fixed number of bits. A *sizeable* component allows you to specify (with the SIZE property) the number of bits the part can represent when you add the component to a logic drawing.

Vectored Components

Pins on a vectored component can have vectored or scalar pin names. Vectored components are not affected by the SIZE property attached to the body when it is added to a logic drawing. A vectored component has the NEEDS_NO_SIZE property attached to the body origin.

Follow these steps to create a vectored version of the TTL 293 component. For more information on any step, refer to the page listed with each step.

1. Edit a new drawing called *293.body*. You do not need to specify the version and page number; the default is for the system to add version 1 and page 1 (*293.body.1.1*).

2. Use the **circle**, **wire**, **arc**, and **dot** commands to draw the body shape, the pins, the AND logic, the clock wedges, and the connection dots. Draw only one output pin instead of adding four individual output pins as you did on the Version 2 body. *(Page 2-9)*

Remember to add the connection dots to the pass-through clock and clear pins.

3 Use the **signame** command to add the pin names to the body. Use the pin name Q <3..0> for the single output pin. *(Page 2–16)*

4 Use the **note** command to annotate the body drawing. Use the **display** command to resize the notes and the **move** command to realign them if necessary. *(Page 2–19)*

5 Attach the NEEDS_NO_SIZE property to the body origin. *(Page 2–21)*

6 Write the drawing to save it.

Figure 2–15 shows a vectored version of the TTL 293 component. The specific differences between Version 1 and Version 2 are:

- Version 2 is 10 grid units high by 5 grid units wide. Version 1 is 8 units high by 5 units wide.
- Version 2 has four output pins with explicit pin names. Version 1 has one output pin with a vectored pin name that supports a fixed number of bits.

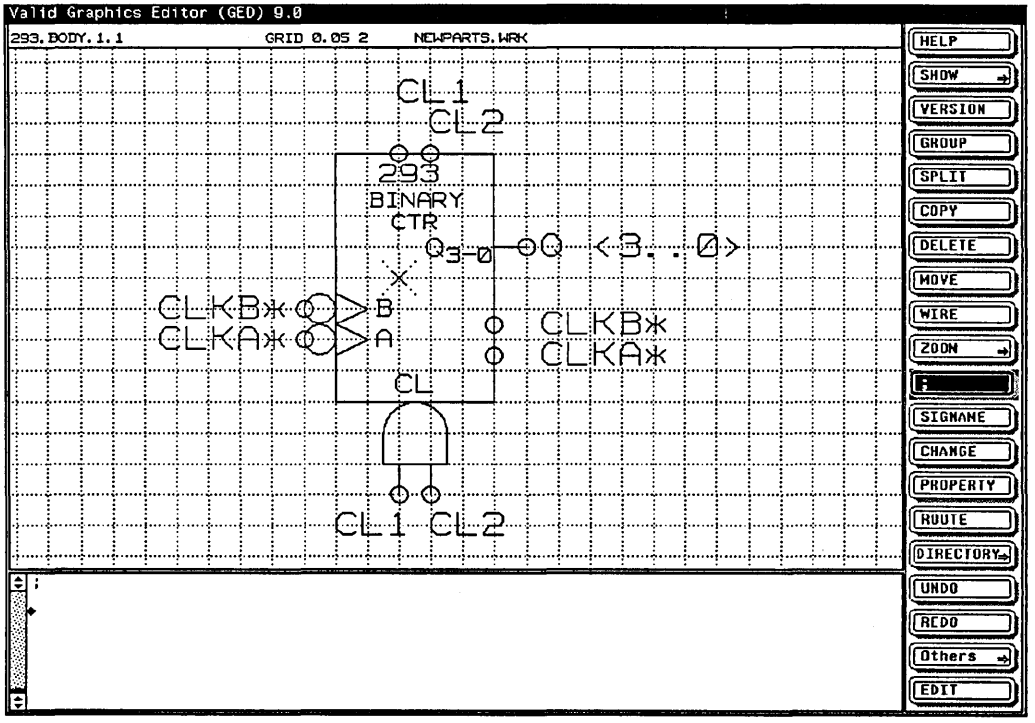


Figure 2–15. Version 1 of the TTL 293 Body

Sizeable Components

When you add a new part to a library, you must decide if it can be made into a sizeable part. If it is possible to make a part sizeable, you must then decide which control signals should be driven in parallel and which should be provided on a per–bit basis. At least one pin of a sizeable component must be a sizeable pin (SIZE–1..0); all other pins can be sizeable, vectored, or scalar pins.

Consider creating a D flip–flop that has a preset and clear for each section. Should these inputs be sizeable (meaning that each bit can be cleared and

set individually), or should these be single-bit signals that set and clear all the flip-flops simultaneously? The answer depends on how you expect the part to be used. The decision does not rule out any particular design, but it does make some designs easier to enter than others.

If you make the preset and clear sizeable, then a user who wishes to clear the entire register must extend the clear signal to the correct size to prevent a width mismatch. Since most sizeable flip-flops are used as registers, this is a good argument for making preset and clear single-bit signals.

On the other hand, if the signals are made single-bit and the user really needs a register where each bit can be asynchronously cleared and preset independently, you must draw the register with one body per bit. This is equivalent to the non-vectored design style for this particular register.

Not all parts can be made sizeable. In an ALU, for example, CARRY IN and CARRY OUT are connected neither in parallel nor on a per-bit basis. Therefore, an ALU is not a sizable part. Gates, on the other hand, require no control signals and can always be made sizeable.

Pin Names for Sizeable Components

See the SCALD Language Reference Manual *for more information on signal syntax in the different library formats.*

When you build a sizeable library component, the pins (or some of the pins) of the component must also be defined as sizeable. Parts may be made to handle an arbitrary number of bits by providing multiple-bit pins and vectored pin names with the SIZE-1 parameter. The width of the data path is then determined by the SIZE property placed on the component when it is added to a logic drawing.

For the standard Valid library format (format 1), a sizeable pin name is specified as:

PIN_NAME=*name* <SIZE-1..0>

The variable *name* is the name of the sizeable pin. The parameter SIZE-1 specifies the number of multiple bits the pin is to represent. The value of SIZE-1 depends on the value assigned to the SIZE property attached to the body when it is added to a logic drawing.

For example, if you attach the property SIZE=4B to a sizeable component (in a logic drawing) with a Y<SIZE-1..0> output pin, the resulting value is Y <3..0>. Attaching the property SIZE=2B results in the value Y <1..0>.

The top half of Figure 2–16 shows the sizeable body version (Version 1) of the LS78 component. The bottom half shows the component when you add it to a logic drawing and attach the SIZE property.

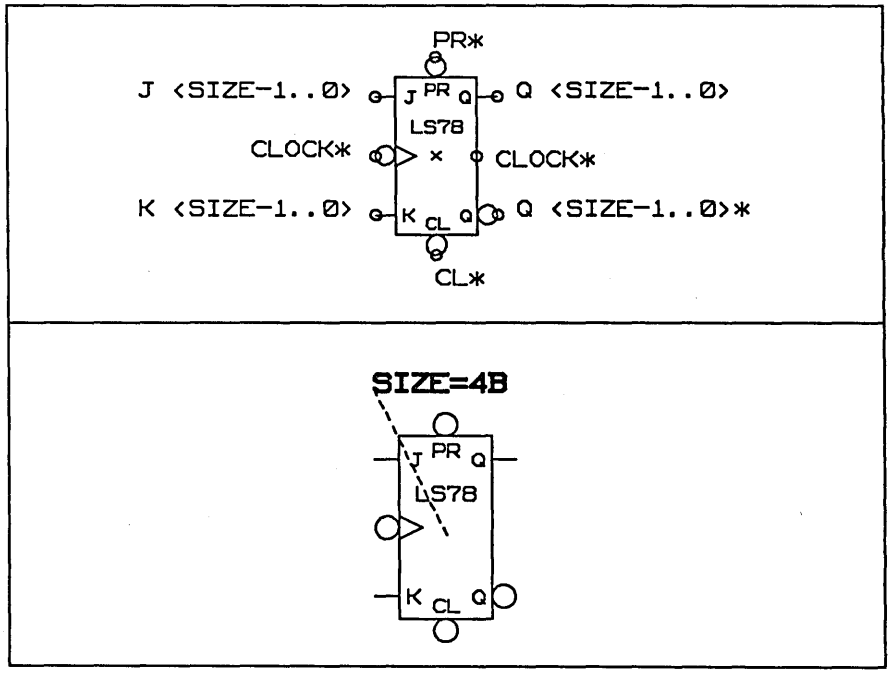


Figure 2–16. LS78 Sizeable Component

Modifying Existing Components

For some components you want to create, you may be able to copy an existing library component and alter the component for your use. For example, the TTL 293 component *looks* exactly like the LSTTL LS293 component; the differences are in the electrical characteristics. You can use GED to make a copy of the LSTTL component and modify it for use as a TTL component.

There are two methods of copying library components:

- Use the **smash** command on an added body drawing.
- Use the **diagram** command to change the name of a borrowed drawing.

Both of these are alternate methods of creating the TTL 293 component you already created in this section.

The Smash Command

Follow these steps to create Version 2 of the TTL 293 body by copying the LS293 component.

1 Edit a new drawing called *293.body.2.1*.

2 Add Version 2 of the LS293 component to the new drawing:

```
add ls293..2
```

Center the LS293 body around the origin of the new drawing.

3 Enter the **smash** command and point to the LS293 component. This command separates the body into individual wires, arcs, and circles. Any properties attached to the LS293 component are deleted.

4 Use the **delete** command to remove the incorrect body name (LS293).

5 Use the **split** command to move the new body name away from the body origin. Place the new body name at the top of the body and display it to the correct size.

6 Use the **dot** command to place connection points on each pin.

7 Use the **signame** command to add the pin names to the body.

8 Use the **property** command to attach the required properties to the body origin.

9 Write the drawing to save it.

The Diagram Command

Follow these steps to use the **diagram** command to create Version 2 of the TTL 293 drawing.

1 Edit Version 2 of the LS293 drawing:

```
edit ls293.body.2.1
```

2 Use the **diagram** command to rename the drawing:

```
diagram 293.body.2.1
```

The system brings up the same drawing under the new drawing name. All properties are attached exactly as they were under the original drawing name.

3 Enter the command:

```
ignore lsttl
```

This command removes the reference to the lsttl library so you can write the drawing into your own *.wrk* file.

4 Use the **write** command to save the drawing under the new drawing name.

5 Enter the command

```
library lsttl
```

to reference the lsttl library again.

6 Use the **change** and **move** commands to edit the body name and center it within the body.

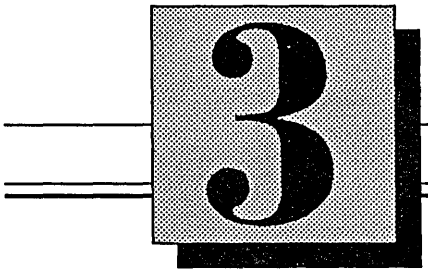
7 Write the drawing to save it.

Completing a Component

To complete a new component, you must:

- Create the physical model for the Packager (add pin numbers, output type, etc.)
- Create a simulation model (if necessary)
- Create a timing model (if necessary)
- Test the component

If you use the **smash** or **diagram** method and modify an existing component to create a new component, you also need to make the necessary changes to the .PART, .SIM, and .TIME drawings for the body to complete the new component.



The Physical Model

This section discusses:

- Creating a .PART drawing
- Modifying an existing .PART drawing
- Creating a library drawing
- Adding body properties
- Adding pin properties
- Adding a drawing body
- Completing a library drawing
- Modifying an existing library drawing
- Editing the *compiler.cmd* file
- Compiling a library drawing
- Creating individual chips files

Creating the .PART Drawing

The DRAWING body cannot be added to a .BODY drawing since it would appear with the body on the schematic.

Compiling a component without a .PART drawing produces the error message "no useable extension found."

The .PART drawing tells the Compiler that the component is a *primitive*, or lowest-level, body. It is not a hierarchical body; there is no logic below the body or inside it. There is only one item included in a .PART drawing: the DRAWING body.

The DRAWING body specifies properties of the entire component. You attach the TITLE property (which is the logical part name) and the ABBREV property (which is an optional abbreviation for the component name to be used when constructing path elements). If the physical part name is different than the body name, you must also attach the PART_NAME property. For example, the logical part name LS00 corresponds to the physical part name 74LS00.

Follow these steps to create a .PART drawing for the TTL 293 component:

1. Access GED and edit a file named *293.part.*
2. Add a DRAWING body to the file:

   ```
   add drawing
   ```

3. Select the **property** command from the menu and point to the DRAWING body.
4. Add the TITLE, ABBREV, and PART_NAME properties below the word DRAWING:

   ```
   title 293
   abbrev 293
   part_name 74293
   ```

5 Enter the command

```
display both
```

and point to each property.

6 **write** the drawing to save it. LAST_MODIFIED now reflects the current date and time.

The .PART drawing for the TTL 293 component is shown in Figure 3–1.

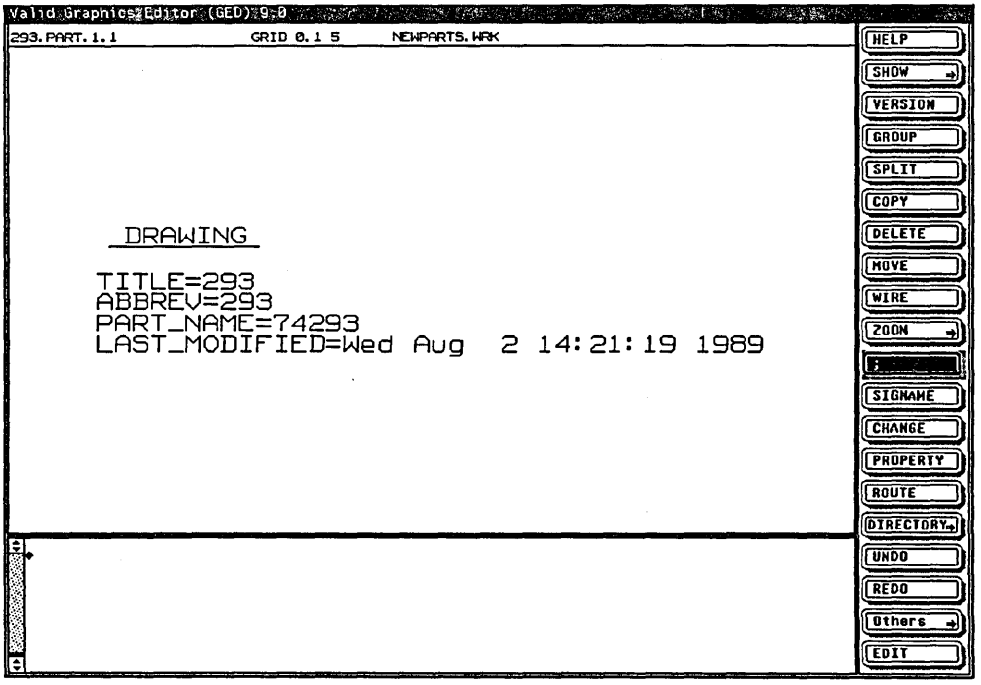


Figure 3–1. The 293.PART Drawing

Modifying an Existing .PART Drawing

If you copy an existing component and modify it to create a new component, you should also copy the existing .PART drawing and modify the DRAWING body to match the new drawing name, abbreviation, and part name.

Creating the Physical Model: Basic Procedure (Checklist)

Creating a physical model by adding physical information to a library component requires the following steps:

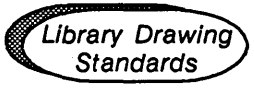
1. Create a library drawing and add the sizeable or vectored version of the body.
2. Attach the body properties.
3. Attach the pin properties.
4. Verify property attachments, check and write the drawing.
5. Edit the *compiler.cmd* file.
6. Compile the library drawing.
7. Move the *chips.dat* file to the individual component directory and rename it *chips_prt*.

Each of these steps is detailed in this section.

Creating the Library Drawing

See Archiving Library Drawings, *page 3–31, for more information on archival.*

Refer to Appendix B for changes between the previous library drawing method and the current library drawing.



The *library drawing* is used to transfer physical information from GED to the Packager. This drawing shows the sizeable version of a component to be packaged (usually Version 1).

The name of the library drawing should reflect the component you are modeling. You can keep or discard the library drawing for a component depending on your choice of archival methods.

Follow these steps to create a library drawing for the TTL 293 component:

1 Access GED and edit the library drawing:

```
edit newparts library
```

2 Add the sizeable version (Version 1) of the component to be compiled:

```
add 293.body
```

- ✓ Parts having asymmetrical sections should have one of each of the sections added to the drawing.
- ✓ Asymmetrical sections cannot have any common pin names.

Adding Body Properties

Several body properties are attached to the origin of a component on the library drawing. These properties pass information about the component to the Compiler or other Valid analysis tools:

- FAMILY
- POWER_PINS
- BODY_TYPE
- COST
- PART_NUMBER
- PHYS_DES_PREFIX

On the UNIX operating system, there is a text file method of adding physical information. Refer to Appendix A for details.

Follow these steps to add the FAMILY and POWER_PINS properties to the TTL 293 component in the NEWPARTS LIBRARY drawing:

1 Select the **property** command from the GED menu.

2 Enter the FAMILY property for the 293 component:

```
FAMILY = TTL
```

3 Enter the POWER_PINS property:

```
POWER_PINS=(VCC:14;GND:7)
```

The TTL 293 component and body properties appear in Figure 3-2.

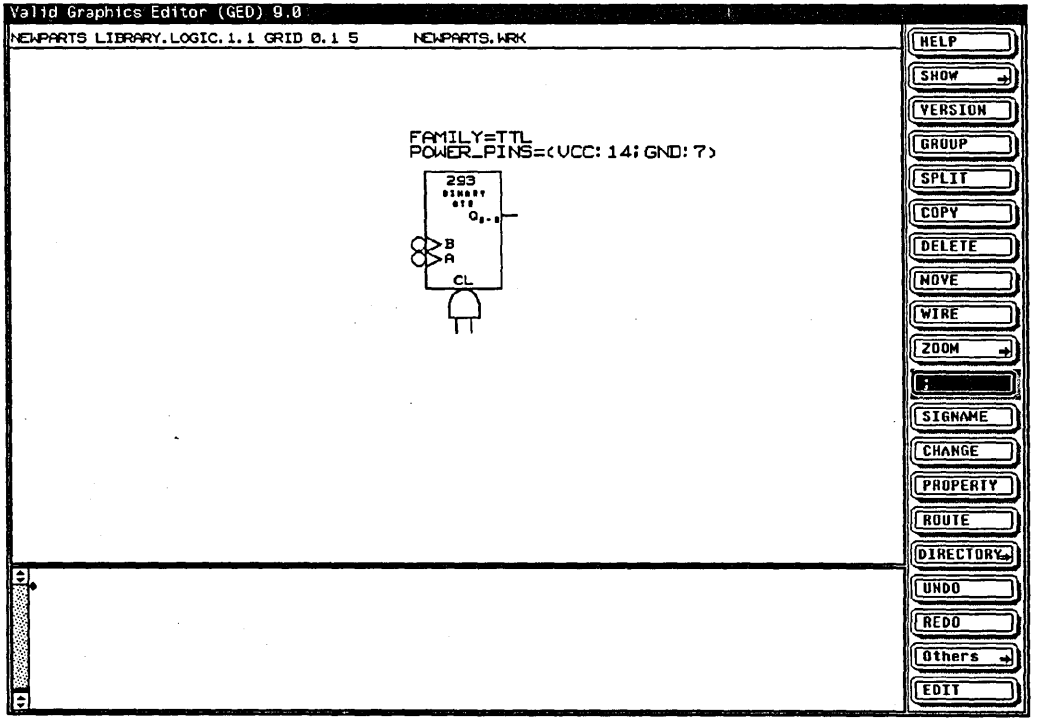


Figure 3-2. The TTL 293 Body Properties

FAMILY Property

The FAMILY property specifies the logic family of a component. The property can have any value. Some of the values used in the Valid libraries are:

FAMILY = TTL
FAMILY = LSTTL
FAMILY = ECL100K

Attach the FAMILY property to the body origin.

POWER_PINS Property

The POWER_PINS property specifies the power and ground pin assignments for each component. The property specifies both the power supply (power rail) names and the pin numbers of pins connected to the supply.

SYNTAX

POWER_PINS=(*supply*:*pin list*; *supply*;*pin list*; ...)

supply

The power supply name. The name must be an alphanumeric identifier starting with a letter (for example, VCC). The supply name may include an underscore (_).

pin list

A list of the pin numbers of the part that connect to the supply. The pin numbers are separated with commas.

A colon is used to separate the supply name and the pin list. A semicolon is used to separate each supply specification.

EXAMPLES

The TMS4050 RAM has the POWER_PINS property:

```
POWER_PINS=(VBB:1; VDD:10; VSS:18)
```

The 100123 bus driver has the property:

```
POWER_PINS=(VCC:6; VCCA:7,9,11,5,3,1; VEE:18)
```

The order of the pin assignments is not important. The POWER_PINS property only applies to library parts and is ignored if attached to a body that is not a library part. Attach the POWER_PINS property to the body origin in the library drawing.

BODY_TYPE Property

You can also use the property COMMENT_BODY=TRUE *to comment out a body.*

The BODY_TYPE property is used to "comment out" bodies that have no logical function. For example, a company logo requires a BODY_TYPE property because the logo has no logical meaning. The BODY_TYPE property has the following syntax:

BODY_TYPE=COMMENT

COST Property

The COST property specifies the cost of a component at your site. This property is not used in the Valid libraries since it contains site-specific information. Since this property is not a constant value, it might be handled more easily using physical part tables. See the *ValidPACKAGER Reference Manual* for information on physical part tables.

PART_NUMBER Property

The PART_NUMBER property is used to assign an internal part number for the component. This property is not used in the Valid libraries.

PHYS_DES_PREFIX Property

The PHYS_DES_PREFIX property allows you to change the prefix of the physical descriptor to match your requirements. For example, you can use physical reference designator of "IC" for an integrated circuit rather than the default "U."

Adding Pin_Number Properties

Each pin (except pass-through pins) of every library component (and all versions of a component) must have a PIN_NUMBER property attached. The PIN_NUMBER property tells the Packager:

- The pin number for the pin
- The number of sections of the component in a package
- The pin numbers for each section

There must be one pin number on each pin for each section. For example, if a package has four sections, there must be four pin numbers on each pin. The Packager prints an error message when a pin is found without a PIN_NUMBER property or without the correct number of pins per section.

Follow these steps to add the PIN_NUMBER property to the TTL 293 component in the NEWPARTS LIBRARY drawing:

1 Choose the **property** command from the GED menu and select the CLKA pin.

2 Enter the PIN_NUMBER property for the CLKA pin:

```
PIN_NUMBER=(10)
```

Locate the property to the left of the CLKA pin.

3 Select the CLKB pin and type the PIN_NUMBER property for the pin:

```
PIN_NUMBER=(11)
```

Locate the property above the CLKA pin number.

4 Select the CL1 pin and type the PIN_NUMBER property for the pin:

```
PIN_NUMBER=(12)
```

Locate the property near the CL1 pin.

5 Select the CL2 pin and type the PIN_NUMBER property for the pin:

```
PIN_NUMBER=(13)
```

Locate the property near the CL2 pin.

The pin number sequence follows the pin name sequence (Q<3..0>).

6 Select the vectored Q pin and type the vectored PIN_NUMBER property for the pin:

```
PIN_NUMBER=(<9,5,4,8>)
```

Locate the property near the Q pin.

The pin number properties for the TTL 293 component are shown in Figure 3–3.

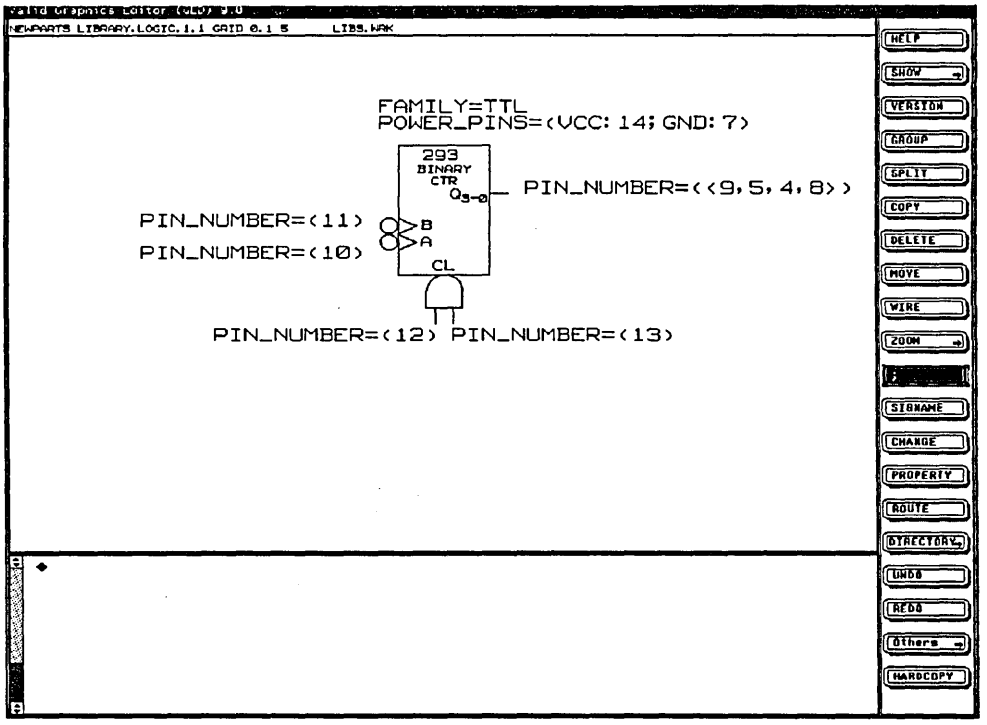


Figure 3–3. The TTL 293 Component with Pin Numbers

Pin Number Formats

There are different pin number formats available depending on the type of library part you are developing. The formats define the following types of pin assignments:

- Single section scalar pin
- Single section vector pin
- Multiple section scalar pin
- Multiple section common pin
- Multiple section common vector pin
- Asymmetrical components

A *scalar* pin is a pin that corresponds to a single-bit signal. A *vector* pin corresponds to a fixed number, multiple-bit signal. A *pin_id* (the PIN_NUMBER property) consists of any alphanumeric character and/or the underscore character. The maximum length of a pin number is 16 characters.

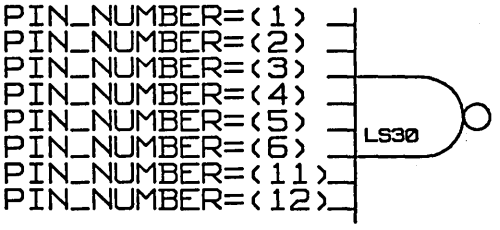
Single Section Scalar Pins

The PIN_NUMBER property for each pin of a simple one-section part has the format:

PIN_NUMBER=(*pin_id*)

The pins of the LS30 NAND gate are examples of single section scalar pins.

The clock and clear pins of the TTL 293 component are also examples of single section scalar pins.



Single Section Vector Pins

A vector pin has a fixed number of bits. It is not affected by the SIZE property. Each bit of the vector connects to the same section of the part. The PIN_NUMBER format for a vector pin is similar to the format for a scalar pin, except that each logical pin number includes several physical pin numbers enclosed between left and right angle brackets:

PIN_NUMBER=(< *pin_id*, *pin_id*, ... >)

The angle brackets indicate that the pin represents multiple bits. The pin numbers in the list are sepa-

rated by commas. For example, a 4–bit pin is specified as follows:

The bits of a multiple–bit pin are assigned to the physical pin numbers specified in the PIN_NUMBER property in the following manner:

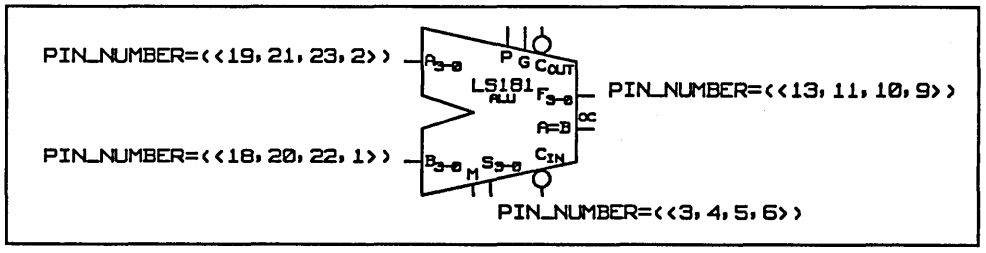
1. The bit having the lowest subscript is assigned to the first pin number in the list.

2. The bit with the next lowest subscript is assigned to the second pin in the list, and so on.

The least significant bit is assigned first because there must always be at least one bit per pin.

```
PIN_NUMBER=(<1,2,4,5>)
```

The data buses of an LS181 ALU are examples of single section vector pins.



The Q pin of the TTL 293 component is also an example of a single section vectored pin.

Multiple Section Scalar Pins

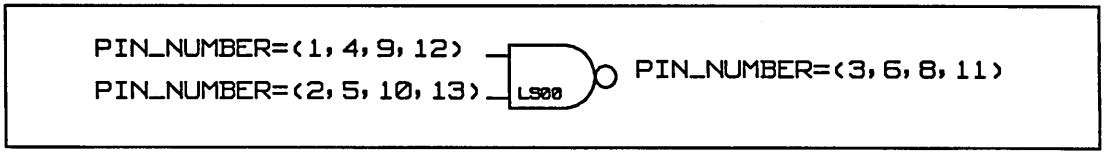
Many physical components contain several identical logical components. Each logical component is considered a *section* of the physical component. For example, the LS00 logical component is a single 2-input NAND gate; the 74LS00 physical component contains four 2-input NAND gates, or four sections. The pins on this type of component are defined as sizeable pins.

To identify the sections of a component, each pin of the logical component has a PIN_NUMBER property that contains a list of pin numbers, one pin number for each section in the part. For a multiple section scalar pin, the PIN_NUMBER property has the following syntax:

PIN_NUMBER=(*pin_id*, *pin_id*, *pin_id*, *pin_id*, ...)

The pin numbers for each section are separated by commas. Pins for the first section are in the last position, pins for the second section are in the second-last position, and so on.

The LS00 component is an example of a sizeable component with multiple section scalar pins. If the component is given the property SIZE=4B, each logical pin of the component has four pin numbers, one for each section:



Multiple Section Common Pins

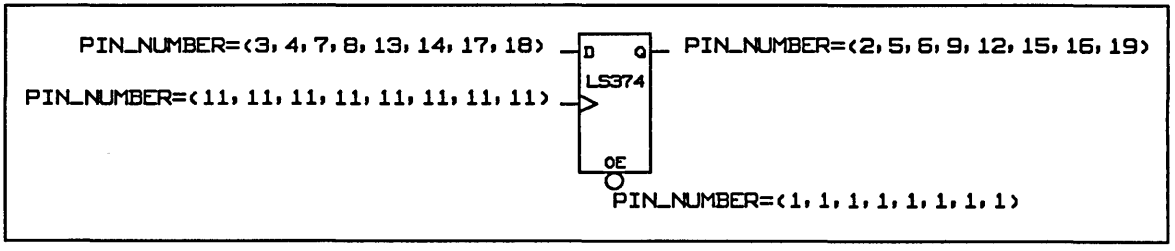
Some multiple-section components have pins that are common to several sections. For example, the LS374 octal register has eight sections with a common clock pin and a common output enable pin.

The syntax for the PIN_NUMBER property is the same for a multiple section common pin as for a multiple section scalar pin:

PIN_NUMBER=(*pin_id*, *pin_id*, *pin_id*, *pin_id*, ...)

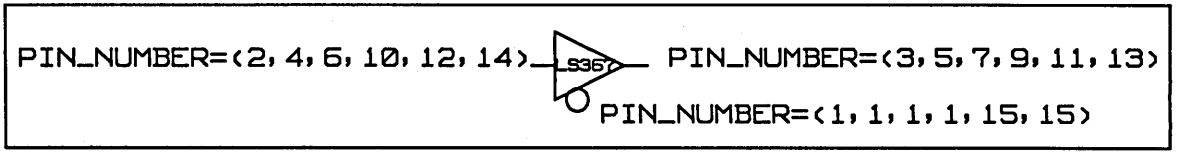
The pin numbers for each section are separated by commas. Pins for the first section are in the last position, pins for the second section are in the second-last position, and so on. Since common pins have the same pin number for each section, and each section must have a pin number entry, the common pins have identical PIN_NUMBER entries.

Each logical pin of the LS374 component has eight pin numbers, one for each section:



The clock and enable pins are common to all eight flip-flops in the package. The D and Q pins are defined so that one bit is assigned to each flip-flop.

Some components with multiple-sections have pins that are common only to certain sections of the part. These are represented the same way as pins that are common to all sections, except that the pin numbers are present only in the sections for which they are common. For example, the LS367 hex bus driver component has the following pin number assignments:



The pin numbers for the open emitter pin show that one open emitter pin is common to four sections of the component (pin 1), and one open emitter pin is common to the other two sections of the component (pin 15).

Multiple Section Common Vector Pins

If a multiple-section component has vectored pins, the PIN_NUMBER property has the following syntax:

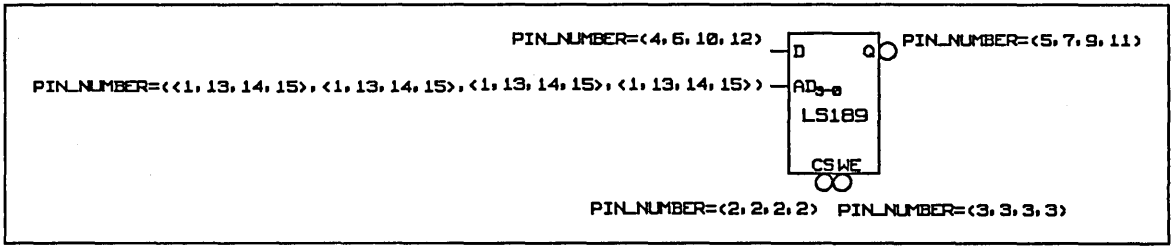
PIN_NUMBER=(<*pin_id*,*pin_id*,...>,<*pin_id*,*pin_id*,...>, ...)

Each pin number and section are separated by commas. The pin numbers enclosed in angle brackets specify individual bits of the pin, not different sections for the pin.

For example, a 3-bit pin in a component with two sections might be specified as:

PIN_NUMBER=(<1,2,3>,<5,6,7>)

An LS189 16-word by 4-bit RAM is an example of a component having vector pins and multiple sections.

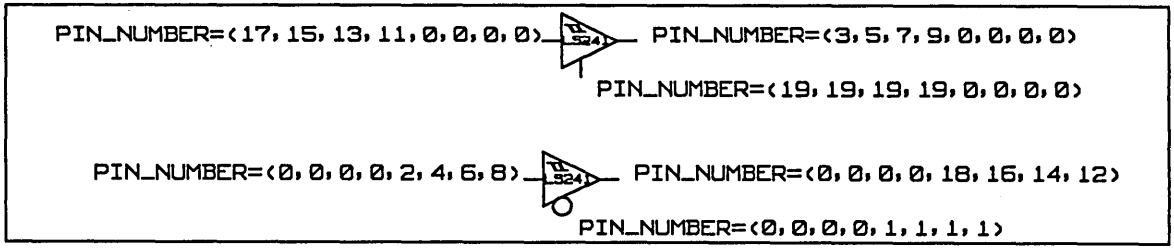


Asymmetrical Components

Some components have multiple sections that are functionally different. In this case, one version of the body is defined for each type of section in the part. To identify which pins are present in a given section, the pins of the different versions all have **different** pin names.

Even though some pins may not be present in a section, the PIN_NUMBER property values for the pins specify all the sections of the part. Any pin that is not present in a given section is specified with a pin number of 0.

For example, the LS241 bus transceiver has four buffers with active-high enables and four buffers with active-low enables. The pin numbers are defined as follows:



Compact Pin Number Syntax

Rather than entering each individual pin number for all the pins of a body, the PIN_NUMBER property allows you to enter pin numbers in a more compact syntax. The syntax allows you to abbreviate bit subranges and repeated sections.

When specifying a subrange, the range values must be separated by two periods (..) for all library formats.

The *subrange* compact syntax allows a list of bit subscripts to be abbreviated to include only the first and last subscripts. The *repeat section* syntax allows identical lists of bit subscripts to be abbreviated to one list and reiterated using a multiplier character. *Combinations* of these two syntaxes can be used to identify more complex pin numbers.

The subrange and repeat section functions also accept alphanumeric pin designations. The beginning and ending pin numbers of an alphanumeric subrange must:

- Begin with the same set of characters
- End with different integers

For example, PIN_NUMBER=(DA7..DA0) is legal while the PIN_NUMBER=(DA7..AA0) is not legal.

Table 3–1 shows an example of each compact syntax. The long pin number under "combination syntax" is shown on two lines for documentation purposes only.

Table 3-1. Compact Pin Number Syntax

Pin Type	Subrange Syntax	Abbreviation
Scalar	PIN_NUMBER=(7,6,5,4,3,2,1,0)	PIN_NUMBER=(7..0)
Vector	PIN_NUMBER=(<7,6,5,4,3,2,1,0>)	PIN_NUMBER=(<7..0>)
	Repeat Section Syntax	
Scalar	PIN_NUMBER=(7,7,7,7)	PIN_NUMBER=(7 * 4)
Vector	PIN_NUMBER=(<1,2,3,4>,<1,2,3,4>)	PIN_NUMBER=(<1..4> * 2)
	Combination Syntax	
Scalar	PIN_NUMBER=(3,3,3,3,4,4,26)	PIN_NUMBER=(3 * 4, 4 * 2, 26)
Vector	PIN_NUMBER=(<2,3,4,5>,<2,3,4,5>, <2,3,4,5>,<23,26,27,28>)	PIN_NUMBER=(<2..5> *3, <23,26,27,28>)

Adding Other Pin Properties

Besides the PIN_NUMBER property, there are several other standard pin properties attached to the pins of each component on the library drawing. These standard pin properties are:

- **OUTPUT_LOAD**
- **INPUT_LOAD**
- **BIDIRECTIONAL**
- **PIN_GROUP**
- **UNKNOWN_LOADING**
- **NO_LOAD_CHECK**
- **NO_IO_CHECK**
- **ALLOW_CONNECT**
- **OUTPUT_TYPE**

If you are attaching more than one pin property to a pin, you only need to select the pin once. That pin becomes the default attachment point for the **property** *command.*

Follow these steps to add the remaining pin properties to the TTL 293 component:

1 Select the **property** command from the GED menu and select the Q output pin.

2 Enter the OUTPUT_LOAD property for the Q pin:

```
OUTPUT_LOAD=(16.0,-0.8)
```

Locate the property below the PIN_NUMBER property for the Q pin.

3 Enter the INPUT_LOAD property for the CLKA and CLKB pins. Although the pin properties are identical, attach one INPUT_LOAD property to *each* pin:

```
INPUT_LOAD=(-3.2,0.08)
```

Locate the property below the PIN_NUMBER properties for the CLKA and CLKB pins.

4 Enter the INPUT_LOAD properties for the CL1 and CL2 pins. Although the pin properties are identical, attach one INPUT_LOAD property to *each* pin:

```
INPUT_LOAD=(-1.6,0.04)
```

Locate the properties below the PIN_NUMBER properties for the CL1 and CL2 pins.

The loading properties for the TTL 293 component are shown in Figure 3-4.

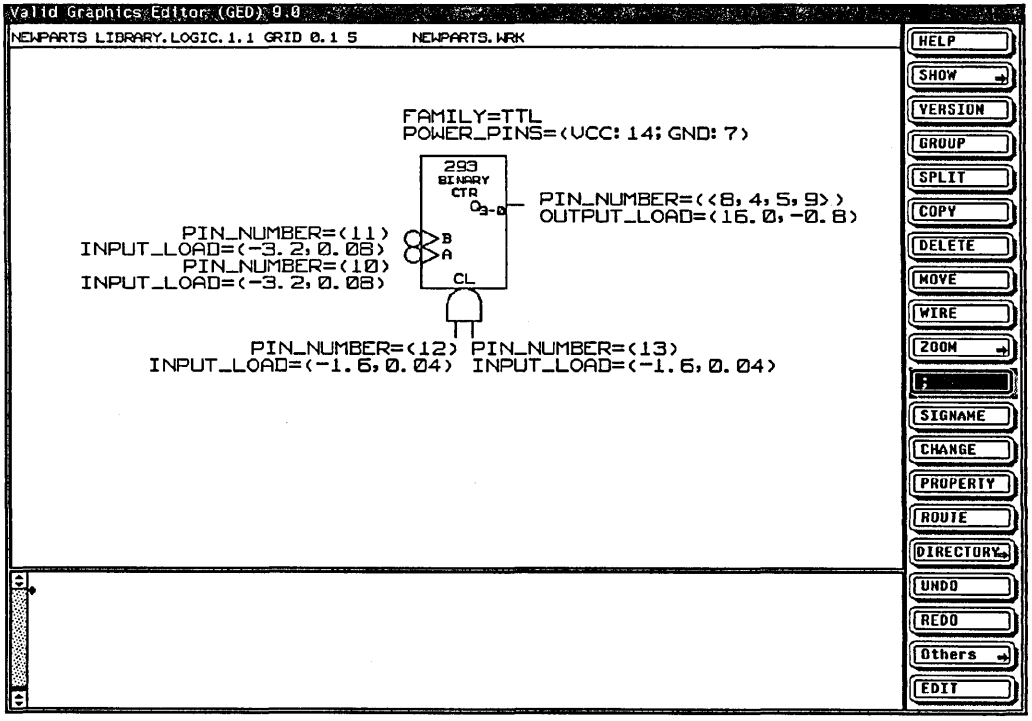


Figure 3-4. The TTL 293 Pin Loading Properties

- All pins must have either the OUTPUT_LOAD or INPUT_LOAD property or both properties.
- If the pin is an output (has the OUTPUT_LOAD property) and can be wire-tied to another output, it must be given the OUTPUT_TYPE property.
- If the pin is both an input and an output, the BIDIRECTIONAL property is required.

OUTPUT_LOAD Property

The OUTPUT_LOAD property is attached to any pin that is an output pin. The syntax of the OUTPUT_LOAD property is:

OUTPUT_LOAD=(*low current,high current*)

See the ValidPACKAGER Reference Manual *for more information on loading properties.*

OUTPUT_LOAD is typically measured in milliamps. If there is a different measurement used in the data book, convert the measurement to milliamps.

The Q pin of the TTL 293 component carries the OUTPUT_LOAD property:

```
OUTPUT_LOAD=(16.0,-0.8)
```

If the loading on a component is not a critical factor, there is a variable OUTPUT_LOAD syntax:

```
OUTPUT_LOAD=(*,*)
```

INPUT_LOAD Property

The INPUT_LOAD property is attached to any pin that is an input pin. The syntax of the INPUT_LOAD property is:

INPUT_LOAD=(*low current,high current*)

INPUT_LOAD is typically measured in milliamps. If there is a different measurement used in the data book, convert the measurement to milliamps.

The input pins of the TTL 293 component carry different INPUT_LOAD properties:

A & B inputs: `INPUT_LOAD=(-3.2,0.08)`
Clear inputs: `INPUT_LOAD=(-1.6,0.04)`

If the loading on a component is not a critical factor, there is a variable INPUT_LOAD syntax:

```
INPUT_LOAD=(*,*)
```

BIDIRECTIONAL Property

A pin with the INPUT_LOAD and OUTPUT_LOAD properties attached is an output pin which may also load the net. To make the pin bidirectional, attach the BIDIRECTIONAL pin property. A pin is *not* considered bidirectional unless the BIDIRECTIONAL property is attached. The syntax of the property is:

BIDIRECTIONAL=TRUE

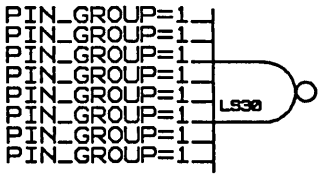
PIN_GROUP Property

The PIN_GROUP property assigns the pins of a component to pinswap groups. The Packager and the GED command **pinswap** use the PIN_GROUP property to identify swappable pins.

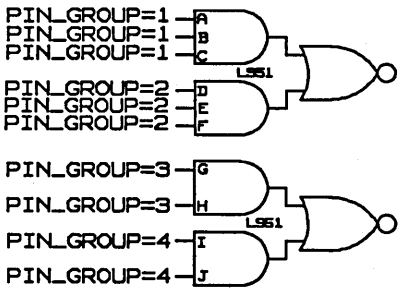
A *pinswap group* contains those pins that are logically equivalent and belong to the same section. If two signals are swapped between two pins that are in a pinswap group, the logical behavior of the circuit is not altered. The PIN_GROUP property has the syntax:

PIN_GROUP=*number*

Each pin in a pinswap group must have a PIN_GROUP property attached. All the PIN_GROUP properties must have the same value. The value of the PIN_GROUP property is not important, only that all pins of a pinswap group have an identical value. Any pin without the PIN_GROUP property cannot be swapped with any other pins.



For example, the input pins of an LS30 NAND gate (A, B, C, D, E, F, G, and H) all belong to the same pinswap group. For each pin, the PIN_GROUP property has the property value "1." The PIN_GROUP property is not attached to the output pin (Y*) since this pin cannot be swapped with any other pin on the component.



The LS51, 2-wide 3-input, 2-wide 2-input AND-OR-INVERT gate is an example of an asymmetrical component with more complex pinswap groups. There are four pinswap groups, one for each AND gate. The inputs for each AND gate are equivalent and therefore swappable. However, inputs from different AND gates are not swappable, so the PIN_GROUP properties have a different value.

The output pins Y0* and Y1* do not have PIN_GROUP properties because they cannot be swapped with any other pins.

OUTPUT_TYPE Property

The OUTPUT_TYPE property is added to the output pins of open-collector, open-emitter, and tri-state gates. The property serves three purposes:

- Allows the pin to be connected to other outputs.
- Specifies the type of output so that only compatible outputs can be connected together.
- Specifies the logic function created by tying the outputs together.

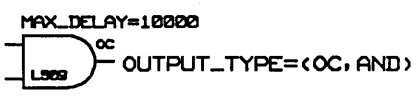
The OUTPUT_TYPE property provides information that is needed by the Packager, the Timing Verifier, and the Simulator. The property must appear on both the timing and simulation models as well as in the library drawing.

Each output pin that can be connected to other output pins must have an OUTPUT_TYPE property. The property value specifies the pin type and also the logic function created by tying the outputs together. The form of the OUTPUT_TYPE property value is:

OUTPUT_TYPE=(*output type*,*logic function*)

Output type can be open collector, open emitter, or tri-state (TS). *Logic function* can be AND, OR, or tri-state (TS). Be sure there is no space after the comma in the property value. The output type and logic functions can be combined as follows:

OUTPUT_TYPE=(OC,AND)	Open collector, AND logic function
OUTPUT_TYPE=(OE,OR)	Open emitter, OR logic function
OUTPUT_TYPE=(TS,TS)	Tri-state, tri-state logic function



The LS09 component shows the open collector, AND logic function OUTPUT_TYPE property.

Since tri-state pins are considered both input and output pins, they need both INPUT_LOAD and OUTPUT_LOAD properties. When a pin is in tri-state mode, the tri-state loading is specified as the INPUT_LOAD.

Load Checking Properties

The properties NO_LOAD_CHECK, NO_IO_CHECK, and ALLOW_CONNECT can be attached to a single pin, a body, or a net (where they check all pins on the net). The UNKNOWN_LOADING property can be attached to a single pin or to a body. See the *ValidPACKAGER Reference Manual* for more information on the load checking properties.

NO_LOAD_CHECK Property

When the NO_LOAD_CHECK property is attached to a pin, it suppresses device loading calculations for that pin. You can suppress the load check for the LOW state, the HIGH state, or BOTH high and low states.

NO_IO_CHECK Property

When NO_IO_CHECK is attached to a pin, it suppresses the input/output checking for that pin. You can suppress the input/output check for the LOW state, the HIGH state, or BOTH high and low states.

ALLOW_CONNECT Property

Attach the property ALLOW_CONNECT=TRUE if there are pins which require multiple outputs of different types to be connected together.

UNKNOWN_LOADING Property

If there are parts in a design that have pins with unknown or unspecified loading, use the property UNKNOWN_LOADING=TRUE to suppress load and I/O checks on a pin.

Completing the Library Drawing

Once all the required properties are attached to the library drawing, follow these steps to complete the drawing:

1. Enter the **show attachments** command to verify all properties are present and correctly attached. Use the **reattach** command to correct any inaccurate attachments.
2. **write** the drawing to save it.

Modifying an Existing Library Drawing

If you are copying an existing library and modifying components in that library, the best way to create the library drawing for the new library is to borrow the library drawing from the library where you borrowed the parts. If you are not changing the physical information, then you don't need to change any of the information on the library drawing other than its title and abbreviation. If you need to change physical information, use the GED **change** command to change the values of the pin and body properties on the library drawing.

Creating the Physical Model File

When the library drawing is correct and complete, you must make the physical model file from the library drawing.

The physical model is used by the Packager and by the **section** and **pinswap** commands to read the pin numbers you attached to the component during the design stage. To create the physical model, you need to:

- Edit the *compiler.cmd* file.
- Compile the library drawing.
- Rename and relocate the resulting data file.

Follow these steps to create the physical model for the TTL 293 component:

1 Edit the *compiler.cmd* file as follows:

- Change the ROOT_DRAWING directive to reflect the correct drawing name.
- Change the output type to "chips."
- Make sure the DIRECTORY directive points to the correct *.wrk* file.

2 Compile the library drawing to create the *chips.dat* file.

Check the cmplst.dat *file if any compilation errors occur. Correct any errors and recompile the drawing until an error-free run occurs.*

3 After the drawing compiles successfully, change the name *chips.dat* to *chips_prt* and move the physical model to the correct component subdirectory:

```
mv chips.dat 293/chips_prt
```

Archiving Library Drawings

Once the physical information is complete and the *chips_prt* file exists, the library drawing is no longer required. You can either archive the library drawing or delete it.

You can archive library drawings by:

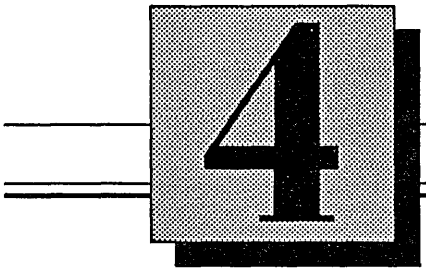
- Creating a "save" directory to store all library drawings.
- Using a tape archive utility (such as *tar*) to save library drawings off the system onto tape.

If disk space is at a premium, you can delete the individual library drawings and just save the *chips_prt* files for each component.

If you must modify a component after you delete a library drawing, you can either:

- Recreate the library drawing, incorporate the old and new information, and recompile the drawing.
- Use a text editor to make changes directly to the *chips_prt* file.

Once you decide on a method of modifying components, whether updating and recompiling the library drawing or editing the *chips_prt* file, you should continue to update that component in the same way. If you edit *chips_prt* and make a correction one time, and then the next time you make a change you modify the archived library drawing, the change you made to the *chips_prt* file is lost when you recompile the drawing and recreate the file. Deciding on one archival method to use saves you from inadvertently losing file changes.



The Simulation Model

This section describes:

- Defining the simulation model
- General design rules for models
- Delay and pulse width standards
- Creating the model (checklist)
- Simulator primitives
- Simulation properties
- Modifying simulation models

Defining the Simulation Model

A *simulation model* emulates the behavior and operation of a design without having to physically build the circuit. Simulation ensures the design's quality early in the design process, when changes and corrections are easier and less expensive to implement.

Simulation models are built from a specific set of parts called *simulator primitives.* You decide which simulator primitives to use by studying the functional specification and data tables in the appropriate data book. It is possible to create different simulation models (using different simulator primitives) for the same component, and obtain the same simulation results.

General Design Rules for Models

There are several important goals you should keep in mind when designing simulation models:

- Keep the model simple. Models do not need to reflect the complete logical behavior of the part in order to provide accurate information. Simple models are easier to design, easier to understand, easier to test, and execute faster. However, the model should be as large and complex as necessary to make it easy to understand. Ease of understanding by the user is better than incremental improvements in execution.
- Try to make the layout of the model follow the layout of the body. The interface signals in the model should appear in approximately the same physical relationship as on the body.
- Try to make it possible to understand all error messages without having to refer to the model.

Reference everything back to the device itself (the component being modeled).

- The user will not understand the internal structure of the models and does not want to have to look at them. Any errors during timing analysis, for instance, must be referred to signal names the user understands. This will not be the case if the model has a lot of unnamed signals. Models should always be designed so that error messages are reported with signal names that mean something to the user.

- All timing checker primitives (such as MIN PULSE WIDTH and SETUP/HOLD) should have their inputs connected to interface signals. When this is not possible, signals internal to a model (local signals) should be given names that describe the signal. Checker bodies have negligible impact on verification time.

When applying this rule remember that for all practical purposes, NOT bodies are wires.

- Do not connect sign–extenders or mergers to the interface signals of a body. (This generates confusing synonyms.) Place a zero–delay, non–inverting buffer of the appropriate SIZE between such structures and the interface signal.

- Many parts have both a true and complement output. Simulator primitives have only a true output. To generate both outputs, use an inverting and a non–inverting buffer, one buffer driving the complemented output and the other buffer driving the un–complemented output.

Delay and Pulse Width Standards

The following standards are used in calculating delay and pulse width information for the simulation and timing models in the Valid libraries.

Calculating Delays

Data books frequently do not specify all three delay values (minimum, typical, and maximum). Table 4–1 shows how to calculate delay values based on the available values. When two values are present, calculate both of the values listed. For the minimum or typical value, choose the lesser of the two values to represent the missing value. For the maximum value, choose the greater of the two values to represent the missing value.

Table 4–1. Calculating Delays

One value present:	Minimum equals	Typical equals	Maximum equals
Minimum	–	2 times min	3 times min
Typical	1/2 of typ	–	2 times typ
Maximum	1/3 of max	1/2 of max	–
Two values present, missing value is:	**Use value of Minimum**	**Use value of Typical**	**Use value of Maximum**
Minimum	–	1/2 of typ	1/3 of max
Typical	2 times min	–	1/2 of max
Maximum	3 times min	2 times typ	–

Data-Dependent Delays

For a data-dependent delay example, see the LS244 simulation model.

Most delays are functions of the value of the data. Rise delay is usually different from fall delay. The library part models usually include both rise and fall delays. In models of tri-state parts, however, including both rise and fall delays would add to the complexity of the model. Therefore, tri-state parts are modeled with a single delay from enable to output, without consideration of whether the output is rising or falling.

Open Collector Gates

Open collector gates have no fixed, maximum time delay. It is not possible for the analysis program to compute the maximum delay from the GED drawing. It is up to the designer to calculate the explicit maximum delay for each open collector gate.

When an open collector gate is added to a GED drawing, the property MAX_DELAY appears above the body with a value of 10000ns. The designer must use the **change** command in GED to change this value to the required value.

Pulse Width

If minimum delays are not specified in the data book, use a minimum pulse width equal to one-half of the period of the input. For example, if the maximum toggle frequency is 10 MHz, estimate the pulse width as follows:

$$Period = \frac{1}{10 \times 10^{6}} = 0.1 \times 10^{-6}$$

$$Minimum\ pulse\ width = \frac{0.1 \times 10^{-6}}{2} = 50\ ns$$

One-Shots

One-shots have no fixed pulse width. It is not possible for the analysis program to compute the pulse width from the GED drawing. It is up to the designer to calculate the pulse width, taking into account all the one-shot tolerances, external component tolerances, temperature variations, and drift over the life of the circuit.

When a one-shot is added to a GED drawing, the property PULSE_WIDTH appears above the body with a value of 10000ns. The designer must use the **change** command in GED to change this value to the required value.

Creating the Model: Checklist

Creating a simulation drawing requires the same basic steps as creating a logic drawing except that the parts used are logic simulator primitives from the Sim Library and support components from the Valid Standard library.

1 Access the Sim library:

```
library sim
```

2 Create a drawing for your component with a .SIM extension (for example, 293.SIM).

3 Add the required simulator primitives.

4 Add a DRAWING body and attach the TITLE and ABBREV properties.

5 Wire the model.

6 Name the input and output signals. Signal names must include the \I interface signal property and correspond to the names of the signals in the body drawing for the part being modeled.

7 Assign the required simulation properties.

8 Verify attachments, check, and write the model.

- Use a B SIZE PAGE as a border.
- Center the drawing on the page.
- Include the name of the drawing and the initials of the creator in the boxes in the lower right hand corner of the page border.
- Enter the page number of the drawing as a note (text size 1.5) in the form "1 of 1."
- Include a note block (notes enclosed with wires to form a block) to document any assumptions and/or critical design decisions that are not obvious to the user.
- Add primitives only from the Standard and Sim libraries.
- Every model must have a DRAWING body (with TITLE and ABBREV properties attached).
- Follow SCALD signal syntax for signal names.
- Do not use bit lists in bit subscripts.
- Make sure all interface signals have the \I property in the signal name.
- All interface signals should have an explicit width specified unless the signal is a scalar.
- All properties attached to bodies should be placed above the body or to the right. Place the properties one above the other and left-aligned. Display both the property value and name for all properties except PATH.

Since it is possible to create different simulator models for the same component, there is no step-by-step description for creating a simulation model for the TTL 293 component. Figure 4–1 shows one possible simulation model for the TTL 293. Simulator primitives and simulation properties are discussed in general following the drawing.

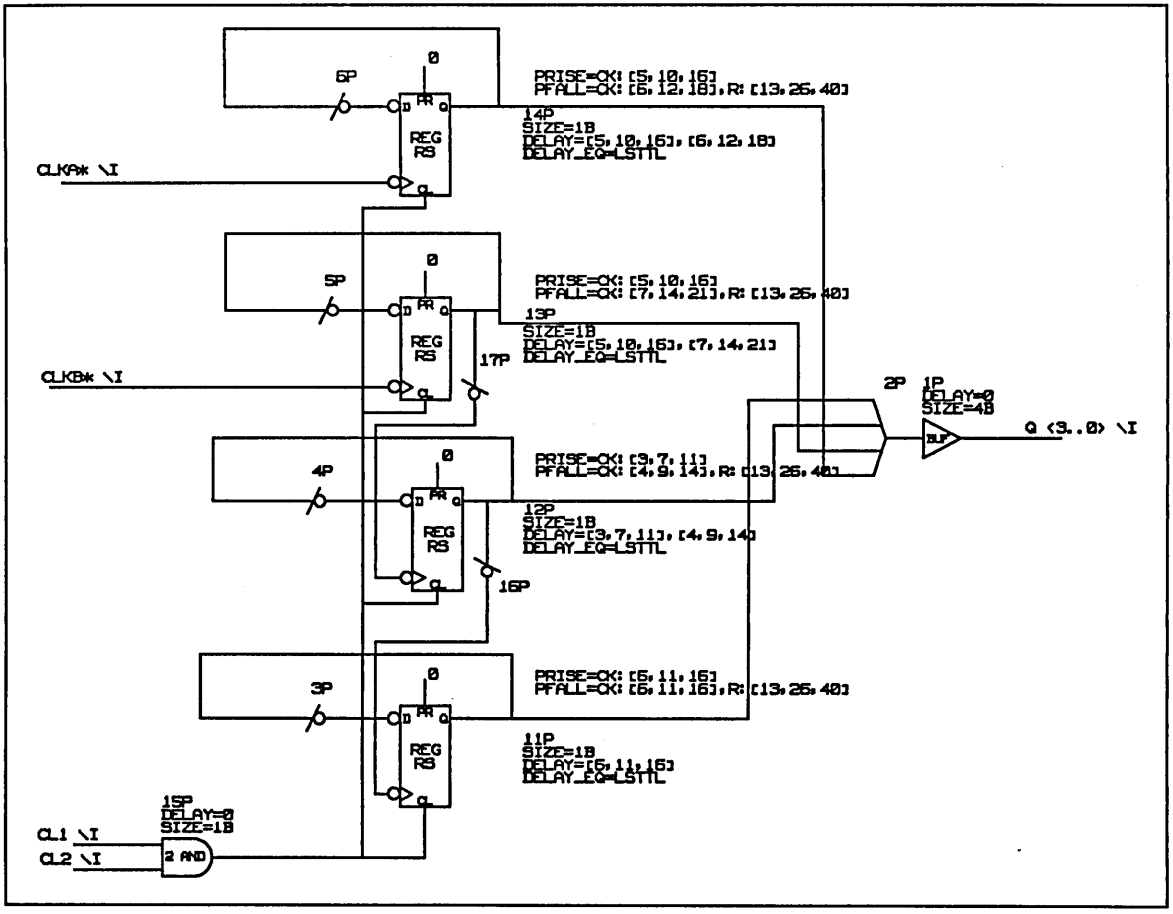


Figure 4–1. TTL 293 Simulation Model

The Simulator Primitives

For additional information on simulation primitives and directives, see the ValidSIM Reference Manual.

The simulation primitives are stored in the Sim library. Many simulator primitives are available, from simple logic gates to a complete ALU. The behavior of each primitive is understood by the Logic Simulator.

Sometimes there are components and primitives that have similar names, for example, the "2AND" component and the "2 AND" primitive. Be sure to leave a space in the primitive names.

Bubbled Pins

Each input and output pin on a primitive can be individually bubbled using the GED **bubble** command. Bubbling a primitive pin inverts the logical function of the primitive. An AND gate with a bubbled output behaves as a NAND gate. Bubbling the output pin of the simulator primitive BUF adds an inverter to your model.

Truth Table Abbreviations

Table 4–2 shows the abbreviations used in the truth tables for the simulation primitives.

Table 4–2. Truth Table Abbreviations

Abbreviation	Meaning
ps	Previous state
U	Unknown value
X	Can be any value
Z	High impedance
→	Transition
≠	Not equal to

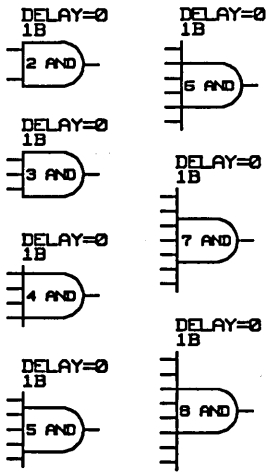
The Logic Gate Primitives

There are three types of logic gate primitives:

- **AND**
- **OR**
- **XOR**

Since any pin of any primitive can be independently bubbled, to create a NAND gate, simply bubble the output of an AND gate.

AND Primitive



There are seven AND primitives (two-input through eight-input):

- 2 AND
- 3 AND
- 4 AND
- 5 AND
- 6 AND
- 7 AND
- 8 AND

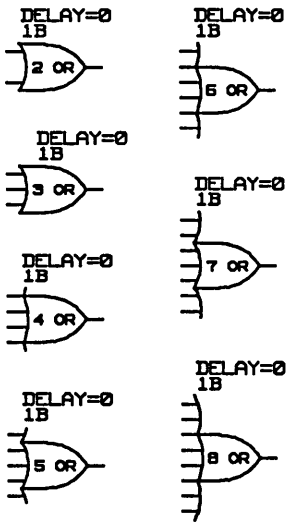
Be sure to leave a space between the numeral and the "AND."

The truth table for an AND primitive is shown in Figure 4-2.

One or More Inputs	All Other Inputs	Output
0	X	0
1	1	1
Z,U	1	U

Figure 4-2. AND Gate Truth Table

OR Primitive



There are seven OR primitives:

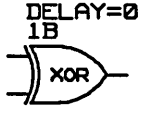
- 2 OR
- 3 OR
- 4 OR
- 5 OR
- 6 OR
- 7 OR
- 8 OR

The truth table for an OR primitive is shown in Figure 4–3.

One or More Inputs	All Other Inputs	Output
0	0	0
1	X	1
Z,U	0	U

Figure 4–3. OR Gate Truth Table

XOR Primitive



The XOR has only a two-input version. The truth table for an XOR primitive is shown in Figure 4–4.

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0
X	U,Z	U
Z,U	X	U

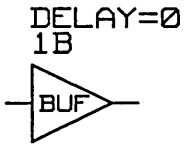
Figure 4–4. XOR Gate Truth Table

The Buffer Primitives

There are three buffer primitives:

- BUF
- TS BUF
- IDENTITY

BUF Primitive



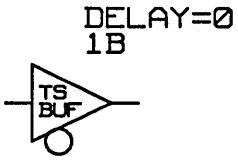
The truth table for the simple buffer primitive BUF is shown in Figure 4–5.

Input	Output
0	0
1	1
Z,U	U

Figure 4–5. BUF Truth Table

To create an inverting buffer, simply bubble the input or output pin of a buffer. Non–inverting buffers are commonly used for delays.

TS BUF Primitive

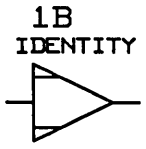


The tri-state buffer primitive TS BUF has an enable input that, when disabled, causes the output to take the value high impedance (Z). The enable input has a width of one bit. The truth table for the TS BUF is shown in Figure 4-6.

Input	Enable*	Output
0	0	0
1	0	1
Z,U	0	U
X	1	Z
X	Z,U	U

Figure 4-6. TS BUF Truth Table

IDENTITY Primitive

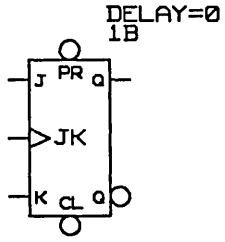


The IDENTITY primitive is similar to BUF except that it propagates the exact signal on the input pin to the output pin, while the BUF primitive converts the Z state to U and soft values to hard values. The truth table for the IDENTITY primitive is shown in Figure 4-7.

Input	Output
0	0
1	1
Z	Z
U	U

Figure 4-7. IDENTITY Truth Table

The JK Primitive



The JK primitive models the J-K Flip Flop. The primitive has the following features:

- Input pins for J and K data inputs
- Asynchronous set and reset functions
- An edge-triggered clock

If the clock input is not bubbled, then the primitive output triggers on a positive edge; if it is bubbled, it triggers on a negative edge. Outputs consist of Q and Q* data outputs. Asserting both the set and reset pins causes both of the outputs to go high. The truth table for the JK primitive is shown in Figure 4-8.

J	K	CLOCK	PR*	CL*	Q	Q*
X	X	X	Z,U	X	U	U
X	X	X	X	Z,U	U	U
X	X	X	0	0	1	1
X	X	X	0	1	1	0
X	X	X	1	0	0	1
X	X	X→Z,U	1	1	U	U
X	X	X→0	1	1	ps	ps
0	0	0→1	1	1	ps	ps
0	1	0→1	1	1	0	1
1	0	0→1	1	1	1	0
1	1	0→1	1	1	≠ Q	≠ Q*

Figure 4-8. JK Primitive Truth Table

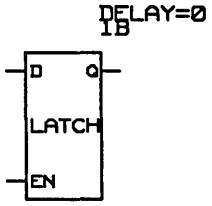
The Latch Primitives

There are three latch primitives:

- **LATCH**
- **LATCH RS**
- **LATCH RS COMP**

LATCH Primitive

The LATCH primitive has an enable input that is level-sensitive. The truth table for the LATCH primitive is shown in Figure 4-9.

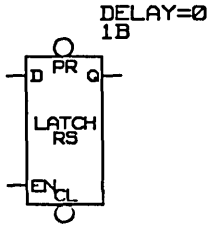


Two other latch primitives, SCAN LATCH *and* SCAN LATCH RS, *appear in the Sim library. These primitives are no longer supported by Valid.*

Data	Enable	Q
0	1	0
1	1	1
X	0	ps
Z,U	1	U
≠ ps	Z,U	U
= ps	Z,U	ps

Figure 4-9. LATCH Truth Table

LATCH RS Primitive

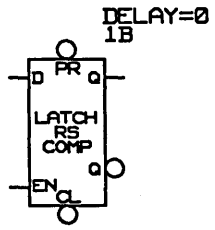


The LATCH RS primitive has an enable input that is level-sensitive and asynchronous set and reset inputs that cause the outputs to take the values one and zero, respectively. On the LATCH RS primitive, reset prevails over set if both are asserted. The truth table for the LATCH RS primitive is shown in Figure 4–10.

Data	Enable	PR*	CL*	Q
X	X	X	Z,U	U
X	X	X	0	0
X	X	Z,U	1	U
X	X	0	1	1
= ps	Z,U	1	1	ps
≠ ps	Z,U	1	1	U
X	0	1	1	ps
0	1	1	1	0
1	1	1	1	1
Z,U	1	1	1	U

Figure 4–10. LATCH RS Truth Table

LATCH RS COMP Primitive



The LATCH RS COMP primitive has an enable input that is level-sensitive and asynchronous set and reset inputs that cause the outputs to take the values one and zero, respectively.

Complementary outputs are provided on the LATCH RS COMP primitive, and both outputs take the value one when both set and reset are asserted. The truth table for the LATCH RS COMP primitive is shown in Figure 4–11.

Data	Enable	PR*	CL*	Q	Q*
X	X	X	Z,U	U	U
X	X	Z,U	X	U	U
X	X	0	0	1	1
X	X	0	1	1	0
X	X	1	0	0	1
= ps	Z,U	1	1	ps	ps
≠ ps	Z,U	1	1	U	U
X	0	1	1	ps	ps
0	1	1	1	0	1
1	1	1	1	1	0
Z,U	1	1	1	U	U

Figure 4–11. LATCH RS COMP Truth Table

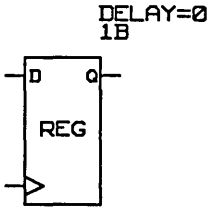
The Register Primitives

There are five register primitives:

- REG
- REG RS
- REG RS COMP
- REG RS COMP 2
- REG CKE

The register primitives have an edge-triggered clock input. When the clock input is not bubbled, the primitive outputs trigger on a positive edge; when the clock input is bubbled, the outputs trigger on a negative edge. The REG RS, REG RS COMP, and REG RS COMP 2 also have asynchronous set and reset inputs that cause the outputs to take the values one and zero, respectively.

REG Primitive

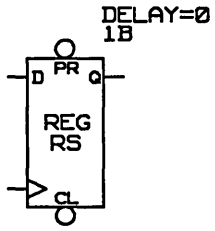


The truth table for the REG primitive is shown in Figure 4–12.

Data	Clock	Output
0	0→1	0
1	0→1	1
Z,U	0→1	U
X	1→0	ps
= ps	X→Z,U	ps
≠ ps	X→Z,U	U

Figure 4–12. REG Truth Table

REG RS Primitive

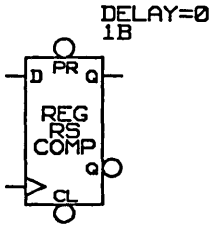


In the REG RS primitive, reset prevails over set if both are asserted. The truth table for the REG RS primitive is shown in Figure 4–13.

Data	Clock	PR*	CL*	Output
X	X	X	Z,U	U
X	X	X	0	0
X	X	Z,U	1	U
X	X	0	1	1
= ps	X→Z,U	1	1	ps
≠ ps	X→Z,U	1	1	U
0	0→1	1	1	0
1	0→1	1	1	1
Z,U	0→1	1	1	U
X	1→0	1	1	ps

Figure 4–13. REG RS Truth Table

REG RS COMP Primitive

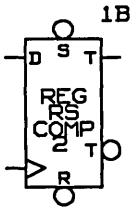


Complementary outputs are provided on the REG RS COMP, and both outputs take the value one when both set and reset are asserted. The truth table for the REG RS COMP primitive is shown in Figure 4–14.

Data	Clock	PR*	CL*	Q	Q*
X	X	X	Z,U	U	U
X	X	Z,U	X	U	U
X	X	0	0	1	1
X	X	0	1	1	0
X	X	1	0	0	1
= ps	X→Z,U	1	1	ps	ps
≠ ps	X→Z,U	1	1	U	U
X	X→0	1	1	ps	ps
0	0→1	1	1	0	1
1	0→1	1	1	1	0
Z,U	0→1	1	1	U	U

Figure 4–14. REG RS COMP Truth Table

REG RS COMP 2 Primitive



The REG RS COMP 2 primitive is a special purpose version of the REG RS COMP primitive. The primitive works the same as the REG RS COMP primitive except that both outputs take the value zero when both S* and R* are asserted, and several additional conditions govern the behavior of preset and clear. The truth table for the REG RS COMP 2 primitive is shown in Figure 4–15.

Data	Clock	S*	R*	T	T*
X	X	X	Z,U	U	U
X	X	Z,U	X	U	U
X	X	0	0	0	0
X	X	0	1	1	0
X	X	1	0	0	1
= ps	X→Z,U	1	1	ps	ps
≠ ps	X→Z,U	1	1	U	U
X	X→0	1	1	ps	ps
0	0→1	1	1	0	1
1	0→1	1	1	1	0
Z,U	0→1	1	1	U	U

Figure 4–15. REG RS COMP 2 Truth Table

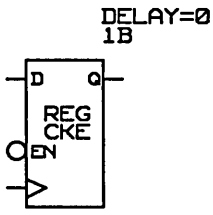
The following additional conditions override the values in the REG RS COMP 2 truth table:

- When an instance of the REG RS COMP 2 primitive has the body property DELAY attached with a value *d*, and when S* and R* both change value from zero to one within *d*

or less of each other, then both T and T* take the value U.

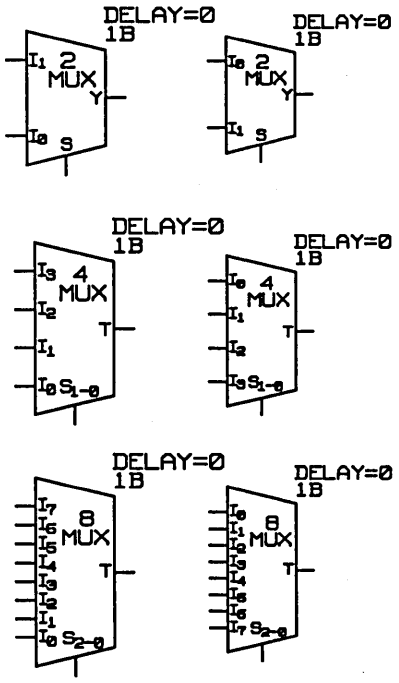
- When S* = U and LAST OUTPUT = 1, then T = 1 and T* = 0.
- When S* = U and LAST OUTPUT ≠ 1, then T = U and T* = U.
- When R* = U and LAST OUTPUT = 0, then T = 0 and T* = 1.
- When R* = U and LAST OUTPUT ≠ 0, then T = U and T* = 0.
- When S* = U and R* = 0, then T = 0 and T* = U.

REG CKE Primitive



The REG CKE primitive is similar to the REG primitive except that it has a clock enable input that enables the clock when asserted.

The Multiplexer Primitives



There are three multiplexer primitives:

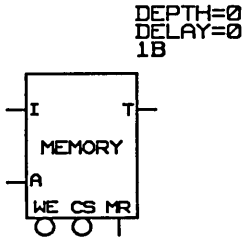
- **2 MUX**
- **4 MUX**
- **8 MUX**

These multiplexer primitives have two, four, and eight inputs, respectively. The SELECT inputs for these parts have a fixed width of one, two, and three bits, respectively. Using a multiplexer can often dramatically reduce the number of simulator primitives needed to model a part. The truth table for the 2 MUX is shown in Figure 4–16. The table can be extended readily for the 4 MUX and 8 MUX.

S	I0 : I1	Y
0	I0 ≠ Z I0 = Z	I0 U
1	I1 ≠ Z I1 = Z	I1 U
Z,U	I0 = I1 ≠ Z I0 = I1 = Z I0 ≠ I1	I1 U U

Figure 4–16. 2 MUX Truth Table

The MEMORY Primitive



The width of each word in the MEMORY primitive is determined by the SIZE property. The number of words is determined by the DEPTH property. The A (address) input has a size corresponding to the number of words. For example, a 256-word RAM has an A input width of eight.

As a convenience to the model builder, the WE (write enable) and CS (chip select) inputs on the MEMORY primitive are bubbled because most actual memory parts have these inputs low asserted. These pins can be unbubbled (using the **bubble** command in GED) if necessary. The MR (master reset) input, when asserted, clears the entire MEMORY to zeros.

Memories can be modeled in either two-state or four-state mode. The default is four-state mode. In four-state mode, each bit of the memory assumes one of three states: zero, one, or U. Use the MEM_STATE directive to select two-state mode. In two-state mode, each bit of the memory assumes one of two states: zero and one.

See the ValidSIM Reference Manual *for information on the* MEM_STATE *directive.*

The truth table for two-state mode is shown in Figure 4-17. The truth table for four-state mode is shown in Figure 4-18. The OUTPUT column shows what value is output in each case. LOC means that the addressed location is output. The WRITE column indicates whether a write operation is performed. *No* indicates that no write operation is performed. A single letter indicates a write operation to the addressed location, and the value written. *All* indicates that the given value is written to all memory locations.

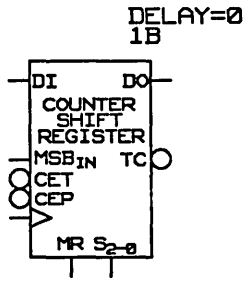
TWO STATE MODE					
MR	CS*	WE*	A	OUTPUT	WRITE
0	1	X	X	Z	no
0	0,U	X	> = DEPTH	U	no
0	0	1	UNDEF	U	no
0	0	1	DEF	LOC	no
0	0	0	DEF	I	I
0	0	U	DEF	1	1
0	0	0,U	UNDEF	1	1 all
0	U	1	X	U	no
0	U	0,U	DEF	U	1
0	U	0,U	UNDEF	U	1 all
1	1	X	X	Z	0 all
1	0	X	X	0	0 all
1	U	X	X	U	0 all
U	1	X	X	Z	1 all
U	0	X	X	U	1 all
U	U	X	X	U	1 all

Figure 4-17. Two-State MEMORY Truth Table

FOUR STATE MODE					
MR	CS*	WE*	A	OUTPUT	WRITE
0	1	X	X	Z	no
0	0,U	X	> = DEPTH	U	no
0	0	1	UNDEF	U	no
0	0	1	DEF	LOC	no
0	0	0	DEF	I	I
0	0	U	DEF	U	U
0	0	0,U	UNDEF	U	U all
0	U	1	X	U	no
0	U	0,U	DEF	U	U
0	U	0,U	UNDEF	U	U all
1	1	X	X	Z	0 all
1	0	X	X	0	0 all
1	U	X	X	U	0 all
U	1	X	X	Z	U all
U	0	X	X	U	U all
U	U	X	X	U	U all

Figure 4-18. Four-State MEMORY Truth Table

The COUNTER SHIFT REGISTER Primitive



This component is very similar to the Fairchild 100K ECL component (F100136).

The COUNTER SHIFT REGISTER primitive operates as either a modulo–16 up/down counter or as a four–bit bidirectional shift register. This primitive has seven inputs:

DI	Parallel data in
MSB IN	Serial data input for shift right. This input produces two outputs: DO (data out) and TC (terminal count; active low).
CET	Count enable trickle input (active low). This input also acts as a serial input for shift left.
CEP	Count enable parallel input (active low)
MR	Master reset
CK	Clock
S	Select inputs (three bits)

The function of the COUNTER SHIFT REGISTER primitive is selected based on the S input as shown in Figure 4–19.

S2	S1	S0	FUNCTION
L	L	L	Parallel Load
L	L	H	Complement
L	H	L	Shift Right
L	H	H	Shift Left
H	L	L	Count Down
H	L	H	Clear
H	H	L	Count Up
H	H	H	Hold

Figure 4–19. COUNTER SHIFT REGISTER Function Table

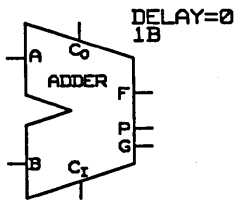
The two count enable inputs are provided for ease of cascading in multistage counters. These two enable inputs must be both asserted for the count up/down operations. One count enable (CET) input also serves as a data input for the shift–left operation. The output also can be cleared asynchronously by bringing the master reset signal active.

The Arithmetic Primitives

There are five arithmetic primitives:

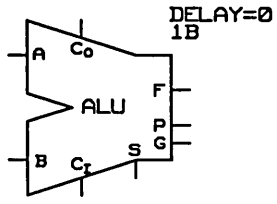
- ADDER
- ALU
- LOOKAHEAD
- CARRY SAVE ADDER
- COMPARATOR

ADDER Primitive



The ADDER primitive takes three inputs: A, B, and CARRY IN; and produces four outputs: F, P, G, and CARRY OUT. The SIZE property determines the width of A, B, and F. F takes the sum of A, B, and CARRY IN. CARRY OUT is asserted if an overflow occurs. G is asserted if the addition of A and B generates a carry. P is asserted if the addition of A, B, and one propagates a carry.

ALU Primitive



The ALU primitive is patterned after the Fairchild 100181 ECL component.

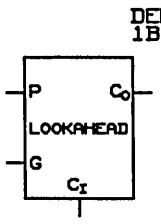
The ALU primitive has inputs and outputs identical to those of the ADDER primitive with the addition of a four-bit select input that selects a function from the table shown in Figure 4-20.

SELECT	FUNCTION
0	A plus B (BCD)
1	A minus B (BCD)
2	B minus A (BCD)
3	0 minus B (BCD)
4	A plus B
5	A minus B
6	B minus A
7	0 minus B
8	(A and B) or (-A and -B)
9	(A and -B) or (-A and B)
10	A or B
11	A
12	-B
13	B
14	A and B
15	0

Figure 4-20. ALU Select Table

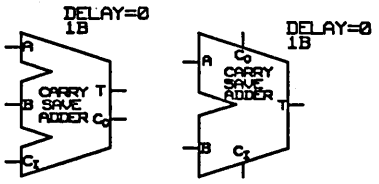
BCD stands for binary-coded-decimal. The behavior of the BCD functions is not defined for SIZE values that are not multiples of four, or for data inputs that are not valid BCD values. *Plus* and *minus* denote two's-complement arithmetic. A '-' denotes one's-complement.

LOOKAHEAD Primitive



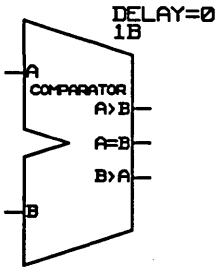
The LOOKAHEAD primitive is a look-ahead-carry generator with three inputs: P, G, and CARRY IN. The primitive produces one output, CARRY OUT. CARRY IN is one bit wide. P, G, and CARRY OUT are sizeable. Each CARRY OUT bit is the carry calculated from CARRY IN and the P and G inputs from the least significant bit through the CARRY OUT bit of the primitive.

CARRY SAVE ADDER Primitive



The CARRY SAVE ADDER takes three inputs: A, B, and CARRY IN, and produces two outputs: T and CARRY OUT. All are sizeable. The two-bit sum is computed for each bit of A, B, and CARRY IN and is stored in the corresponding bits of CARRY OUT and T. T is the low-order bit of the sum, and CARRY OUT is the high-order bit of the sum.

COMPARATOR Primitive



The COMPARATOR primitive takes two inputs, A and B, and produces three one-bit outputs: LT (B>A), EQ (A=B), and GT (A>B). LT is asserted if B > A. EQ is asserted if A = B. GT is asserted if A > B.

The Timing Checker Primitives

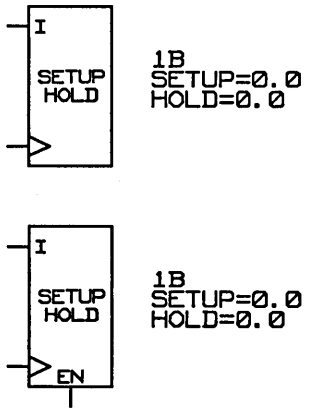
Four timing checker primitives are available as simulation primitives:

- **SETUP HOLD**
- **SETUP RISE HOLD FALL**
- **EDGE TO EDGE**
- **MIN PULSE WIDTH**

Simulation speed is slower when these primitives are used.

The TIMING_CHECK directive in the *simulate.cmd* file enables and disables the timing checker primitives. The TIMING_CHECK ON/OFF command can be used to enable and disable the primitives interactively during the simulation.

SETUP HOLD Primitive



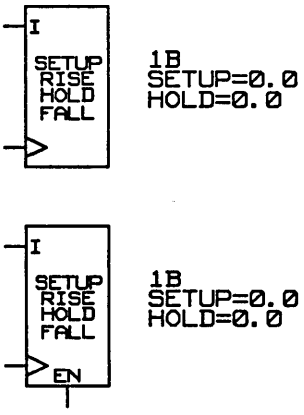
The SETUP HOLD primitive has a clock and data input. For an active-high clock, it generates an error message in the output listing when the data input is not stable from SETUP ns before the rising edge of the clock until HOLD ns after the clock is high. The SETUP HOLD primitive has two default body properties attached:

SETUP = 0.0
HOLD = 0.0

The properties SETUP and HOLD are assigned the required property values by using the GED **change** command. This primitive is used to check the set-up and hold times of registers and latches.

The SETUP HOLD primitive has an optional enable input that turns checking on and off. If the enable input is any value other than ZERO, then checking is enabled.

SETUP RISE HOLD FALL Primitive



The SETUP RISE HOLD FALL primitive has a clock and data input. For an active–high clock, it generates an error message in the output listing if the data input is not stable from SETUP ns:

- Before the rising edge of the clock
- While the clock is high
- Until HOLD ns after the clock has gone low

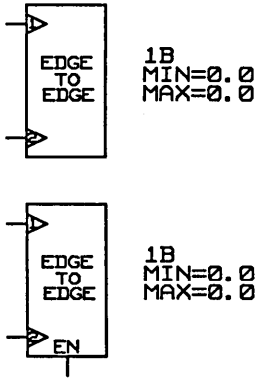
The SETUP RISE HOLD FALL primitive has two default body properties attached:

SETUP = 0.0
HOLD = 0.0

The properties SETUP and HOLD are assigned the required property values by using the GED **change** command. This primitive is used to check the set–up and hold times of data being written into memories.

The primitive has an optional enable input that can be used to turn off checking. If the enable input is used, any value other than zero enables checking. If checking is enabled at any time between the rising edge and the falling edge of the clock, checking is performed for that clock pulse.

EDGE TO EDGE Primitive



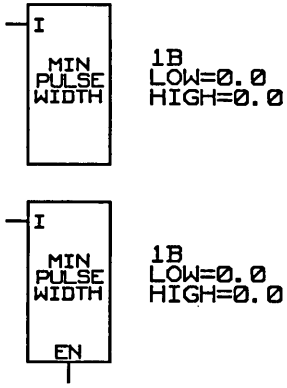
The EDGE TO EDGE primitive has two inputs, CK1 and CK2. It checks that the RISING edge on CK2 is at least a minimum delay from the RISING edge on CK1 and no more than the maximum delay. The EDGE TO EDGE primitive has two default body properties:

MIN = 0.0
MAX = 0.0

The properties MIN and MAX are assigned the required property values by using the GED **change** command. Use only rising delays.

The primitive has an optional enable input that turns the checking on and off. If the enable input is any value other than zero, checking is enabled. If checking is enabled any time during the rising edge of CK1, then checking is performed for that edge. If there is no edge on CK2 (that is, if CK2 does not change state), then no error message is generated.

MIN PULSE WIDTH Primitive



The MIN PULSE WIDTH primitive has one data input. It checks that its data input has no pulses on it that are low for less than LOW ns, and no pulses on it that are high for less than HIGH ns. The MIN PULSE WIDTH primitive has two default body properties:

LOW = 0.0
HIGH = 0.0

The properties LOW and HIGH are assigned the required property values by using the GED **change** command.

The primitive has an optional enable input that turns checking on and off. If the enable input is any value

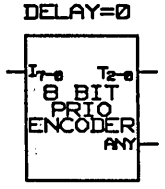
other than zero, checking is enabled. If checking is enabled any time during a given pulse, then the width of that pulse is checked.

The Encoder and Decoder Primitives

There are four encoder/decoder primitives:

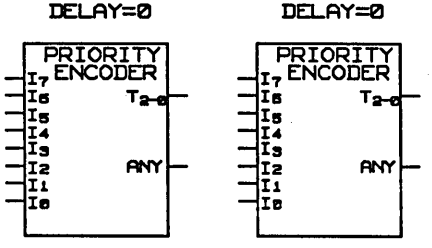
- **8 BIT PRIO ENCODER**
- **PRIORITY ENCODER**
- **1 OF 8 DECODER**
- **8 BIT DECODER**

8 BIT PRIO ENCODER Primitive



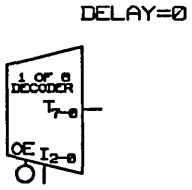
The 8 BIT PRIO ENCODER primitive takes an eight-bit input and produces two outputs: T, which is three bits wide and ANY, which is one bit wide. ANY is asserted if any input bit is asserted. T is the bit number of the most significant bit asserted, if any, where zero is the most significant input.

PRIORITY ENCODER Primitive



The PRIORITY ENCODER primitive takes eight one-bit inputs, I7..I0, and produces two outputs: T, which is three bits wide, and ANY, which is one bit wide. ANY is asserted if any input bit is asserted. T is the bit number of the most significant input which is asserted, if any, where I7 is the most significant input and has a bit number of seven (111 binary).

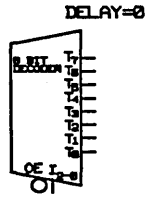
1 OF 8 DECODER Primitive

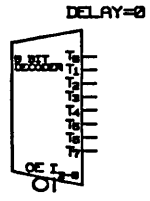


The 1 OF 8 DECODER primitive takes two inputs: SELECT (I<2..0>), which is three bits wide, and OUTPUT ENABLE, which is one bit wide. It produces an eight-bit output T<7..0>. If OUTPUT ENABLE is asserted, SELECT selects which bit of T is asserted. When SELECT contains Z and/or U, those SELECT bits are treated as "don't care" for selecting output bits and the selected output bits are set to U.

I<2..0>	OE	T<7..0>
X	1	all bits 0
i = defined value	Z,U	the i-th bit U and the rest 0
(0/1,Z/U,0/1) etc.	Z,U	the (0/1,*,0/1)-th bits U and the rest 0
i = defined value	0	the i-th bit 1 and the rest 0
(0/1,Z/U,0/1) etc.	0	the (0/1,*,0/1)-th bits U and the rest 0

8 BIT DECODER Primitive





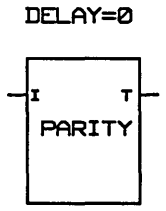
The 8 BIT DECODER primitive is identical in operation to the 1 OF 8 DECODER primitive except for the output of the primitives. The output of the 1 OF 8 DECODER is an eight-bit bus. The output of the 8 BIT DECODER is eight individual bits.

Other Primitives

There are four other Simulator primitives:

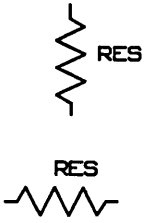
- **PARITY**
- **RES**
- **PASS TRANSISTOR**
- **UNI PASS TRANSISTOR**

PARITY Primitive



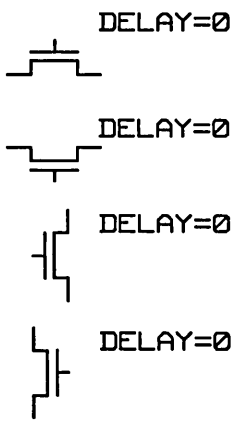
The PARITY primitive's I input can be sized and produces a one-bit output T. T is one if the total of the asserted (one) inputs is odd.

RES Primitive



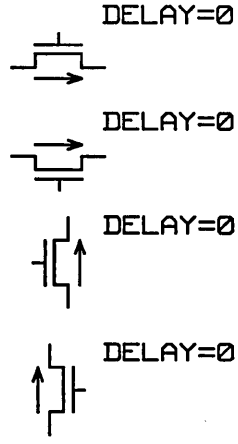
The resistor primitive RES is fully bidirectional and acts like a wire except that HARD strength signals are converted to SOFT strength when they pass through. RES primitives always have zero delay. The RES primitive is sizeable and the pins cannot be bubbled.

PASS TRANSISTOR Primitive



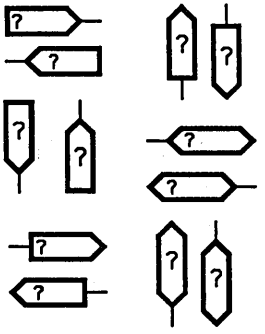
The PASS TRANSISTOR primitive is fully bidirectional and acts like a switch. The G pin of the PASS TRANSISTOR controls whether the A and B pins are connected together. An active G pin (zero if the pin is bubbled, otherwise one) causes the PASS TRANSISTOR to act like a wire, connecting the A and B nets. An inactive G pin causes the PASS TRANSISTOR to act as if it were not in the circuit. The delay from A to B or B to A is always zero. The G pin has an input delay that assumes the value of the DELAY property on the PASS TRANSISTOR. The A and B pins of the PASS TRANSISTOR are sizeable and are not bubbled. The G pin is always one bit wide and can be bubbled.

UNI PASS TRANSISTOR Primitive



The UNI PASS TRANSISTOR is a unidirectional version of the PASS TRANSISTOR and results in more rapid simulation for MOS circuits. Pins and properties of the UNI PASS TRANSISTOR primitive are identical – a G pin that controls whether the A and B pins are connected. However, because this transistor is unidirectional, the A pin is an input pin rather than an output.

The FLAG Primitive



There are twelve different versions of the FLAG primitive. The first four versions are primary input flags. The second four versions are primary output flags. The last four versions are bidirectional flags.

The FLAG primitives are used for formatting the Primary I/O Trace program information. To use the trace program, each primary I/O signal must be designated on the top level schematic (the root drawing) with a FLAG body.

The question mark identifies the PATH property attached to the primitive. The PATH gets written automatically; you do not have to define the property.

User-coded Primitives

The Simulator allows you to code simulator models in Pascal or C and refer to them using standard SCALD drawings. Existence of these user-coded primitives (UCPs) means that you can expand the "parts set" understood by the Logic Simulator. For more information on user-coded primitives, see the *ValidSIM Reference Manual.*

Simulation Properties

When creating simulation models, you can size simulation bodies and add delay values to each primitive using body and pin properties. Simulation body properties are:

- **DELAY**
- **RISE**
- **FALL**
- **SIZE**

Pin-to-pin delays are specified using the pin properties:

- **PDELAY**
- **PRISE**
- **PFALL**

Simulation models can carry body properties and pin properties simultaneously. When the delays through a primitive are all one set of values except for a single delay path, the one path can be specified using the pin properties, and the other delays can be specified using the body properties. For designs where delays are related to changes in output loading, temperature, and voltage, the Delay Estimator and Expression Evaluator can be used.

Table 4-3 summarizes the most common simulation properties. For detailed information on simulation properties and directives, refer to the *ValidSIM Reference Manual.*

Table 4-3. Simulation Properties

Prop Type	Prop Name	Controlling Directive(s)	Comments
Body			Body properties attach to the origin of a primitive to control the size or delay time of the primitive.
	DELAY	**DELAY_MODE RISE_FALL**	Specifies the delay time as a single value or the rise/fall delay values as [*min,typ,max*]. DELAY_MODE directive selects which value to use for current simulation run; *max* is the default. DELAY property is the default delay value used if RISE_FALL directive is OFF.
	RISE	**RISE_FALL**	Can be attached in addition to DELAY property. When directive is ON (default), RISE delay overrides DELAY property.
	FALL	**RISE_FALL**	Can be attached in addition to DELAY property. When directive is ON (default), FALL delay overrides DELAY property.
	SIZE		Specifies the number of bits on a pin. Can also use SIZE=SIZE to match primitive size to the SIZE property attached to the body drawing.
Pin		**PIN_DELAY**	Pin delay properties are allowed on both input and output pins. The PIN_DELAY directive affects all of the pin delay properties; when the directive is ON, pin delay properties override body delay properties. If PDELAY is not defined and the RISE_FALL directive is OFF, the maximum value between PRISE and PFALL is used.
	PDELAY	**RISE_FALL**	PDELAY overrides PRISE and PFALL when the RISE_FALL directive is OFF.
	PRISE	**RISE_FALL**	PRISE and PFALL override PDELAY when the RISE_FALL directive is ON.
	PFALL	**RISE_FALL**	PRISE and PFALL override PDELAY when the RISE_FALL directive is ON.

Simulation Body Properties

Simulation body properties are attached to the origin of the primitive and control the behavior of the entire primitive.

The DELAY Property

Delays are given in nanoseconds. Primitives without an explicit DELAY are assumed to have a delay of zero. By convention, primitives are given delays to model the worst-case behavior of the part being modeled, but this is not required. The syntax of the DELAY property in simulation models is:

SYNTAX

DELAY=[*min,typ,max*],[*min,typ,max*]

Rise delay (the first set of values) and *fall delay* (the second set of values) define the rise and fall times of the output of the signal. The *min, typ, max* delays must be enclosed in square brackets and separated by commas. If the rise and fall delay values are the same, only one [*min, typ, max*] entry is required.

The Simulator's DELAY_MODE directive selects which of the three delay values is used for the current simulation run (*min, typ,* or *max;* the default is *max*).

For the Simulator to function correctly, the simulation model must represent one of the possible timing behaviors for the component. Exercise care when specifying delay values; in particular, zero-delay components can result in unexpected behavior in a circuit.

The RISE Property

In addition to using the DELAY property, rise delays can be specified using the RISE body property with a rise delay time value. If the DELAY and RISE properties are both attached to a body, the Simulator's RISE_FALL directive selects which delay value is used.

The FALL Property

In addition to using the DELAY property, fall delays can be specified using the FALL body property with a fall delay time value. If the DELAY and FALL properties are both attached to a body, the Simulator's RISE_FALL directive selects which delay value is used.

The SIZE Property

Most simulator primitives can have a SIZE property to specify the number of bits on a pin. For example, to compute the sum of two 16-bit signals, a single adder primitive with a SIZE of 16 can be used instead of 16 adder primitives. A primitive can be given the property SIZE=SIZE, which means that the size of the primitive is taken from the SIZE property attached to the part being modeled. Two special primitives, the 8 BIT PRIO ENCODER and the 1 OF 8 DECODER have a fixed SIZE of eight bits.

Many primitives have inputs and outputs that are not affected by the SIZE property. All enable inputs, clock inputs, and chip select inputs have a fixed width of one bit. The select input of an eight-bit multiplexer is always three bits wide.

Simulation Pin Properties

For complex primitives with multiple input and output pins, accurate modeling of the delays within the primitive is tedious if not impossible. The pin-to-pin delay feature allows you to associate separate delay values for individual paths from input pin to output pin.

Pin delay properties are allowed on both input and output pins. Any conflict between input pin properties and output pin properties (two different delay values specified for one pin-to-pin path) is reported as an error. Delay values specified between two input pins or two output pinss are also reported as an error.

To override the specified body DELAY properties on a model and use the pin delay properties, add the directive PIN_DELAY ON to the *simulate.cmd* file (the directive is OFF by default). This directive controls whether the model uses the DELAY, RISE, and FALL properties or PDELAY (pin delay), PRISE (pin rise delay), and PFALL (pin fall delay). When the directive is ON, pairs of input and output pins use the body DELAY properties if pin delays are not specified.

Modifying Simulation Models

Simulation models are designed to correctly model the part while including no extraneous data. The smaller the model, the faster the Simulator can run. Models are therefore difficult to intuitively grasp; make as few changes to models as absolutely necessary. The most frequent change to a simulation model is the change of the value of a delay property. Use the **change** command in GED to change the value. Be careful to change the correct instance of the delay property in the model. To accommodate different delay paths through the device, propagation delays are usually divided up and placed in different locations in the model. Examine the model carefully to choose which delays you need to change.

The Timing Model

This section describes:

- Defining the timing model
- Creating the model (checklist)
- Timing primitives
- Timing properties
- Modifying timing models

Defining the Timing Model

Timing models reflect the timing behavior of a design. Models are built as simply as possible so that the Timing Verifier runs quickly and efficiently. Timing models therefore focus on timing characteristics and do not exhaustively simulate the logical behavior of the component.

Timing models are built from a specific set of parts called *timing primitives*. You decide which timing primitives to use by studying the functional specification and data tables in the appropriate data book. It is possible to create different timing models (using different timing primitives) for the same component, and obtain the same timing results.

Timing models need to correctly model the delays of all signals through the component (propagation delay). For clocked and complex components, the model must check:

- Setup and hold times
- Pulse width
- Edge to edge time (when appropriate)

When designing timing models, you should keep certain goals in mind. These goals are the same as those for designing simulation models and are described under "General Design Rules for Models" in Section 4, *The Simulation Model*. The standards used in calculating delay and pulse width information for the timing models are also the same as those described under "Delay and Pulse Width Standards" in Section 4.

Creating the Model: Checklist

Creating a timing drawing requires the same basic steps as creating a logic drawing except that the parts used are timing primitives from the Time library or components from the Standard library.

1. Access the Time library:

   ```
   library time
   ```

2. Create a drawing for your component with a .TIME extension (for example, 293.TIME).

3. Add the required timing primitives and a DRAWING body (with TITLE and ABBREV properties).

4. Add a PIN NAMES component. This component accesses the .BODY drawing, collects all the pin names assigned to the body, and lists them beneath the PIN NAMES header. All pins identified on the logic drawing *must* be accounted for on the timing model.

5. Wire the model.

6. Name the input and output signals. Signal names must include the \I interface signal property and correspond to the names of the signals in the body drawing for the part being modeled.

7. Assign the required timing properties.

8. Verify attachments, check, and write the model.

- Use a B SIZE PAGE as a border.
- Center the drawing on the page.
- Include the name of the drawing and the initials of the creator in the boxes in the lower right hand corner of the page border.
- Enter the page number of the drawing as a note (text size 1.5) in the form "1 of 1."
- Include a note block (notes enclosed with wires to form a block) to document any assumptions and/or critical design decisions that are not obvious to the user.
- Add primitives only from the Standard and Time libraries.
- Every model must have a DRAWING body (with TITLE and ABBREV properties attached).
- Every model should have a PIN NAMES body.
- Follow SCALD signal syntax for signal names.
- Do not use bit lists in bit subscripts.
- Make sure all interface signals have the \I property in the signal name.
- All interface signals should have an explicit width specified unless the signal is a scalar.
- All properties attached to bodies should be placed above the body or to the right. Place the properties one above the other and left-aligned. Display both the property value and name for all properties except PATH.

Since it is possible to create different timing models for the same component, there is no step-by-step description for creating a timing model for the TTL 293 component. Figure 5-1 shows one possible timing model for the TTL 293. Timing primitives and timing properties are discussed in general following the drawing.

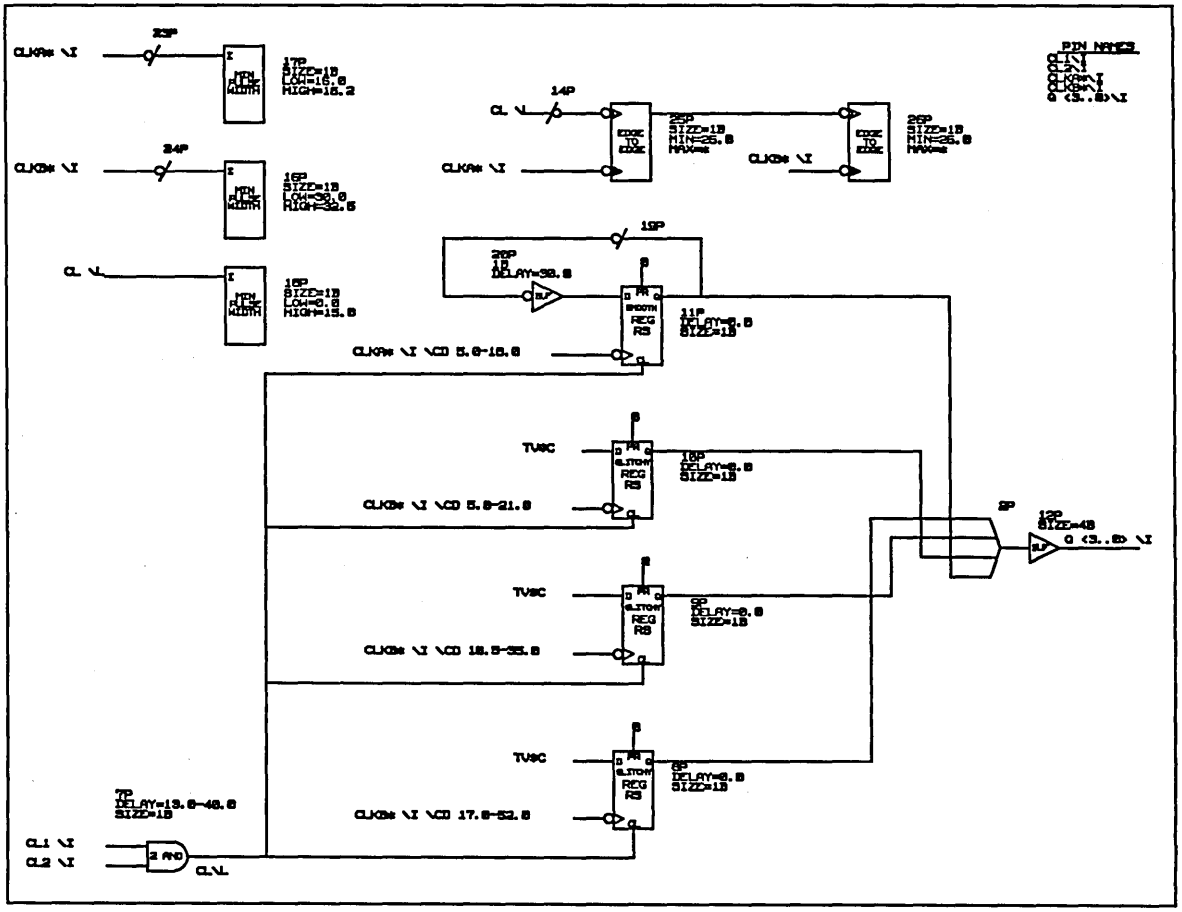


Figure 5-1. TTL 293 Timing Model

The Timing Primitives

For additional information on timing primitives and properties, see the ValidTIME Reference Manual.

The timing primitives are stored in the Time library. There are three groups of timing primitives:

- Standard function primitives (based on functions like gates and flip-flops)
- Non-standard functions useful for timing models
- Error-checking primitives added to models to detect timing errors

Sometimes there are components and primitives that have similar names, for example, the "2AND" component and the "2 AND" primitive. Be sure to leave a space in the primitive names.

Bubbled Pins

Each input and output pin on a primitive can be individually bubbled using the GED **bubble** command. Bubbling a primitive pin inverts the logical function of the primitive. This allows you to create inverting buffers, NAND gates, NOR gates, negative-edge-triggered registers, and so on. For example, bubbling the output pin of the timing primitive BUF adds an inverter to your model.

Truth Table Abbreviations

Table 5–1 shows the abbreviations used in the truth tables for the timing primitives.

Table 5–1. Truth Table Abbreviations

Abbreviation	Meaning
C	Constant
F	Fall
ps	Previous state
R	Rise
S	Stable
U	Unknown value
X	Can be any value
Z	High impedance
→	Transition
≠	Not equal to

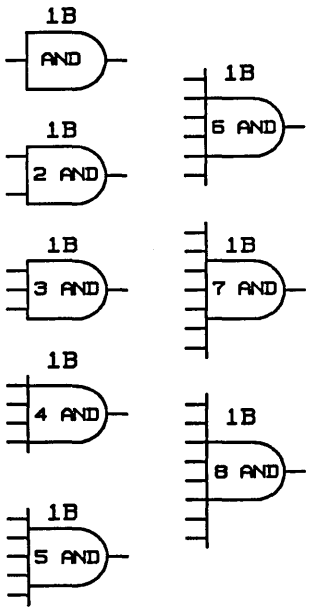
In cases where more than one entry applies to a given set of input conditions, the first entry takes precedence.

Standard Function Primitives

These time primitives are based on familiar SSI and MSI components. They perform some (not all) of the functions of these parts. There are 11 standard function primitives:

- AND
- OR
- XOR
- LATCH
- LATCH RS
- TS BUF
- REG
- REG RS
- 2 MUX
- 4 MUX
- 8 MUX

AND Primitive



There are eight AND primitives:

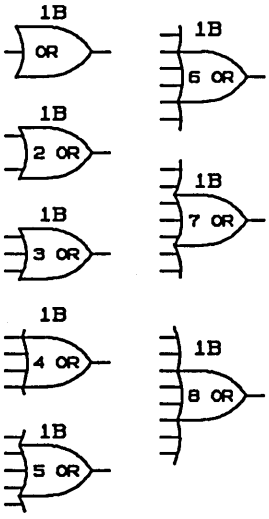
- AND
- 2 AND
- 3 AND
- 4 AND
- 5 AND
- 6 AND
- 7 AND
- 8 AND

The truth table for an AND primitive is shown in Figure 5-2.

AND	0	1	S	R	F	C	U	Z
0	0	0	0	0	0	0	0	0
1	0	1	S	R	F	C	U	U
S	0	S	S	R	F	C	U	U
R	0	R	R	R	C	C	U	U
F	0	F	F	C	F	C	U	U
C	0	C	C	C	C	C	U	U
U	0	U	U	U	U	U	U	U
Z	0	U	U	U	U	U	U	U

Figure 5-2. AND Primitive Truth Table

OR Primitive



There are eight OR primitives:

- OR
- 2 OR
- 3 OR
- 4 OR
- 5 OR
- 6 OR
- 7 OR
- 8 OR

The truth table for an OR primitive is shown in Figure 5–3.

OR	0	1	S	R	F	C	U	Z
0	0	1	S	R	F	C	U	U
1	1	1	1	1	1	1	1	1
S	S	1	S	R	F	C	U	U
R	R	1	R	R	C	C	U	U
F	F	1	F	C	F	C	U	U
C	C	1	C	C	C	C	U	U
U	U	1	U	U	U	U	U	U
Z	U	1	U	U	U	U	U	U

Figure 5–3. OR Primitive Truth Table

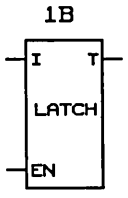
XOR Primitive

The XOR has only a two-input version. The truth table for an XOR primitive is shown in Figure 5–3.

XOR	0	1	S	R	F	C	U	Z
0	0	1	S	R	F	C	U	U
1	1	0	S	F	R	C	U	U
S	S	S	S	C	C	C	U	U
R	R	F	C	C	C	C	U	U
F	F	R	C	C	C	C	U	U
C	C	C	C	C	C	C	U	U
U	U	U	U	U	U	U	U	U
Z	U	U	U	U	U	U	U	U

Figure 5–4. XOR Primitive Truth Table

LATCH Primitive



The LATCH primitive has a data input and an enable input. The primitive is affected by the value of the TRANSITION property and the value of the Timing Verifier's LATCH_ERR_MODEL directive. The truth tables for the LATCH primitive are shown in Figure 5-5 and Figure 5-6. When enable is bubbled, the inverse of the truth table applies.

LATCH			
EN	**Last Output**	**Data**	**Output**
0	0,1,S	X	= 0,1,S
0	R,F,C,U,Z	X	S
1	X	0,1,S,R,F,C	= DATA
1	X	U,Z	U
R	= DATA	0,1,U,Z	0,1,U,U
R	S	S	S *
R	= DATA	All other conditions	C
R	≠ DATA	U,Z	U
R	0	1,S	R
R	1	0,S	F
R	R,F,C,U,Z	0,1,S	C
R	R,1	R	R
R	F,0	F	F
R	All other conditions	All other conditions	C
F	= DATA	0,1,S,U,Z	0,1,S,U,U
F	= DATA	R,F,C	C
F	X	U,Z	U,U
F	0	1,S	R

Figure 5-5. LATCH Truth Table (Part 1)

* *If there has been no data transition since EN was last 1 or R and the latch is being simulated SMOOTH.*

LATCH			
EN	**Last Output**	**Data**	**Output**
F	1	0,S	F
F	C	0,1,S	= DATA
F	R,1	R	R
F	F,0	R	F
F	All other conditions	All other conditions	C
S	= DATA	X	= DATA
S	≠ DATA	0,1,S	S
S	1	R	R
S	0	F	F
S	≠ 1	R	C
S	≠ 0	F	C
S	X	C	C
S	All other conditions	All other conditions	U
C	X	U,Z	U
C	= DATA	0,1,S,R,F,C	= DATA
C	All other conditions	All other conditions	C
Z	X	X	U
U	X	X	U

Figure 5-6. LATCH Truth Table (Part 2)

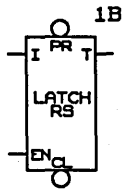
If the DATA undergoes a transition while the latch is closing (the ENABLE signal has the value F), then a setup/hold time violation has occurred. Under these conditions, the signal values are calculated as OPEN, CLOSED, or CONSERVATIVE, depending on the value of the LATCH_ERR_MODEL directive. The default value is CONSERVATIVE. Figure 5-7 shows the truth tables for each of the three values of the LATCH_ERR_MODEL directive.

LATCH_ERR_MODEL = OPEN			
Last EN	**Last Output**	**Data**	**Output**
F	X	U,Z	U
F	0	0,1,S	R
F	1	0,1,S	F
F	C	0,1,S	0,1,S
F	S,R,F,U,Z	0,1,S	C
F	R,1	R	R
F	F,0	F	F
F	All other conditions	All other conditions	C
LATCH_ERR_MODEL = CLOSED			
Last EN	**Last Output**	**Data**	**Output**
F	0,1,S	X	0,1,S
F	R,F,C,U,Z	X	S
LATCH_ERR_MODEL = CONSERVATIVE			
Last EN	**Last Output**	**Data**	**Output**
F	X	U,Z	U
F	0	0,1,S	R
F	1	0,1,S	F
F	S,R,F,C,U,Z	0,1,S	C
F	R,1	R	R
F	F,0	F	F
F	All other conditions	All other conditions	C

Figure 5-7. LATCH_ERR_MODEL Truth Table

The LATCH primitive has the default property TRANSITION=GLITCHY. When the LATCH is clocked, the output of the LATCH always changes, even when the input remains stable. If the property TRANSITION=SMOOTH is attached to the LATCH, the output of the LATCH does not change when the LATCH is clocked and the input remains stable.

LATCH RS Primitive



The CHG function is discussed on page 5–23.

The LATCH RS primitive is a LATCH primitive that has asynchronous set and reset inputs. First the LATCH output is computed for the current input values, then the SET/RESET function is applied to the outputs. The SET/RESET function inherits the state of the TRANSITION property (SMOOTH/GLITCHY) and functions differently depending on the value. Figure 5–8 shows the truth table for the SET/RESET function in GLITCHY mode. Figure 5–9 shows the truth table for the SET/RESET function in SMOOTH mode.

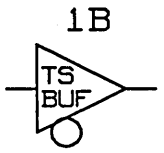
SET/RESET in GLITCHY MODE			
RESET	**SET**	**OUTPUT**	**NEW OUTPUT**
0	0	X	OLD OUTPUT
0	X	1	1
0	1	≠ 1	1
0	R,F,C	≠ 1	C
0	S	0,S	S
0	S	R,F,C	C
0	S	U,Z	U
0	U,Z	≠ 1	U
X	0	0	0
1	0	X	0
R,F,C	0	≠ SET	C
U,Z	0	≠ SET	U
S	0	1,S	S
S	0	R,F,C	C
S	0	U,Z	U
All	other	conditions	CHG (OUTPUT, RESET, SET)

Figure 5–8. SET/RESET in GLITCHY Mode Truth Table

SET/RESET in SMOOTH MODE			
RESET	SET	OUTPUT	NEW OUTPUT
0	0	X	OUTPUT
0	1	1	1
0	X	X	1
0	R	0	R
0	S	0,S	S
0	S	R,F,C	C
0	S	U,Z	U
0	F,C	X	C
0	U,Z	X	U
X	0	0	0
1	0	X	0
R	0	0	F
S	0	0,1,S	S
S	0	R,F,C	C
S	0	U,Z	U
F,C	0	X	C
U,Z	0	X	U
1,R	F	0,1,S	0,F,F
1,R	F	R,F,C	C
1,R	F	U,Z	U
F	1,R	0,1,S	R,1,R
F	1,R	R,F,C	C
F	1,R	U,Z	U

Figure 5-9. SET/RESET in SMOOTH Mode Truth Table

TS BUF Primitive



The tri-state buffer primitive has two inputs, data and enable. The data input and output signals are affected when you attach the SIZE property to the TS BUF primitive; the enable signal is common to all buffers.

The default operating mode for the TS BUF primitive is known as *tri-state mode*. When the enable is STABLE, the output is unknown. This is a conservative model of tri-state behavior. The alternate operating mode is called *wire-or mode*. This mode is less conservative and accommodates designs in which the enable signal is specified as STABLE/CHANGING.

See the ValidTIME Reference Manual *for more information on Timing Verifier directives.*

Mode selection for the TS BUF primitive is controlled by the Timing Verifier's TS_BUF_TYPE directive. Figure 5-10 shows the truth tables for tri-state mode and wire-or mode for the TS BUF primitive.

Enable Input

TRI-STATE MODE

Data Input: TS BUF	0	1	S	R	F	C	U	Z
0	Z	0	U	C	C	C	U	U
1	Z	1	U	C	C	C	U	U
S	Z	S	U	C	C	C	U	U
R	Z	R	U	C	C	C	U	U
F	Z	F	U	C	C	C	U	U
C	Z	C	U	C	C	C	U	U
U	Z	U	U	U	U	U	U	U
Z	Z	U	U	U	U	U	U	U

WIRE-OR MODE

Data Input: TS BUF	0	1	S	R	F	C	U	Z
0	Z	0	0	C	C	C	U	U
1	Z	1	1	C	C	C	U	U
S	Z	S	S	C	C	C	U	U
R	Z	R	R	C	C	C	U	U
F	Z	F	F	C	C	C	U	U
C	Z	C	C	C	C	C	U	U
U	Z	U	U	U	U	U	U	U
Z	Z	U	U	U	U	U	U	U

Enable Input

Figure 5-10. TS BUF Truth Tables

If you tie together the outputs of two or more TS BUF primitives, you create a *tri-state bus*, or TS BUS. The TS BUS is a special type of primitive because it is not represented by a GED drawing. You cannot add a TS BUS to a timing model. A TS BUS is shown in Figure 5-11.

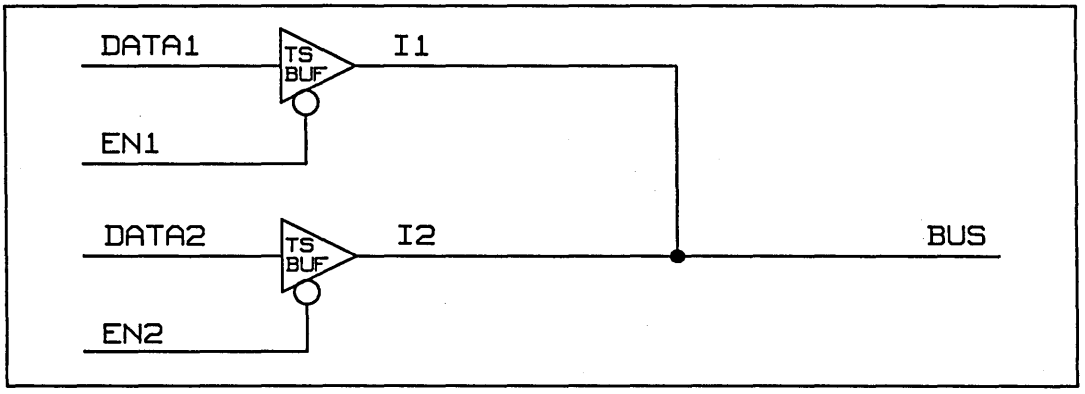


Figure 5-11. TS BUS Primitive

Because the drivers are tri-state and share the use of the BUS by means of separate ENABLE signals, the logical function represented is not the same as that of a wire-gate. With a tri-state bus, the only two meaningful configurations are:

- Only one TS BUF is enabled at a time
- If two are enabled, they carry identical output signal values

Almost all other conditions produce the signal value U (unknown) on the bus.

The Timing Verifier evaluates this circuitry in accordance with the tables shown in Figure 5-12. The TS BUS, like the TS BUF primitive, operates in tri-state mode by default but can also operate in wire-or mode. Mode selection for the TS BUS is controlled by the TS_BUF_TYPE directive.

I1

TRI-STATE MODE

I2 \ TS BUS	0	1	S	R	F	C	U	Z
0	0	U	U	U	F	U	U	0
1	U	1	U	R	U	U	U	1
S	U	U	U	U	U	U	U	S
R	U	R	U	R	U	U	U	R
F	F	U	U	U	F	U	U	F
C	U	U	U	U	U	U	U	C
U	U	U	U	U	U	U	U	U
Z	0	1	S	R	F	C	U	Z

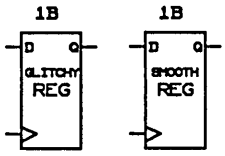
WIRE-OR MODE

I2 \ TS BUS	0	1	S	R	F	C	U	Z
0	0	S	S	R	F	C	U	0
1	S	1	S	R	F	C	U	1
S	S	S	S	C	C	C	U	S
R	R	R	C	R	C	C	U	R
F	F	F	C	C	F	C	U	F
C	C	C	C	C	C	C	U	C
U	U	U	U	U	U	U	U	U
Z	0	1	S	R	F	C	U	Z

I1

Figure 5–12. TS BUS Tables

REG Primitive



The REG primitive implements a rising edge triggered register. The truth table for the REG primitive differs depending on the value of the clock signal. The tables are shown in Figure 5–13.

REG when CLOCK = 1			
Last Clock	**Input**	**Last Output**	**Next Output**
0	0,1	0,1	LAST OUT
0	1,R	0,R	R
0	0,F	1,F	F
0	S	S	S*
1	X	≠ 0,1,S	S
1	X	0,1,S	LAST OUT
S	X	X	LAST OUT
R	0,1	S	LAST OUT
F	X	≠ 0,1,S	S
C,U,Z	INPUT =	LAST OUT	LAST OUT
C,U,Z	INPUT ≠	LAST OUT	S

REG when CLOCK = C or R			
Last Clock	**Input**	**Last Output**	**Next Output**
X	0,1	0,1	LAST OUT
X	1,R	0,R	R
X	0,F	1,F	F
X	S	S	S*

REG when CLOCK = 0, S or F			
Last Clock	**Input**	**Last Output**	**Next Output**
X	X	≠ 0,1,S	S
X	X	0,1,S	LAST OUT

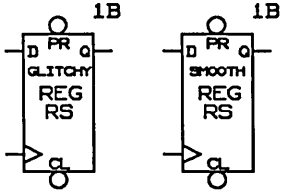
REG when CLOCK = U or Z			
Last Clock	**Input**	**Last Output**	**Next Output**
X	X	X	U

Figure 5–13. REG Truth Tables

* *If the* REG *is smooth and there were no input transitions.*

When the REG primitive has the property TRANSITION=GLITCHY and the REG is clocked, the output of the REG always changes, even when the input remains stable. When the REG primitive has the property TRANSITION=SMOOTH, the output of the REG does not change when the REG is clocked and the input remains stable.

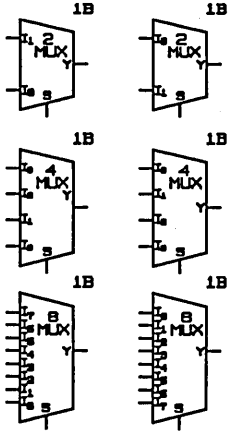
REG RS Primitive



The REG RS primitive is the same as the REG primitive except that it also has asynchronous reset and set inputs. First, the REG output is computed for the current input values, then the SET RESET function is applied to the output.

The SET/RESET function inherits the state of the TRANSITION property (SMOOTH/GLITCHY) and functions differently depending on the value. The REG RS primitive uses the same SET/RESET truth tables as the LATCH RS primitive. The truth tables for the SET/RESET function are shown in Figure 5–8 (page 5–14) and Figure 5–9 (page 5–15).

2 MUX, 4 MUX, 8 MUX Primitives



The 2 MUX, 4 MUX, and 8 MUX primitives implement two–input, four–input, and eight–input multiplexers, respectively. If any of the select inputs on these multiplexers has a known value of zero or one, then only the possibly selected state inputs are considered when calculating the output value. If more than one data input might be selected, the output value is calculated by using the CHG function on the set of selected data inputs.

If the MUX has no TRANSITION property or if TRANSITION=GLITCHY, then any input transition

causes an output transition of the appropriate slope. If TRANSITION=SMOOTH, then if the output state before and after an input transition is the same, there is no output transition.

Non-Standard Function Primitives

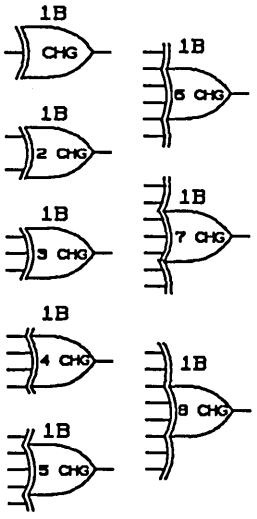
The non-standard time primitives are particularly suited to modeling timing functionality. Some of these primitives (such as BUF and RES) are familiar components, but may be used somewhat differently in a timing model. Others (such as CHG) were created especially for the Timing Verifier. These components:

- Attach delay properties to various parts of a model
- Provide accurate load calculations
- Assure efficient and correct functioning of the model

There are seven non-standard function primitives:

- **CHG**
- **BUF**
- **IDENTITY**
- **RES**
- **THRESHOLD**
- **TRANSMISSION GATE**
- **UNI TRANS GATE**

CHG Primitive



The most important non-standard primitive is the CHG (change) primitive. This primitive tells you whether an input signal is stable, changing, or unknown; frequently, this is all the information the Timing Verifier needs. When you add delay to the CHG primitive, you effectively model simple propagation delay through a component.

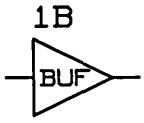
For example, to model the propagation delay from the A and B inputs to the sum (Y output) of an adder, use the CHG primitive and attach the appropriate DELAY property. This model is very simple because the delay through this component is the same regardless of the values being added. Adding in the appropriate delay for CARRY IN complicates the model only slightly.

The truth table for a CHG primitive is shown in Figure 5-14.

CHG	0	1	S	R	F	C	U	Z
0	S	S	S	C	C	C	U	U
1	S	S	S	C	C	C	U	U
S	S	S	S	C	C	C	U	U
R	C	C	C	C	C	C	U	U
F	C	C	C	C	C	C	U	U
C	C	C	C	C	C	C	U	U
U	U	U	U	U	U	U	U	U
Z	U	U	U	U	U	U	U	U

Figure 5-14. CHG Primitive Truth Table

BUF Primitive



The BUF primitive is used as a convenient place to attach delay properties in a model. It is also used to isolate outputs so that correct load calculations can be performed. The value of a signal is not changed by the BUF primitive (except the value Z). Buffers are also used to isolate outputs for correct load checking.

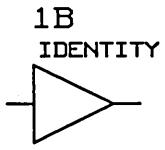
The truth table for the BUF primitive is shown in Figure 5–15.

Input	Output
0	0
1	1
S	S
R	R
F	F
C	C
U	U
Z	U

Figure 5–15. BUF Truth Table

To create an inverting buffer, simply bubble the input or output pin. Non-inverting buffers are commonly used for delays.

IDENTITY Primitive



The IDENTITY primitive is a special case of the BUF primitive. It retains the identity of all signal values including Z. It also retains the signal strength of all input signals.

The truth table for the IDENTITY primitive is shown in Figure 5–16.

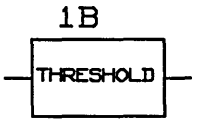
Input	Output
0	0
1	1
S	S
R	R
F	F
C	C
U	U
Z	Z

Figure 5–16. IDENTITY Truth Table

RES Primitive

The resistor primitive RES has the same truth table as the IDENTITY primitive. However, the RES primitive converts the strength of HARD input signals to SOFT signal strength. Since most other signal strengths in a design are HARD, this means that the value of the RES output can be overridden by a competing HARD value. This primitive is used to assure the correct modeling of circuits using pull-up resistors. For HARD and SOFT input strengths, the RES outputs a SOFT signal strength; for UNDRIVEN input strengths, the RES outputs an UNDRIVEN signal strength.

THRESHOLD Primitive



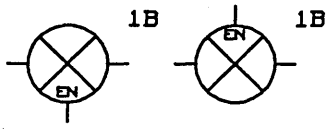
The THRESHOLD primitive has a threshold input and a single output pin. The primitive behaves somewhat like an input-state (0 or 1) detector. Its output remains changing until its threshold input is asserted. This primitive is seldom used.

The truth table for the THRESHOLD primitive is shown in Figure 5-17.

Input	Output
0	C
1	1
S	C
R	C
F	C
C	C
U	U
Z	U

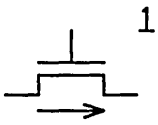
Figure 5-17. THRESHOLD Truth Table

TRANSMISSION GATE Primitive



The TRANSMISSION GATE primitive has an enable input (EN) and two bidirectional pins (T1 and T2). If the enable input is zero, then both T1 and T2 are set to high impedance (Z). If the enable input is one, then T1 and T2 are tied together using the same function as the TS BUS (see page 5-17).

UNI TRANS GATE Primitive



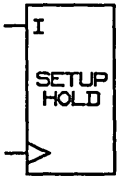
The UNI TRANS GATE primitive is a uni-directional transistor. It has a gate input, an input pin, and an output pin. (The arrow points in the direction of the output.) If the gate input is zero, then output is set to high-impedance. If the gate input is set to one, then the value and strength of the input pin is passed to the output pin.

Error-Checking Primitives

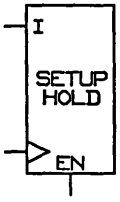
The error-checking primitives do not model functionality. They are added to timing models of clocked components to check for setup and hold time violations and other clock-related errors. There are four timing checker primitives:

- **SETUP HOLD**
- **SETUP RISE HOLD FALL**
- **MIN PULSE WIDTH**
- **EDGE TO EDGE**

SETUP HOLD Primitive



1B
SETUP=0.0
HOLD=0.0



1B
SETUP=0.0
HOLD=0.0

The SETUP HOLD primitive has a clock and data input. For an active-high clock, it generates an error message in the output listing when the data input is not stable from SETUP ns before the rising edge of the clock until HOLD ns after the clock is high. The SETUP HOLD primitive has two default body properties attached:

SETUP = 0.0
HOLD = 0.0

The properties SETUP and HOLD are assigned the required property values by using the GED **change** command. This primitive is used to check the set-up and hold times of registers and latches.

The SETUP HOLD primitive has an optional enable input that turns the checking on and off. If the enable input is any value other than zero, then checking is enabled. If checking is enabled any time during the rising edge of the clock input, then checking is performed for that edge.

SETUP RISE HOLD FALL Primitive

1B
SETUP=0.0
HOLD=0.0

1B
SETUP=0.0
HOLD=0.0

The SETUP RISE HOLD FALL primitive has a clock and data input. For an active-high clock, it generates an error message in the output listing when the data input is not stable from SETUP nanoseconds in the following circumstances:

- Before the rising edge of the clock
- While the clock is rising
- While the clock is high
- During the falling edge of the clock
- Until HOLD nanoseconds after the clock has gone low

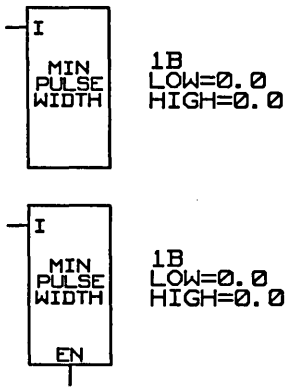
The SETUP RISE HOLD FALL primitive has two default body properties attached:

SETUP = 0.0
HOLD = 0.0

The properties SETUP and HOLD are assigned the required property values by using the GED **change** command. This primitive is used to check the setup and hold times of data being written into memories.

The primitive has an optional enable input that can be used to turn off checking. If the enable input is used, then any value other than zero causes checking to be enabled. If checking is enabled at any time between the rising edge and the falling edge, checking is performed for that clock pulse.

MIN PULSE WIDTH Primitive



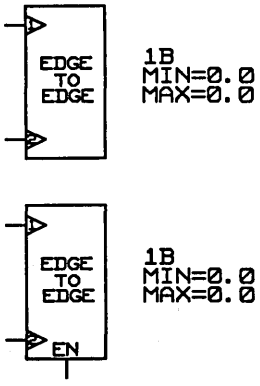
The MIN PULSE WIDTH primitive has one data input. It checks that its data input has no pulses on it that are low for less than LOW ns, and no pulses on it that are high for less than HIGH ns. The MIN PULSE WIDTH primitive has two default body properties attached:

LOW = 0.0
HIGH = 0.0

The properties LOW and HIGH are assigned the required property values by using the GED **change** command.

The primitive has an optional enable input which turns checking on and off. If the enable input is any value other than zero, then checking is enabled. If checking is enabled any time during a given pulse, then the width of that pulse is checked.

EDGE TO EDGE Primitive



The EDGE TO EDGE primitive has two inputs, CK1 and CK2. It checks that the beginning of a rising edge on CK2 is at least a minimum delay from the end of a rising edge on CK1 and that the end of a rising edge on CK2 is no more than a maximum delay from the beginning of a rising edge on CK1. The EDGE TO EDGE primitive has two default body properties attached:

MIN = 0.0
MAX = 0.0

Use the GED **change** command to assign values to the MIN and MAX properties. Only rising delays are used.

The primitive has an optional enable input that turns checking on and off. If the enable input is any value other than zero, then checking is enabled. If checking is enabled any time during the rising edge of CK1, then checking is performed for that edge. If there is no edge on CK2 (that is, if CK2 does not change state), then no error message is generated.

Timing Properties

When creating timing models, you can size timing bodies and add delay values to each primitive using body and pin properties. Timing properties are:

- DELAY
- FALL
- RISE
- SIZE

All Timing Verifier delay properties can be attached to a signal or a pin. The delay is applied at each input pin to which the wire with the delay property (or signal name containing the delay property) is attached. For designs where delays are related to changes in output loading, temperature, and voltage, the Delay Estimator can be used.

Table 5–2 summarizes the timing properties. For detailed information on timing properties and directives, refer to the *ValidTIME Reference Manual.*

Table 5–2. Timing Properties

Prop Name	Controlling Directive(s)	Comments
DELAY	DELAY_MODEL RISE_FALL_MODELS	Specifies the delay time as a single value or the rise/fall delay values as *min, max,* or *min–max*. DELAY_MODEL directive selects which value to use for current timing run; *min–max* is the default.
RISE	DELAY_MODEL RISE_FALL_MODELS	Assigns RISE delay to a timing primitive. The directives DELAY_MODEL and RISE_FALL_MODELS select delay values.
FALL	DELAY_MODEL RISE_FALL_MODELS	Assigns FALL delay to a timing primitive. The directives DELAY_MODEL and RISE_FALL_MODELS select delay values.
SIZE		Specifies the number of bits on a pin. Can also use SIZE=SIZE to match primitive size to the SIZE property attached to the body drawing.

The DELAY Property

Delays are given in nanoseconds. Primitives without an explicit DELAY are assumed to have a delay of zero. By convention, primitives are given delays to model the worst-case behavior of the part being modeled, but this is not required. The syntax of the DELAY property in timing models is:

SYNTAX

DELAY=*min*
DELAY=*max*
DELAY=*min-max*

The Timing Verifier's DELAY_MODEL directive selects which value (*min, typ,* or *max*) to use for current timing run. *Min* tells the Timing Verifier to use only minimum delays. *Max* tells the Timing Verifier to use only maximum delays. *Min-max* is the default. It tells the Timing Verifier to use both the minimum and maximum available delays.

For the timing to function correctly, it is only necessary to define one possible timing behavior of the part (*min* or *max*). Exercise care when specifying delay values for parts; in particular, zero-delay parts can result in unexpected behavior in a circuit.

The RISE Property

Rise delays are specified using the RISE property with a rise delay time value. The Timing Verifier's DELAY_MODEL and RISE_FALL_MODELS directives are used to select delay values.

The FALL Property

Fall delays are specified using the FALL property with a rise delay time value. The Timing Verifier's DELAY_MODEL and RISE_FALL_MODELS directives are used to select delay values.

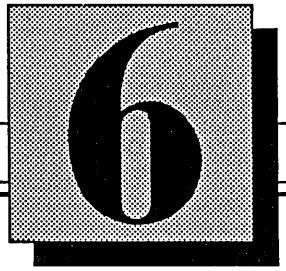
The SIZE Property

Most timing primitives can have a SIZE property to specify the number of bits on a pin. A primitive can be given the property SIZE=SIZE, which means that the size of the primitive is taken from the SIZE property attached to the part being modeled.

Many primitives have inputs and outputs that are not affected by the SIZE property. All enable inputs, clock inputs, and chip select inputs have a fixed width of one bit. The select input of an eight-bit multiplexer is always three bits wide.

Modifying Timing Models

Timing models are carefully designed to correctly model the part while including no extraneous data. The smaller the model, the faster the Timing Verifier can run. Models are therefore difficult to intuitively grasp; make as few changes to models as absolutely necessary. The most frequent change to a timing model is the change of the value of a delay property. Use the **change** command in GED to change the value. Be careful to change the correct instance of the delay property in the model. To accommodate different delay paths through the device, propagation delays are usually divided up and placed in different locations in the model. Examine the model carefully to choose which delays you need to change.



Creating Support Components

This section discusses creating and using the following support components:

- Connector
- Resistor pack
- Ground

Creating A Connector

There are many sizes of connectors you might require to complete a design. Rather than creating each size connector, you can create a generic "pin" connector that you can use to design any size connector.

Follow this procedure to create a generic connector:

1 Access GED and edit a new body drawing:

```
edit conn128.body.1.1
```

2 **split** the body name (note) away from the body origin and delete the body name.

3 Use the **wire** command to create a single pin shape (⊢×–). Center the pin on the origin.

4 Use the **dot** command to add a connection point to the pin (⊢×●).

5 Use the **signame** command to attach the pin name:

```
CON_PIN<SIZE-1..0>\NAC
```

The \NAC (no assertion check) property tells the compiler that both high-asserted and low-asserted signals can be attached to the pin. Place the pin name close to the pin.

6 Attach the $PN (pin number) property to the pin and give the property the value "?" (ques-

tion mark). Place the property to the left of the pin, display it right-justified (so text expands to the left and does not overwrite the pin), and display only the property value (?).

7 Attach the following properties to the body origin:

```
LOCATION=?
PATH=?
```

Place the LOCATION property immediately above the pin, display it left-justified, and display only the property value. Place the PATH property to the left of the pin, display it right-justified, and make it invisible.

The *conn128* connector is shown in Figure 6-1. The top drawing shows all the attached properties, and the bottom drawing shows how the *conn128.body* drawing should actually appear.

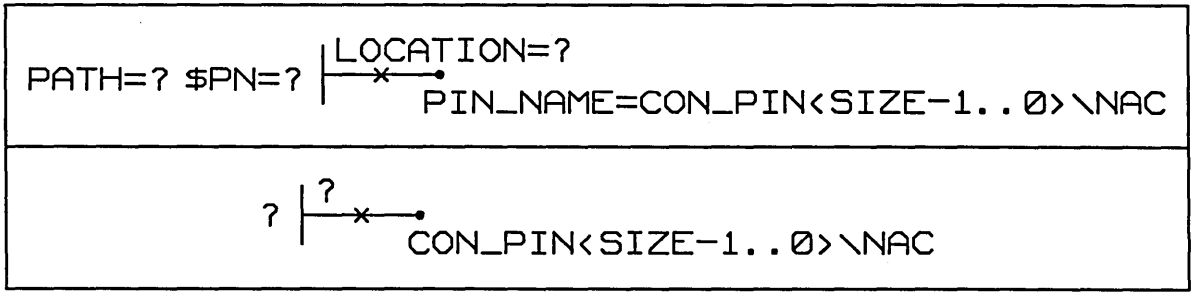


Figure 6-1. Body Drawing for CONN128 Connector

Tips on Attaching Connector Properties

- Attaching the LOCATION, PATH, and $PN properties during component creation allows you to define their default positions on a drawing.
- Making the LOCATION property a hard property (not starting with a $) encourages the user to assign a value to the property before the schematic can be written.
- Attaching the $PN property as a soft property allows the system to assign the pin number but locates the number where you want it.

Creating a Second Version of the Connector

You can create a second version of the *conn128* connector where the pin extends to the left of the origin (an output connector). The properties are the same as the original version, but their placement is different. Make the following changes to the attached properties:

- Move the PATH and $PN properties to the right of the pin. Make sure the $PN property has enough space to print up to a three-digit pin number.
- Make the PATH property display left-justified and the LOCATION property display right-justified. (The $PN property remains right-justified.)

Version 2 of the *conn128* connector is shown in Figure 6–2. The top drawing shows all the attached properties, and the bottom drawing shows how the *conn128.body* drawing should actually appear.

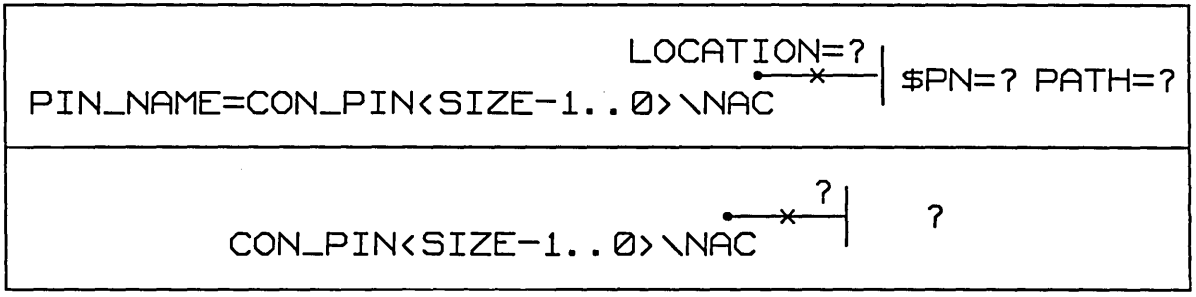


Figure 6–2. Body Drawing for Version 2 of the CONN128 Connector

The .PART Drawing

To define the connector as a lowest-level component, create a .PART drawing that contains a DRAWING body with the TITLE and ABBREV properties attached. The .PART drawing for the *conn128* connector is shown in Figure 6–3.

```
DRAWING

TITLE=CONN128
ABBREV=128-PIN_CON
LAST_MODIFIED=Fri Sep 15 11:14:20 1989
```

Figure 6–3. Part Drawing for CONN128 Connector

The Physical Model

Next, create the physical model of the *conn128* connector by adding physical information to a library drawing. Follow these steps:

1. Create a library drawing and add the *conn128* connector body:

```
edit conn library.logic
add conn128.body
```

2. Attach the PIN_NUMBER property. Use the compact pin number syntax discussed in Section 3 to enter the pin numbers:

```
PIN_NUMBER=<128..1>
```

3. Add a DRAWING body and attach the TITLE and ABBREV properties.

4. Verify property attachments, check and write the drawing.

Note: When you check and write the *conn library* drawing, you get the following error message:

```
"LOCATION" property is only a placeholder
...done checking
CHECK detected problems with this drawing.
Type ; to write anyway or anything else to abort
```

Enter a semicolon to write the drawing anyway. The LOCATION property will be defined when the connector is added to a drawing.

The *conn library* drawing now looks like the one in Figure 6–4.

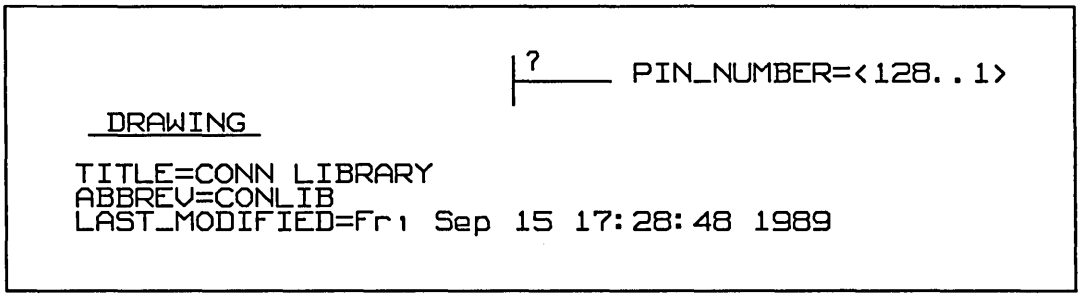


Figure 6–4. Conn Library Drawing for the CONN128 Component

Once you enter all the physical information into the library drawing, follow these steps to complete the physical model:

1. Edit the *compiler.cmd* file, change the ROOT_DRAWING to *conn library,* redirect OUTPUT from *logic* to *chips*, and make sure the correct *.wrk* file is specified.

2. Compile the library drawing to create the *chips.dat* file.

3. After the drawing compiles successfully, change the name *chips.dat* to *chips_prt* and move the physical model to the correct component subdirectory:

```
mv chips.dat conn128/chips_prt
```

The *chips_prt* file for the *conn128* connector looks similar to the one shown in Figure 6–5.

```
FILE_TYPE=LIBRARY_PARTS;
TIME='COMPILATION ON FRI SEP 15 13:43:34 1989';
primitive 'CONN128':
  pin
    'CON_PIN'<0>;
     PIN_NUMBER='(128..1)';
  end_pin;
  body
    BODY_NAME='CONN128';
    .
    .
    .
  end_body;
end_primitive;
END.
```

(The body properties differ from file to file depending on the physical requirements of a component.)

Figure 6-5. CONN128 Chips_prt File

Creating Additional Physical Models

Once you create the body drawing for the generic connector pin, you can create other components using the same body drawing. For each new component you want to create, you need to:

- Copy the body drawing
- Create a new part drawing
- Edit the library drawing
- Compile the component

Follow these steps to use the *conn128* connector to create a connector called *din3_32*, a 96-pin connector that has three rows of 32 pins each.

You can copy both versions of the conn128 *to make both versions of the* din3_32 *if necessary.*

1 Edit the *conn128.body* drawing and use the **diagram** command to rename the drawing:

```
edit conn128.body
diagram din3_32.body
```

2 **write** the new drawing to save it.

3 Create a .PART drawing for the new component.

4 Edit the *conn library* drawing and use the **change** command to alter the PIN_NUMBER property:

```
pin_number=<C32..C1,B32..B1,A32..A1>
```

5 Check and write the drawing. You still get the error message about the LOCATION property. Enter a semicolon to write the drawing anyway. The LOCATION property will be defined when the connector is added to a drawing.

6 Compile the library drawing (output chips) to create the *chips.dat* file.

7 After the drawing compiles successfully, change the name *chips.dat* to *chips_prt* and move the physical model to the correct component subdirectory:

```
mv chips.dat din332/chips_prt
```

The system name for the din3_32 connector does not contain an underscore.

The *chips_prt* file for the *din3_32* connector should look similar to the one shown in Figure 6–6.

```
FILE_TYPE=LIBRARY_PARTS;
TIME='COMPILATION ON MON SEP 18 08:55:14 1989';
primitive 'DIN3_32';
  pin
    'CON_PIN'<0>:
     PIN_NUMBER='(C32..C1,B32..B1,A32..A1)';
  end_pin;
  body
    BODY_NAME='DIN3_32';
    .
    .
    .
  end_body;
end_primitive;
END.
```

Figure 6–6. DIN3_32 Chips_prt File

Creating a Connector Break

You can use the *conn128* body to create connectors ranging from one pin to 128 pins. When you create a small connector, it is simple to show the whole connector in one place on a schematic. But if you create a large connector, it may not be necessary or even possible to show all the pins in the same location on a schematic. A simple comment–body called a *connector break* indicates that the connector shown is only a portion of the complete connector. Follow these steps to create a *connector break* body:

Since the connector break is a comment–body, it does not need a .PART drawing and it is not compiled.

1. Edit a body drawing called *conbrk* and delete the body name (note).
2. Use the **wire** command to create the following shape:

   Center the shape around the body origin.
3. Use the **dot** command to add connection points to the ends of the wires:

4. Use the **signame** command to attach an NC signal name to each end (since the pin can be left unconnected):

5. Attach the property COMMENT_BODY=TRUE to the body origin and make the property invisible.

Using the Connector and Connector Break Bodies

To create connector packages on a logic drawing, you add as many connector pins and connector breaks as you need. For example, you might need to show eight pins of a 128–pin connector in one place on a logic drawing. Follow these steps to create the necessary connector on a schematic:

1 Edit your logic drawing and add a connector pin:

```
edit test
add conn128
```

2 Use the **change** command to assign a value to the LOCATION property, then make the property invisible.

3 Select the **copy** command and copy the connector pin seven times. Place the copies below the original pin.

4 Add a *conbrk* body and place it slightly below the connector pins (so that it is not overwritten by any pin numbers).

5 Use the **wire** command to complete the connector shape. Start the wire at one end of the connector break and place it over the connectors to connect them. Use the **signame** command to attach the same signal name as you assigned to the LOCATION property.

6 Use the **section** command to assign pin numbers to each connector pin.

You can also add a conbrk *to the top of a connector body.*

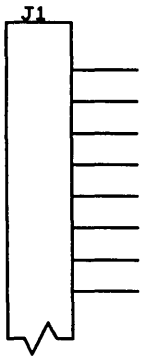


Figure 6–7 shows several versions of the connector created using both versions of the *conn128* and *din3_32* bodies.

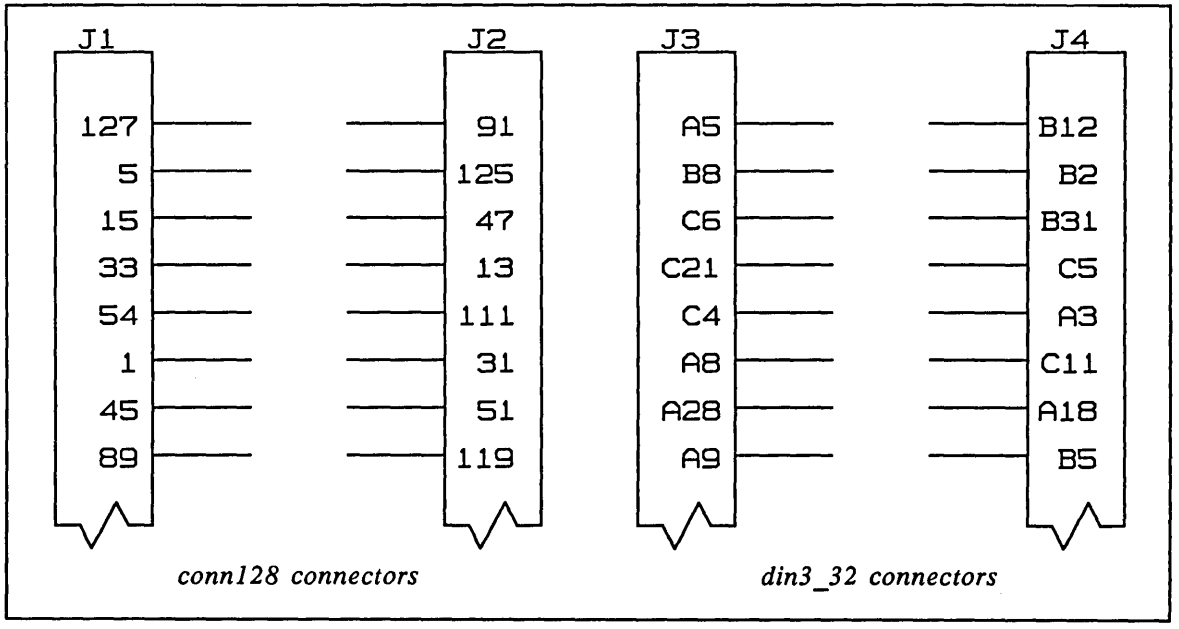


Figure 6–7. Connector Examples

Simulation and Timing Models

The simulation information is provided in Section 4 and the timing information is provided in Section 5.

Create a simulation or timing model of the *conn128* body to emulate the behavior and operation of the connector. The .SIM and .TIME models for the connector are identical. Both models are synonymed (with the synonym body from the Standard library) to the signal name NC because the connector is the logical completion of the circuit.

Figure 6-8 shows the simulation or timing model for the *conn128* connector.

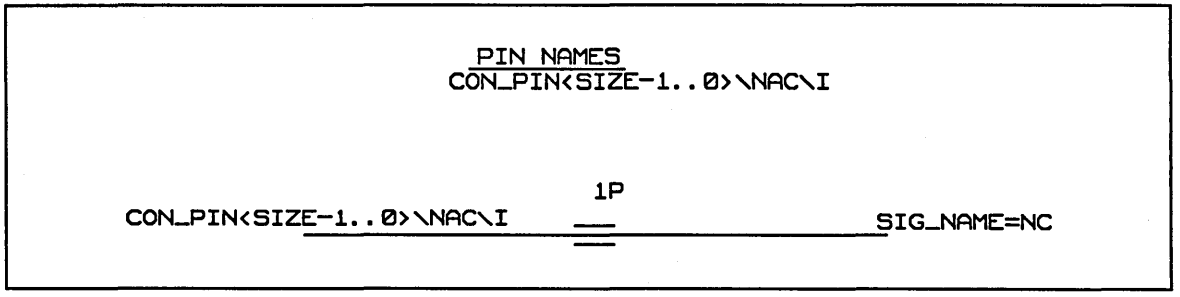


Figure 6-8. Simulation or Timing Model for the CONN128 Connector

Creating A Resistor Pack

You can create a sizeable resistor that you can use alone or that you can group to create a standard resistor pack. Use GED to create the resistor body *res5 sip6.body.1.1* (5 resistor, 6-pin, single in-line package) shown in Figure 6-9. Use the **property** command to add the property VALUE=? and display only the property value.

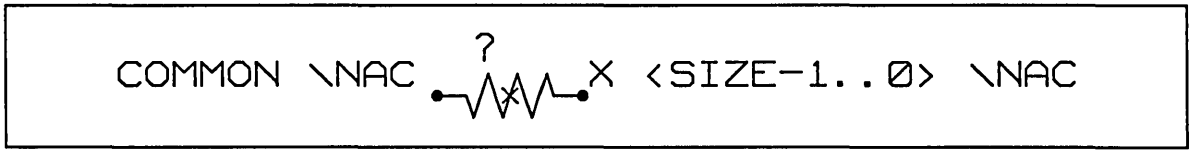


Figure 6-9. Version 1 of the RES5 SIP6 Resistor

This standard resistor pack includes five resistors with one common pin. Create the resistor pack as the second version of the original resistor. The *res5 sip6.body.2.1* is shown in Figure 6-10. Attach the VALUE property to the origin and display only the property value. Attach the HAS_FIXED_SIZE=5B property to the body origin and make the property invisible.

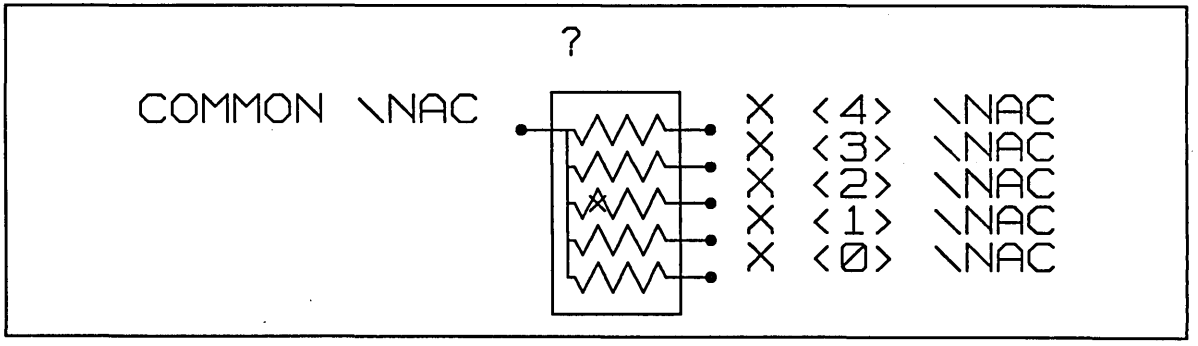


Figure 6-10. Version 2 of the RES5 SIP6 Resistor

The .PART Drawing

To define the resistor as a lowest-level component, create a .PART drawing which contains a DRAWING body with the TITLE and ABBREV properties attached. The .PART drawing for the *res5 sip6* resistor is shown in Figure 6–11.

```
  DRAWING
TITLE=RES5 SIP6
ABBREV=R5S6
PART_NAME=RES5SIP6

LAST_MODIFIED=Sun Oct  1 16:54:38 1989
```

Figure 6–11. Part Drawing for RES5 SIP6 Resistor

The Physical Model

Create the physical model of the *res5 sip6* resistor by adding physical information to a library drawing. The drawing should look like the one shown in Figure 6–12.

```
                              PATH=1P
                              PART_NAME=RES5SIP6
                              UNKNOWN_LOADING=TRUE
                              ALLOW_CONNECT=TRUE
                              PHYS_DES_PREFIX=Z

PIN_NUMBER=(1, 1, 1, 1, 1)  ?     PIN_NUMBER=(6, 5, 4, 3, 2)
                            -/\/\-

DRAWING:
  TITLE=RES_LIB
  ABBREV=RES5
  LAST_MODIFIED=Mon Oct  2 15:00:35 1989
```

Figure 6–12. Library Drawing for the RES5 SIP6 Resistor

Note:

When you check and write the *res lib* drawing, you get the following error message:

```
"VALUE" property is only a placeholder
...done checking
CHECK detected problems with this drawing.
Type ; to write anyway or anything else to abort:
```

Enter a semicolon to write the drawing anyway. You define the VALUE property when the resistor is added to a drawing.

After you enter all the physical information into the library drawing, compile the library drawing (with output chips) and move the *chips.dat* to the *chips_prt* file in the *res5sip6* subdirectory.

Physical Part Tables

You can use physical part tables to create different types (versions) of resistors using *res5 sip6* resistor body. Resistor versions can vary in resistance value, power dissipation, cost, or tolerance. For information on creating and using physical part tables, refer to the *ValidPACKAGER Reference Manual.*

Simulation and Timing Models

The .SIM and .TIME models for the *res5 sip6* resistor are identical. Figure 6–13 shows the simulation or timing model for the *res5 sip6* resistor.

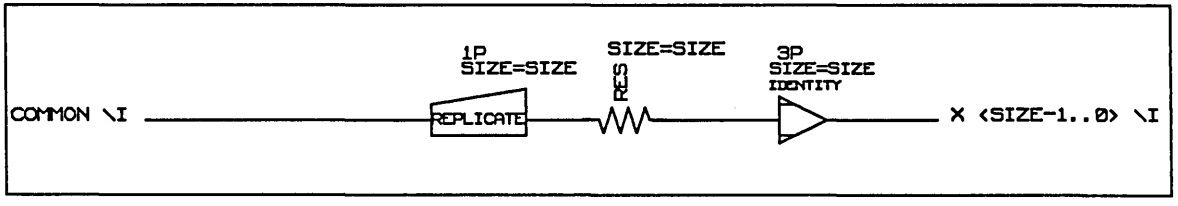


Figure 6–13. .SIM or .TIME Drawing for the RES5 SIP6 Resistor

Creating A Ground

You can create a ground (logic 0) body to use as a permanent enable input. Use GED to create the symbol shown in Figure 6–14. Use the **display heavy** command to change the thickness of the horizontal lines.

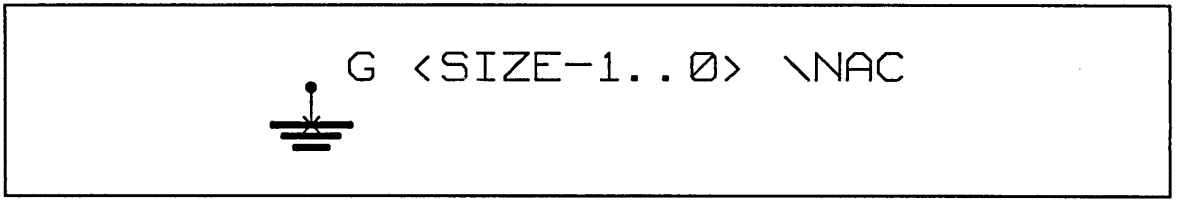


Figure 6–14. BODY Drawing for the GND Ground

Making the ground body sizeable allows you to attach the ground to a sizeable component (for example, a pull-down through a sized resistor on a bus) without having to replicate the ground each time. You can ignore the size capabilities of the ground if you do not need a sizeable ground, since SIZE=1B is the default value of the SIZE property.

The .LOGIC Drawing

Because the ground body is not an actual physical body, there is no .PART drawing defined for the component. The ground body requires a .LOGIC drawing to allow the Packager to resolve the ground body into a net name.

For more information on the replicate parameter (\R), refer to the SCALD Language Reference Manual.

Figure 6–15 shows the .LOGIC drawing for the *gnd* component. The pin name for the ground is synonymed to *gnd* so that the entire ground net in a schematic is synonymed to the same name. The \R SIZE parameter replicates the ground to the size specified by the SIZE property attached to the body when it is added to a logic drawing. The \G global parameter forces the entire net to have the same net name when a schematic that includes the *gnd* body is compiled.

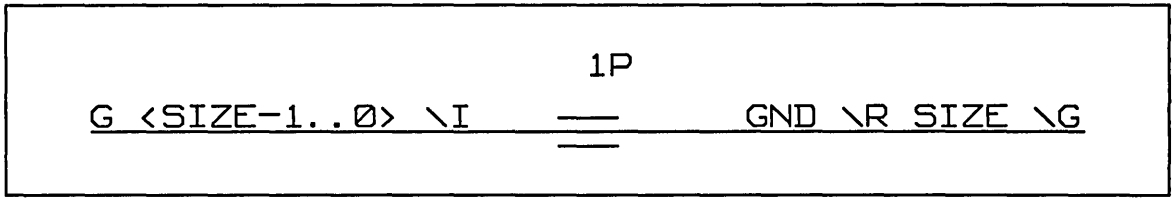


Figure 6–15. LOGIC Drawing for the GND Component

Simulation and Timing Models

Figure 6–16 shows the .SIM or .TIME model for the *gnd* body. The pin name for the ground is synonymed to zero because the ground in a logic design is typically zero. The \R SIZE parameter replicates the ground to the specified size when it is added to a logic drawing. There is no \G global parameter attached since constants are global by nature.

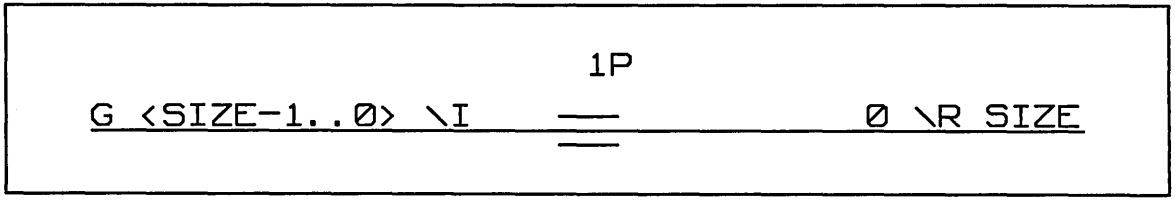
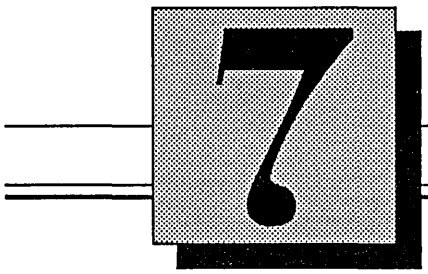


Figure 6–16. Ground .SIM or .TIME Drawing



Testing the Library

This section discusses:

- Creation checklist
- Testing issues

Creation Checklist

The following checklist summarizes some important points to remember when creating models. If problems arise, check these areas to make sure the new information meets the desired standards.

- ✓ Are all the library files owned by the librarian?
- ✓ Do the model shape and the signal names used on the model correspond to your corporate conventions?
- ✓ Do the signal names follow the correct Valid Library Format?
- ✓ Do the signal names follow the correct bit ordering?
- ✓ If you are having problems accessing the library where the model resides, is the library listed in the *master.lib* file?
- ✓ Are you using a test directory instead of the final directory to develop models for each part (and avoiding problems with a production directory)?
- ✓ Are new components approximately the same size and shape as other library components so they integrate well with other components on the schematic?
- ✓ Is your component centered around the body origin?

- ✓ Are high-asserted pins drawn with a 0.1-inch wire and low-asserted pins with a 0.1-inch bubble (circle)?
- ✓ Are clock pins marked with a clock wedge if required?
- ✓ Did you use bus-through pins whenever possible? Are their pin names the same as the visible pin to which they are connected?
- ✓ Do low-asserted pin names end with the correct low-assertion character (an asterisk in Library Format 1)?
- ✓ Are all properties correctly attached?
- ✓ Are the \NAC and \NWC properties attached to pins on bodies when necessary?
- ✓ Does the body contain notes detailing each pin name and the purpose of the body?
- ✓ If it is not a sizeable component, is the NEEDS_NO_SIZE property attached to the body?
- ✓ If it is the flat representation of a sizeable component, is the HAS_FIXED_SIZE property attached to the body?
- ✓ Are pins assigned to bubble groups when appropriate?
- ✓ Are pins that should start in the bubbled state defined with the BUBBLED property?

- Have you created all the required versions of a body?
- If you used the **smash** or **diagram** commands to create a new component, did you also copy and change the .PART, .SIM, and .TIME drawings if necessary?
- Did you create a .PART drawing for the new component?
- Do asymmetrical components have one of each section represented in the library drawing?
- Are bodies that have no logical function "commented out" with the property COMMENT_BODY=TRUE or the property BODY_TYPE=COMMENT?
- Does each pin of each component have a PIN_NUMBER property attached (except bus-through pins)?
- Do different sections of asymmetrical components have different pin names?
- Do all pins have an INPUT_LOAD or OUTPUT_LOAD property (or both)?
- Did you convert the INPUT_LOAD or OUTPUT_LOAD to the required measurement (typically milliamps) if the data book used a different unit of measurement?
- Are swappable pins assigned to the same pin group?

- ✓ Is the OUTPUT_TYPE property attached to the output pins of open-collector, open-emitter, and tri-state components?
- ✓ Are any necessary load-checking properties attached to the library drawing?
- ✓ Did you rename the *chips.dat* file to *chips_prt* and move it to the correct subdirectory?
- ✓ Are all the nets on the simulation model named so that there are no ambiguous error messages due to unnamed signals?
- ✓ Did you make sure GED can access the required libraries (such as Sim and Time)?
- ✓ Do all the interface signals in the simulation and timing models include the \I interface property in the signal name?
- ✓ Do all interface signals that require an explicit width have the width specified?
- ✓ Do the simulation and timing models have the correct body and pin delay properties?
- ✓ If pin delay properties are used in the simulator model, is the PIN_DELAY directive set to ON in the *simulator.cmd* file? Is the RISE_FALL directive ON if the PRISE and PFALL properties are to override the PDELAY property?
- ✓ Are all delays calculated in nanoseconds?

- Is the DELAY_MODE directive in the *simulator.cmd* file set to the correct value *(min, typ, max)*?
- Did you check for zero-delay parts in the simulation and timing models to eliminate unexpected behavior in the circuit?
- Did you add the PIN NAMES component to the timing model to check that all the pins are accounted for (and spelled correctly)?

Testing Issues

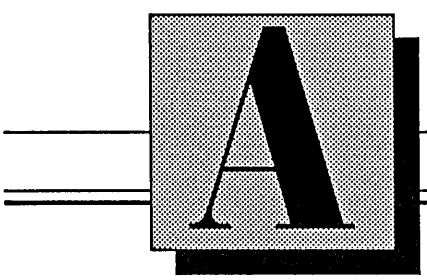
There is no one program that tests all aspects of a component. The main issue is to test the component for accuracy throughout the various design tools you use for your components.

The minimum testing of a new component should include compiling the bodies and the models. Any syntax errors are discovered during compilation and can be corrected. Complex parts may take a long time to verify; be sure to budget enough library development time for testing.

- If you have more than one component to test, you can create an EXAMPLE OF EACH *LIBRARY* PART drawing and use it for testing, since when you compile that drawing, it invokes all versions of all parts.
- Make sure you can add all versions of a new component.
- Try the **bubble** command on appropriate pins to make sure they are defined with a bubbleable pin and they are in the correct bubble group, if any.
- Attach a wire to each connection point to be sure that the wire attaches to the correct location.
- Wire bus-through pins to check that they connect correctly.
- For sizeable components, first set the SIZE parameter to 1 (the default) and then to some

other value to test the vector part implementation.

- Exercise timing models to make sure that the model behaves correctly and that the DELAY, RISE, and FALL property values have been correctly assigned.
- If a directive or property has more than one possible value, test the model using the entire range of values.
- Generate errors in setup and pulse width to make sure that the signals reported by the Timing Verifier have names that are easily understood without looking at the model; all errors should be reported in terms of the pins of the part.
- Use the TIMES property to verify drive and load capabilities. Try overloading the connections and checking the resulting error messages from the Packager to be sure all loading violations are detected.
- Use the **section** command to make sure the component sections correctly.
- Use the **pinswap** command to make sure the pin groups have been defined properly.



Text File Method of Adding Physical Information (UNIX Only)

This section discusses:

- Using *phys_dat* to add physical information
- *Phys_dat* file syntax
- Pin number formats for the *phys_dat* file

Using Phys_dat to Add Physical Information

On the UNIX operating system, you can use a text file method to specify the physical information for a library component. The text file is called the *phys_dat* file, and the method is similar to the library drawing method except that the pin and body properties are entered into a text file rather than directly onto a drawing. This method may prove faster, especially with reference elements with a large number of pins. Once the property information is entered into the text file, the *chips.dat* file is automatically updated with the information from the file by running the *addphysinfo* script. Follow these steps to create a *chips_prt* file using the *phys_dat* file:

1. Access GED and edit a library drawing.
2. Add the sizeable version (Version 1) of the component to be compiled.
3. Add a DRAWING body and attach the TITLE and ABBREV properties.
4. **write** the drawing to save it.
5. Edit the *compiler.cmd* file and change the ROOT_DRAWING directive to the correct library name. Make sure the correct *.wrk* file is specified.
6. Compile the library drawing (output chips) to create the *chips.dat* file.
7. Move to the component subdirectory and use the system text editor to create a new file

The chips.dat *file is only a template at this point.*

called *phys_dat*. The file requires the same information as described in Section 3, *The Physical Model*:

- Part name
- Family
- Power and ground pin assignments
- Pin name
- Pin number(s)
- Output type
- Input/output load
- Bidirectional pins

The information must be entered in the order shown above. Figure A–1 shows a *phys_dat* file for the TTL 293 component.

```
PART 74293
 FAMILY TTL
 POWER_PINS (VCC:14;GND:7)
PINS
 Q<3>            (8)     OUTPUT   (16.0,-0.8)
 Q<2>            (4)     OUTPUT   (16.0,-0.8)
 Q<1>            (5)     OUTPUT   (16.0,-0.8)
 Q<0>            (9)     OUTPUT   (16.0,-0.8)
 CLKA*           (10)    INPUT    (-3.2,0.08)
 CLKB*           (11)    INPUT    (-3.2,0.08)
 CL1             (12)    INPUT    (-1.6,0.04)
 CL2             (13)    INPUT    (-1.6,0.04)
END
```

Figure A–1. Phys_dat File for the TTL 293 Component

8 Move up one directory level and execute the *addphysinfo* script on the *chips.dat* file:

```
cd ..
addphysinfo chips.dat
```

The *addphysinfo* script automatically updates the *chips.dat* file to include the information in the *phys_dat* file. The script creates several files that all begin with the prefix *lib*. For example, the *liblst.dat* file contains a summary of execution. After an error-free run of the script, these files can be deleted.

9 Change the name *chips.dat* to *chips_prt* and move the physical model to the correct component subdirectory:

```
mv chips.dat 293/chips_prt
```

Phys_dat Syntax

The *phys_dat* file specifies the same information as the library drawing but uses a different entry format. A pin entry in the *phys_dat* file has the following syntax:

SYNTAX

pin (*pin_id*) *output_type* (*load*) [(*load*)] **[bidir]**

Separate the values in the pin entries with spaces or tabs.

pin — The pin name that was assigned in the body drawing. Pin name entries can be scalar or vector.

pin_id — Any combination of alphanumeric characters and/or the underscore character that defines the pin number. Pin numbers can be scalar or vector, single or multiple section, or asymmetrical. The pin number formats are described beginning on page A-6.

output_type — Defines whether the pin is an input pin, output pin, open collector pin, or tri-state pin.

load — The input/output loading value of the pin. If the pin is a tri-state, there are two sets of load values to define both the input load and the output load.

bidir — Specifies a pin that is both an input pin and an output pin.

Section 2, *Component Creation*, contains more information on pin names. Section 3, *The Physical Model*, contains details on pin numbers, output types, loading values, and the bidirectional property.

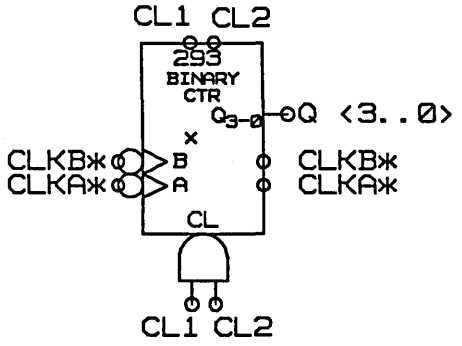
Pin Number Formats

The pin number formats described in Section 3 can also be defined in the *phy_dat* file:

- Single section scalar pin
- Single section vector pin
- Multiple section scalar pin
- Multiple section common pin
- Multiple section common vector pin
- Asymmetrical components

Single Section Scalar Pins

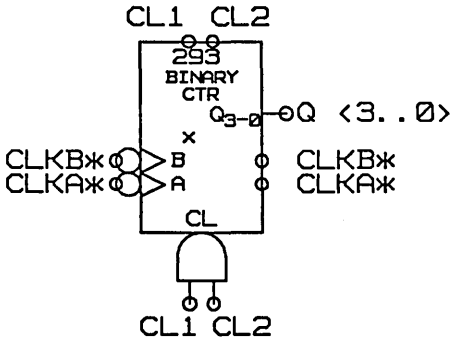
In single section scalar format, you enter each pin and its associated pin number individually in the *phys_dat* file. The *phys_dat* file for the TTL 293 component contains examples of single section scalar pin entries:




```
Q<3>        (8)         OUTPUT      (16.0,-0.8)
Q<2>        (4)         OUTPUT      (16.0,-0.8)
Q<1>        (5)         OUTPUT      (16.0,-0.8)
Q<0>        (9)         OUTPUT      (16.0,-0.8)
CLKA*       (10)        INPUT       (-3.2,0.08)
CLKB*       (11)        INPUT       (-3.2,0.08)
CL1         (12)        INPUT       (-1.6,0.04)
CL2         (13)        INPUT       (-1.6,0.04)
```

Single Section Vector Pins

Single section vector pins can be entered individually (like those of the TTL 293) or in a vector format. The Q pins of the TTL 293 can be shown in vector format:



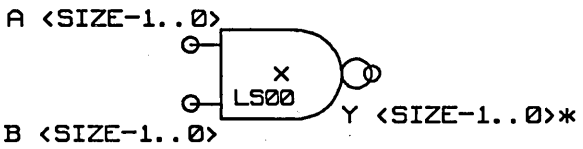

```
Q<3..0>     (8,4,5,9)      OUTPUT     (16.0,-0.8)
CLKA*       (10)           INPUT      (-3.2,0.08)
CLKB*       (11)           INPUT      (-3.2,0.08)
CL1         (12)           INPUT      (-1.6,0.04)
CL2         (13)           INPUT      (-1.6,0.04)
```

Multiple Section Scalar Pins

Each pin of a multiple section is defined by a list of pin numbers, one pin number for each section in the part. The pin numbers for each section are separated by commas. Pins for the first section are in the last position in the *phys_dat* file, pins for the second section are in the second–last position, and so on.

The LS00 component is an example of a sizeable component with multiple section scalar pins. If the component is given the property SIZE=4B, each logi-

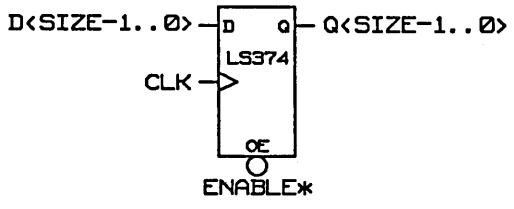
cal pin of the component has four pin numbers, one for each section:




```
A<0>        (1,4,9,12)      INPUT       (-0.4,0.02)
B<0>        (2,5,10,13)     INPUT       (-0.4,0.02)
Y<0>*       (3,6,8,11)      OUTPUT      (8.0,-0.4)
```

Multiple Section Common Pins

The LS374 octal register is an example of a component that has multiple sections with common pins. The pin numbers for multiple sections appear more than once in the file entry. Pins for the first section of the part appear in the last position in the *phys_dat* file, pins for the second section are in the second-last position, and so on.




```
D<0>    (3,4,7,8,13,14,17,18)               INPUT     (-0.4,0.02)
OE*     (1,1,1,1,1,1,1,1)                   INPUT     (-0.4,0.02)
CLOCK   (11,11,11,11,11,11,11,11)           INPUT     (-0.4,0.02)
Q<0>    (2,5,6,9,12,15,16,19)               TS        (-0.02,0.02)    (24.0,-2.6)
```

The clock and enable pins are common to all eight flip–flops in the package. The D and Q pins are defined so that one bit is assigned to each flip–flop.

Pins that are common only to certain sections of a component are represented in the same manner as pins that are common to all sections, except that these pin numbers are present only in the sections for which they are common. The LS367 component is an example of a component with multiple–section common pins:

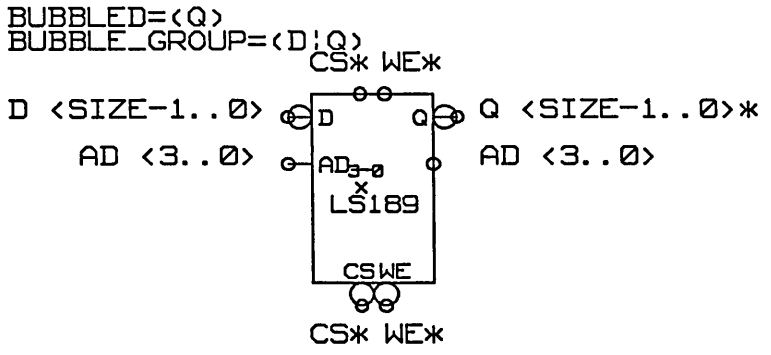
```
                BUBBLE_GROUP=(A:Y)
                    A <SIZE-1..0>
                                    OE*
                                  LS367
                                          Y <SIZE-1..0>
                                    OE*

A<0>      (2,4,6,10,12,14)      INPUT      (-0.4,0.02)
OE*       (1,1,1,1,15,15)       INPUT      (-0.4,0.02)
Y<0>      (3,5,7,9,11,13)       TS         (-0.02,0.02)     (24.0,-2.6)
```

The pin numbers for the open emitter pin show that one output enable pin is common to four sections of the component, and one output enable pin is common to the other two sections of the component.

Multiple Section Common Vector Pins

If a multiple-section component has vectored pins, each pin number and section are separated by commas. The pin numbers enclosed in angle brackets specify individual bits of the pin, not different sections for the pin.

An LS189 16-word by 4-bit RAM is an example of a component having vector pins and multiple sections.

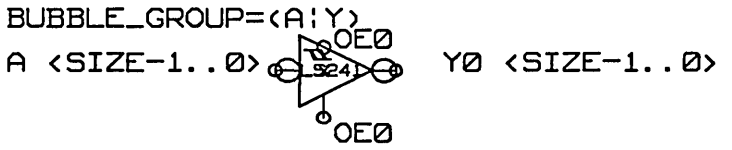


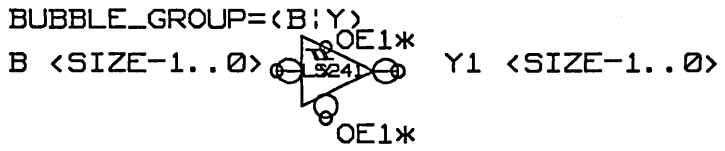

```
D<0>      (4,6,10,12)                                                  INPUT (-0.4,0.02)
AD<3..0>  (<15,14,13,1>,<15,14,13,1>,<15,14,13,1>,<15,14,13,1>) INPUT (-0.4,0.02)
WE*       (3,3,3,3)                                                    INPUT (-0.4,0.02)
CS*       (2,2,2,2)                                                    INPUT (-0.4,0.02)
Q<0>*     (5,7,9,11)                                                   TS    (-0.02,0.02) (24.0,-2.6)
```

Asymmetrical Components

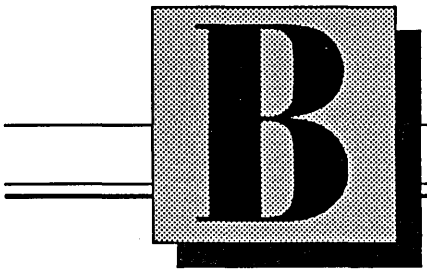
Components with multiple sections that are functionally different, such as the LS241 bus transceiver, need one version of the body defined for each type of section in the component. To identify which pins are present in a given section, the pins of the different versions must have **different** pin names.

Even though some pins may not be present in a section, the *phys_dat* file specifies all the sections of the component. Any pin that is not present in a given section is specified with a pin number of zero.






```
A<0>    (17,15,13,11,0,0,0,0)    INPUT    (-0.2,0.02)
OE0     (19,19,19,19,0,0,0,0)    INPUT    (-0.2,0.02)
Y0<0>   (3,5,7,9,0,0,0,0)        TS       (-0.02,0.02)   (24.0,-15.0)
B<0>    (0,0,0,0,2,4,6,8)        INPUT    (-0.2,0.02)
OE1*    (0,0,0,0,1,1,1,1)        INPUT    (-0.2,0.02)
Y1<0>   (0,0,0,0,18,16,14,12)    TS       (-0.02,0.02)   (24.0,-15.0)
```



Changes in the Library Drawing Method

The current library drawing method of creating the physical model of a component differs from the previous method. This section discusses:

- The previous method versus the current method
- Why the change occurred

Previous versus Current Method

Creating a library drawing used to require changing the entire library. Modifications in the Valid design tools make it more reasonable to create and maintain the physical model for one component at a time.

Creating a library drawing for one component at a time means that there is no need to use the *makechipsfiles* utility to separate the *.prt* file into separate *chips_ prt* files. You can simply rename the *chips.dat* file to *chips_ prt* and move it to the correct subdirectory.

Table B–1 shows the differences between the previous method of creating a library drawing and the current method.

Table B–1. Previous and Current Library Drawing Methods

PREVIOUS METHOD	CURRENT METHOD
1 Create a library drawing.	1 Create a library drawing.
2 Add the sizeable or vectored version of *each component* in the library to the drawing.	2 Add the sizeable or vectored version of *one component* to the drawing.
3 Attach the body properties.	3 Attach the body properties.
4 Attach the pin properties.	4 Attach the pin properties.
5 Verify property attachments, check and write the drawing.	5 Verify property attachments, check and write the drawing.
6 Edit the *compiler.cmd* file.	6 Edit the *compiler.cmd* file.
7 Compile the library drawing.	7 Compile the library drawing.
8 Move the *chips.dat* file to a file with the same name as the library and a *.prt* extension (for example, *lsttl.prt* or *newparts.prt*).	8 Move the *chips.dat* file to the individual component directory and rename it *chips_prt*.
9 Run the *makechipsfiles* utility on the *.prt* file to create the individual chips files (*chips_prt* files).	

Reasons for the Change

There are three reasons why the method of creating the physical model changed.

The *.prt* file is no longer necessary.

GED originally did not accept pin numbers on a component until the component was first compiled, packaged, and back annotated. When the **section** command was created, the system accepted pin numbers during schematic entry, but the command could not read the *.prt* file directly, so the physical information for the component was placed in a new file called *chips_prt*. This file was located in the component subdirectory along with the rest of the component files. Since the *chips_prt* file was new, the Packager still read the *.prt* file for physical information, and the **section** and **pinswap** commands read the *chips_prt* file. Now the Packager has the ability to retrieve information from the *chips_prt* file, so the *.prt* file is no longer required.

Modifying the entire library is not required.

The original method of creating a library drawing required updating the entire library each time you created or modified a library component. When a library drawing contains only the component that is being modified, you save time and effort creating the library drawing. Also, any manual changes you made to the physical information files for the other components in the library are not overwritten when you recompile the library.

Physical information for asymmetrical components does not get overwritten.

Asymmetrical components have always required that you manually modify the *chips_prt* file to include the physical information for all the sections of the component. When you update a single component instead of the entire library, you do not overwrite the manual changes to the *chips_prt* file for asymmetrical components, and you do not have to re-enter the changes by hand.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Z