

Viva!®

development software

3.1 Users Guide



starbridge®

Table Of Contents

Chapter 1: Getting Started with Viva	3
Getting Started with Viva	4
Viva Workspaces.....	4
Viva Tree Groups	4
Viva Preferences.....	5
Load.....	5
Ambiguous objects loaded in Sheets	5
Ambiguous objects loaded in Systems.....	5
Delete Unused Data Sets	6
Files	6
Window	7
I2ADL Editor	7
Compiler	7
Messages	8
Creating a Sample Project.....	9
Chapter 2: Viva Projects.....	13
Viva Projects	14
Managing Viva Trees	15
Chapter 3: Viva Libraries	17
Viva Libraries	18
Creating Libraries	18
Library and System Management	19
Ambiguous objects.....	20
Overloading and Underloading	23
Updating Library Versions.....	24

Table Of Contents

Sharing designs and libraries among multiple users	24
Chapter 4: Viva Sheets.....	25
Viva Sheets	26
Converting Sheets to Objects	28
Navigating Sheets	30
Viewing Sheets	31
Zoom	31
View Sheet in Bitmap Editor	31
Chapter 5: Viva Objects.....	33
Viva Objects	34
Object Footprints.....	37
Object Polymorphism.....	39
Object Attributes	42
Object Attributes with Special Meanings	51
Viewing Object Detail.....	53
Chapter 6: System Descriptions.....	55
System Descriptions	56
X86 System Description	56
FPGA Systems	56
Viva Platform Guides.....	57
Project Attributes.....	58
Viewing Project Attributes	58
Changing Project Attributes.....	58
X86 Versus FPGA Applications	59
Differences Executing for X86 vs FPGA	59

Accounting for X86/FPGA Differences	60
Chapter 7: Data Sets	61
Data Sets	62
Exposers and Collectors	63
Bit Pattern Data Sets (MSB, LSB, and BIN)	64
Custom Data Sets and the Data Set Editor	65
List Data Set.....	67
Casting Data	68
Data Set Recursion	69
Static Data Sets in Viva	70
Consistent Structure and Design	71
Chapter 8: Synthesizing Your Application	73
Synthesizing Your Application	74
Creating Executables.....	75
Running Saved Viva Executables	75
Resource Usage	76
Chapter 9: Advanced Tasks	77
Creating and Removing a Debugging Horn.....	78
Specifying Runtime Preferences.....	81
Disabling the intermediate data results display	81
Troubleshooting Compile Issues.....	82
Directory System Attributes	83
Reconfiguring FPGAs with the \$Spawn Object	85
Implementing \$Spawn	86
Precompiling Portions of a Viva Project.....	87

Table Of Contents

EDIF File Support	88
Chapter 10: Saved Files (text based)	89
Saved Files (text based)	90
Header	90
COM Dependency Reference	90
Data Set Definitions	91
Object Definition and Object Prototype	92
Object Definition	93
Behavior Definition	93
System Definition	93
Chapter 11: Go Done Busy Wait	95
Go Done Busy Wait	96
Usage Guidelines	97
Go – Input	98
Done – Output	99
Busy – Output	100
Wait – Input	101
GDBW_Clr Object	102
Chapter 12: Recursion	103
Recursion	104
Variant Overloading	105
Recursive Footprints and Data Set Polymorphism	107
Chapter 13: Synthesizer Graphics Display	113
Synthesizer Graphics Display	114
Chapter 14: Synching	115

Synching	116
Meeting Timing Constraints	118
Chapter 15: Memory Objects.....	119
On-Chip Memory Objects	120
RAM.....	120
Queue.....	121
Examples of using on-chip memory objects.....	121
Emulated RAM Objects.....	122
Chapter 16: Host to FPGA Communication	123
Behavioral Communications System (BCS)	124
Chapter 17: COM.....	125
COM.....	126
Registering DLL/OCX	127
COM and Viva.....	128
Dynamic COM Objects	131
Object Browser.....	133
COM Form Designer	134
Creating COM Components with Microsoft Visual C++ 6.....	135
Creating Simple COM Components	135
Adding methods to COM components	136
Removing a Function	139
Building an Events.....	140
Index	143

Introduction to Viva®

Viva is a Windows-based graphical development environment.

Traditionally, FPGA development has required the use of low-level, text-based design languages, such as VHDL and Verilog that were originally designed for circuitry layout rather than for high-performance and embedded computing. Starbridge observed a need to get faster turnaround at a lower cost when solving complex problems. A need to rapidly accelerate large, computationally intensive applications or algorithms in order to decrease the time and expense required. FPGAs are recognized in the industry for their ability to handle millions of instructions per second in parallel fashion. Viva is a new high-level, graphical, object-oriented programming tool that brings FPGA programming to a new level. Viva simplifies hardware design for applications, providing a platform for creating reusable and expandable library objects written for numerical methods.

Viva turns FPGA environments into general-purpose, high-performance and embedded computing systems capable of processing many algorithms in minutes that would otherwise take months. Viva makes the power of FPGAs accessible to researchers, scientists, and other hardware designers and programmers involved in high-performance and embedded computing. For example, developers can create algorithms for FPGA hardware platforms of any size, shape, or capacity and programmers can change research parameters on the fly, spending less time waiting for results. Programs written in Viva are efficient, parallel applications that enhance high-performance and embedded computing environments and any other FPGA-based hardware.

Chapter 1: Getting Started with Viva

Getting Started with Viva

The Viva development environment is comprised of projects, sheets, objects and data sets, and system descriptions. A project contains all the components for a particular application you may be working with. Sheets contain the designs you create while working in the I2ADL Editor in Viva where you assemble objects and map out the flow control between objects. Objects are functions that are created to manipulate types of data (data sets). Objects can be dragged onto a sheet, assembled into applications, or other objects. System descriptions are files which contain the data necessary for synthesizing to hardware.

Viva Workspaces

The Viva user interface is made up of the following workspaces where you perform the editing of your project:

- I2ADL Editor
- Data Set Editor
- Resource Editor
- System Editor

The I2ADL Editor is the primary workspace where you place objects and establish the data flow using transports which connect objects in the workspace. Use the Data Set Editor to create custom data sets. Tabs at the bottom left of the workspace enable you to specify workspaces.

Viva Tree Groups

The Viva user interface contains the following trees which are used to navigate the elements of your project and the structure of the object libraries:

- Project or Work in Progress (WIP) Tree
- Project Objects Tree
- System Objects Tree
- COM Objects Tree

The Project Tree is located in the upper-right frame of the Viva interface. The Object Tree frame is located at the lower-right of the Viva interface. Tabs at the bottom of the Object Tree frame enable you to specify which object tree to view.

There are several ways to expand and collapse tree groups. One option is to click the plus (+) or minus (-) icon next to the appropriate folder. Another method is to use the arrow keys. Use the right arrow key to expand and the left arrow key to collapse an objects folder. Also, double-click a folder to expand and collapse.

Viva Preferences

There are five separate tabs associated with the Viva preferences: Load, Files, Window, I2ADL Editor, and Compiler. These tabs allow you to customize how Viva looks and functions. To access these tabs, from the **Tools** menu, click **Preferences**.

Load

Use the load tab to determine the behavior of Viva as it reads in project, sheet, and system description files. The following load functions are available for modification:

Ambiguous objects loaded in Sheets

An ambiguous objects is an objects with the same name and the same footprint as another object but may differ in underlying design. Viva does not allow ambiguous objects. When loading sheets you can choose the following options regarding the handling of ambiguous object conflicts:

- Select **Retain Existing** to keep the objects currently used in the project.
- Select **Replace Existing** to replace the objects currently used in the project with the objects in the file you are loading.
- Select **Prompt** to have Viva prompt you when encountering an ambiguous object conflict. When prompted, Viva will specify the ambiguous object it has encountered and you have the option to retain the existing object or replace the existing object with the object in the file you are loading.

Ambiguous objects loaded in Systems

An [ambiguous object](#) is an object with the same name and the same footprint as another object but may differ in underlying design. Viva does not allow ambiguous objects. When loading system descriptions you can choose the following options regarding the handling of ambiguous object conflicts:

- Select **Retain Existing** to keep the objects currently used in the project.
- Select **Replace Existing** to replace the objects currently used in the project with the objects in the file you are loading.
- Select **Prompt** to have Viva prompt you when encountering an ambiguous object conflict. When prompted, Viva will specify the ambiguous object it has encountered and you have the option to retain the existing object or replace the existing object with the object in the file you are loading.

Delete Unused Data Sets

An unused data set is one that has no objects in use by the project on any of the application worksheets. Viva will automatically check for unused data sets. If Viva finds unused data sets, it performs the action you specify on the Load tab for unused data sets.

- Click **No** to keep unused data sets.
- Click **Yes** to delete unused data sets.
- Click **Prompt** to have Viva prompt you when loading files containing unused data sets. When prompted, you will have the option to keep or delete the unused data sets in entirety (not data set by data set).

Files

Use the Files tab to change the number of recently used files presented in their respective histories and specify the Viva behavior associated with the save commands. The file histories available are project, sheets, systems, COM forms, and executables. The number of recently used files displayed is changed either by entering a number directly or selecting a number using the spin boxes. The allowable range is 0 to 15 with the default value being 6 for each file type.

Select the **Save System Objects with Sheets** option to embed the currently used system objects in the application worksheets you are saving. This is normally not needed because systems are self-contained.

The **Automatically Change Version with Save** option allows you to automatically increment the file name each time it is saved. This will help you avoid an inadvertent loss of data because each save is placed in its own file. For example, if you are working on a project named MyProject, and it has already been saved to disk, having this option specified will change the file name to MyProject1. The rules for versioning files automatically are:

- If the name of the file being saved does not contain both alpha and numeric characters, then the new file name will be created by appending a "1" (one) or "a" to the original file name.
- If the last character of the file name being saved is numeric, then the new file name increments the numeric suffix. For example, MyProject1 becomes MyProject2; MyProject9 would become MyProject10, and so on.
- If the last character of the file name being saved is alphabetic, then the new file name increments the alphabetic suffix. For example, MyProject2004a becomes MyProject2004b, MyProject2004z would become MyProject2005a, and so on.

To keep from incrementing a file name while this preference is on, use the **Save As** option.

The **Skip Backup on Save** option turns off the specification to automatically rename a project, sheet, and system files. Instead, it applies a .bak suffix.

Window

Use the Window tab to choose whether the tree views in the right-hand pane automatically adjust their width to fit the contents of the tree. As the object/system tree panes expand and contract, the application workspace is adjusted as well.

The other setting on the Windows tab allows you to specify where the Viva application window is displayed in relationship to your desktop when the application is started. The options for this function are centered on screen, maximized, and the size and position of the window when Viva was last closed.

I2ADL Editor

In the I2ADL Editor tab, set the background color of the application workspace by entering a hex color number or by clicking the **Change** button and selecting a color from the displayed color palette. The default value is white.

The **Node Snap Tolerance** setting determines how close an object node needs to come to another object node to automatically connect. The default value is 11 pixels.

The **Confirm close of changed WIP (Work In Progress) sheets** option determines whether Viva will automatically discard changes to WIP sheets without a confirming prompt.

The **Confirm** section of this preferences tab allows you to turn on and off several confirmation dialogs provided by Viva. If you check the boxes provided, a confirmation message will be displayed when the associated action takes place. The default setting is for all boxes to be checked. For example, with the **Confirm Close of Changes & Unnamed Projects** box unchecked, changes to new projects will be discarded without a confirming prompt when the project is closed.

The **Confirm file name when saving new files** option indicates to Viva that whenever you save a new file with the default name, you will receive a prompt to specify the name. Otherwise, Viva will save the new file in the last used directory using the default name. For example, a project will be saved as 'new project' and a new sheet will be saved as 'sheet1'. The default is to have this option checked and to allow the default name to be changed when you save a new file.

Click **Close changed WIP sheets** for Viva to automatically close sheets that have been changed when you switch from one sheet to another. Click **Close unchanged WIP sheets** for WIP sheets that are viewed, but not changed, to automatically close when you switch to another sheet. This is useful in keeping the number of open WIP sheets down.

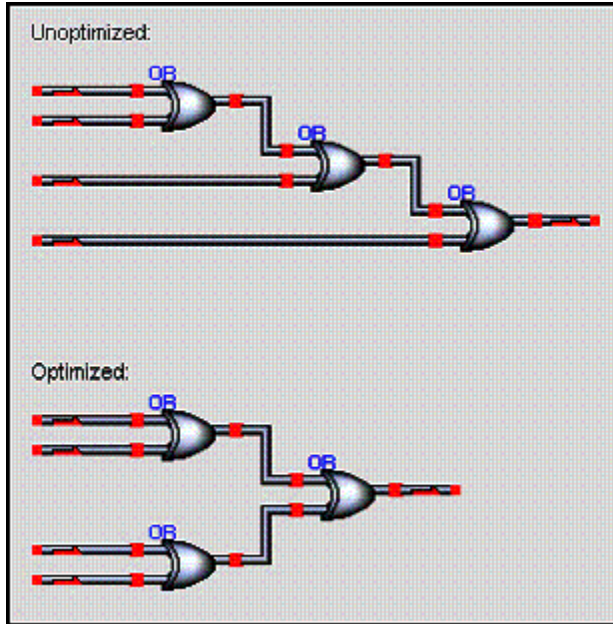
Compiler

Click **Automatically Run after compile** to execute the application after compiling. If you disable this feature, you can launch the application from the Sheet menu or by clicking the **Play/Stop Sheet** button.

The **Save project before compile** option causes Viva to automatically save a project each time you compile.

Optimize gates will arrange the AND and OR gates into a binary-ordered pattern. This process enhances parallelization of the program, which increases execution speed and helps to meet timing specifications. The following figure illustrates the function of the Optimize gates option.

Figure 1: Optimizing gates



Note: AND and OR gates with an output that connects to more than one downstream object will not be optimized.

The Object Browser and COM Form Designer menu commands deal with Component Object Model (COM) features. They allow you to utilize predefined COM objects as well as create forms that host ActiveX controls. For more information see the section titled *COM*.

Messages

The Messages Tab displays a list of common warning messages received in Viva. If you uncheck a message Viva will still perform the check but will not display the warning message. Warning messages as well as information, error, or user created messages are displayed in the Messages window at the bottom of the Viva interface. User created messages are created by designing an output horn within your design and assigning a trap attribute. The trap attribute defines the message that will output.

Creating a Sample Project

This project provides specific examples that correspond with the basic tasks used when creating applications in Viva. The program you will create is an application that squares a variable-input number.

Loading CoreLib

1. Launch Viva.
2. From the **Library** menu, select **Open Library..**
3. In the **Open** dialog box, navigate to the Viva\VivaSystem directory and open the CoreLib.ipg file.

Note: Viva libraries use the same file extension as Viva sheets and it is possible to load a library as a sheet which will then display in your WIP tree.

4. On the right side of the screen in the object tree on the **Project** tab, note the addition of CoreLib to the Libraries tree.

After loading CoreLib, click the plus sign (+) next to CoreLib to expand the library. Note that many new nodes appear as branches of CoreLib.

Placing objects on a sheet

1. From the **Sheet** menu, select **New Sheet**. A blank sheet titled Sheet_1 in the upper-left corner will appear.
2. On the Primitive Objects branch, locate the Input object.
3. Place your pointer on the Input object, then click and hold the left mouse button while moving the mouse such that the graphic representation of the Input object appears on the sheet.
4. To place the Input object on the sheet, release the mouse button when you have placed the object in the preferred location.
5. Open the **Edit Attributes** dialog box for the Input object. To open the **Edit Attributes** dialog box, right-click the Input object.
6. The default node name is *In*. Change the name of the object to X.
7. Ensure that the specified data set is the default, Variant, then click **OK**.

To create the squaring program for this program, you need a multiplier. To place a multiplier onto the sheet and connect it to the Input object, do the following:

8. In the object tree, locate the Arithmetic branch on the CoreLib branch.

9. Click the plus sign (+) to expand the Arithmetic branch.
10. Drag the Mul (multiplier) object onto the sheet.

Connecting objects with transports

Note: The squares on the sides of the Mul object represent the object's inputs and outputs.

1. Hold your pointer over the Input X purple box. The pointer changes to the cross. Click the mouse button.
2. Move your pointer to the A box on Mul until the pointer changes to a cross.
3. Click the mouse button to create a transport that represents a connection from Input X to Mul input A.

Splitting the transport

To square the number, you also need to connect Input X to Mul input B. This requires that you split the connection from Input X to Mul input A. To add a connection from the transport to Mul input B, do the following:

1. Move your pointer to the B box on Mul until the pointer changes to a cross.
2. Click the mouse button.
3. Place the pointer on the transport that goes from Input X to Mul input A where you prefer to make an additional connection.
4. Click to create a connection from Input X to Mul input B via a split transport.

At this point, your project should look like following figure.

Sample project after splitting the transport



Note: It is not necessary to add ClkG to the program. If not connected, ClkG is automatically connected to the system clock at build time.

Adding a OneShot object and an output

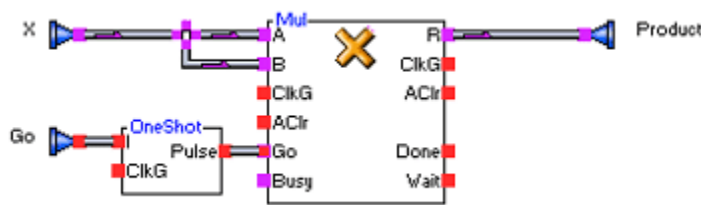
The OneShot object takes a 'Go' input and sends a single 'Go' signal to the multiplier, which causes the multiplier to compute the input and output an answer.

To drag a OneShot object onto the sheet and add an Output object, do the following:

1. In the object tree, locate the Control branch found on the CoreLib branch.
2. Click the plus sign (+) to expand the Control branch.
3. Drag the OneShot object onto the sheet such that you connect the Pulse box on OneShot to the Go box on Mul.
4. Add an Input object to OneShot and name it Go.
5. To complete development of the functional square program, drag an Output object onto the sheet and connect it to output P on Mul.
6. Name the output Product.

At this point, your project should look like the following figure.

Sample project before synthesizing



Synthesizing and testing the program

After completing this program, you must synthesize it. To synthesize the program, from the **Sheet** menu, select **Start/Cancel Synthesis**.

As this program has an undefined data set, the **Choose Data Set** box opens with a list of data sets. To proceed, choose the MSB008 Data Set and click **OK**.


Note: Alternatively, you could have specified the appropriate data set for the X Input by right-clicking on the object to open the Edit Attributes dialog box where you can specify the data set.

After Viva compiles the program, the **Runtime** dialog box opens. The scroll bars in the **Runtime** dialog box are labeled for each input and output.

Experiment with the inputs to test the outcome of your program:

1. To change the value of X, click the right or left scroll arrows until the centered number beneath the scroll bar is the appropriate value.
2. Click the right or left scroll arrow to toggle the value of Go. Click to 0 to start a new calculation; click to 1 to compute the result.

The square of the number specified in the X scroll bar appears on the Product scroll bar (unless you input a value that is too large for the MSB008 Data Set, which, in this case, is 256).

To stop running the program, click  on the Runtime dialog box or, from the **Sheet** menu, select **Play/Stop Sheet**.

Converting the program into an object

You will likely want to convert the squaring machine into an object. To do so, do the following:

1. From the Sheet menu, choose Convert Sheet to Object.
2. In the Object Name field of the Convert Sheet to Object dialog box, type the name Square and click OK.

Note: The newly created Square object becomes a new branch of the Composite Objects branch.

3. From the **Sheet** menu, select **New Sheet**.
4. Drag the Square object onto the new sheet.
5. To test the functionality of this new object, connect up two inputs and any output and choose Start/Cancel Synthesis from the Sheet menu.
6. Test the new object by experimenting with the scroll bars.

Chapter 2: Viva Projects

Viva Projects

When you first start Viva, a new project is automatically loaded for you. A Viva project file saves your project's sheets, programming objects, and systems descriptions. All the work you perform while working within a project is saved with that project so that when you open a previously created project, the project's sheets, objects, and system descriptions are loaded for your use. All of the changes to these elements are saved in the project file you specify when you save a project. Viva project files are saved with the file extension of .idl (short for I2ADL).

Use the Project menu to save a project or retrieve a previously created project.

Note: Viva will prompt you to save changes to unnamed projects and sheets. You can turn off the prompt by clearing the associated confirm close options in Preferences.

Creating a new project

Click the **Project** menu and then click **New Project**.

Viva creates a project with a single, default behavior sheet named Sheet_1, assigns the project the default system description of X86 and loads CoreLib, the design library that a user will use to create their application in Viva.

Note: You cannot have multiple projects open in the same instance of Viva at the same time. To work in multiple projects in Viva, open an additional copy of the Viva application for each project you are working on.

Opening an existing project

1. Click the **Project** menu and then click **Open Project**.

The open file dialog opens.

2. Locate and select the project file you want to open and click **Open**.

Saving a project

1. Click the **Project** menu and then click **Save Project** or **Save Project As...**

If you use **Save Project As...** or if it is the first time you are saving the project the save file dialog will open.

2. Navigate to where you want your project saved, enter the name of your project and click **Save**.

Managing Viva Trees

The following tasks will assist you in easily managing and organizing the WIP and Object trees in the Viva interface:

- To change an object's tree group, drag the object onto another tree group.
- Drag a tree group folder to another location within the objects tree to move all of the objects in it.
- With the Objects Tree frame active, type the first letter an object name to quickly navigate to the object. This only works with objects in expanded tree groupings.
- Click a tree group folder name twice (pausing between clicks) to rename it.
- Click an object name to select it (or use the arrow keys on the keyboard to select an object), then press DELETE to delete the selected object.
- Click a folder name to select it (or use the arrow keys on the keyboard to select a folder), then press DELETE to delete the selected folder and its contents.

Warning: You cannot undo a delete function.

- Drag a folder onto the I2ADL Editor Workspace to create a text object. This feature, which only works on a top-level sheet, causes the Sheet Save option to save all the objects in the specified tree group.

Chapter 3: Viva Libraries

Viva Libraries

Viva is shipped with the Viva CoreLib library. Viva libraries use the same file extension as Viva sheets and can be loaded as a sheet. Any library loaded as a sheet can be modified (it is strongly recommended that you do not make changes to CoreLib). Loading libraries as libraries protects the objects from being overwritten. You may still make changes to the objects in when loaded as a library but you must save any changes as new objects.

Creating Libraries

You can create your own library objects. Objects you create are stored in the object tree as defined in the Tree Group field of the Object Attributes dialog (see Converting Sheets to Objects). As you accumulate your own library of objects you can save that library as a sheet which can later be loaded as a sheet or as a library.

To store a tree group as a sheet:

1. Assemble the objects you want in your library in an object tree grouping.
2. Create a new sheet.
3. Drag the tree group containing your library from the object tree to the new sheet. This will create a text object on the sheet. For example:

`<TreeGroup>ExampleLib`

This text object acts as a place holder for all objects in the tree group.

4. Save the sheet using the library name for your new library.

The sheet can now be loaded as a library.

Library and System Management

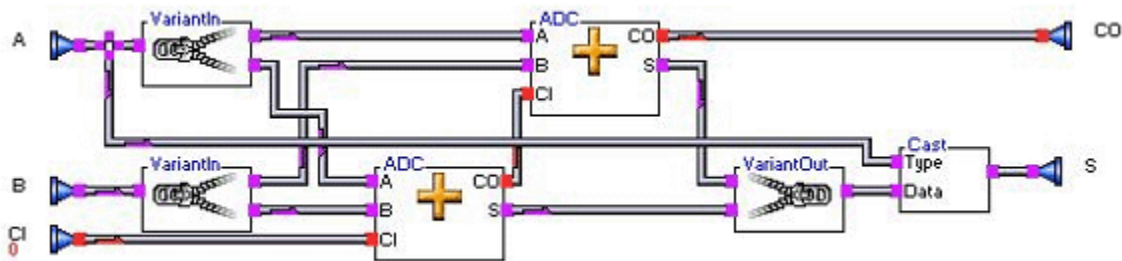
Managing library and system objects in your projects is a critical skill in Viva development because many common objects have different implementations in different systems. If you decide to change the system in your project, say from x86 to an FPGA-based system, you must make sure that your project contains the correct objects.

Ambiguous objects

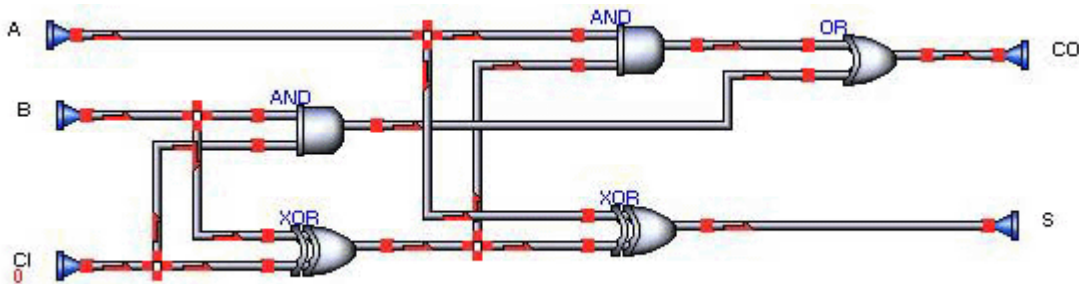
An object's name and number of inputs and outputs define the object's footprint. Most objects have several footprint variants, meaning versions of the object with the same number of inputs and outputs, but where at least one input carries a different data set. When two different footprint variants of an object carry the same data sets on all inputs, they are ambiguous. Viva does not allow ambiguous objects to exist in the same project. The reason for this is that the synthesizer would be forced to make a random decision when resolving objects through Variant Select and thus the object's behavior could never be guaranteed.

To demonstrate the necessity of ambiguous objects, examine the x86 and FPGA implementations of Add with Carry, or full adder (ADC). Both of these objects are contained in the individual System Object Libraries. The Variant version of an x86 ADC demonstrates a classic recursive footprint. There is only one leaf node version of an x86 ADC, the Bit case. Note that it simply defines the behavior of a bit-level ADC.

X86 Variant ADC

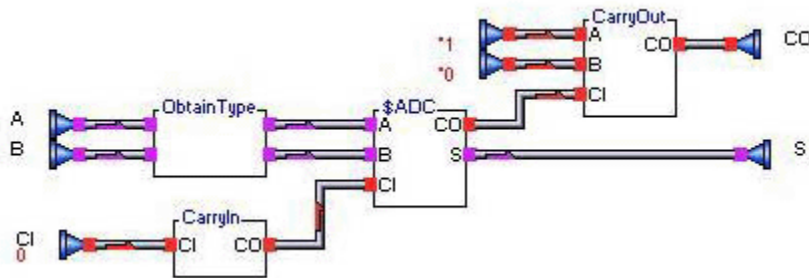


x86 Bit-Level ADC

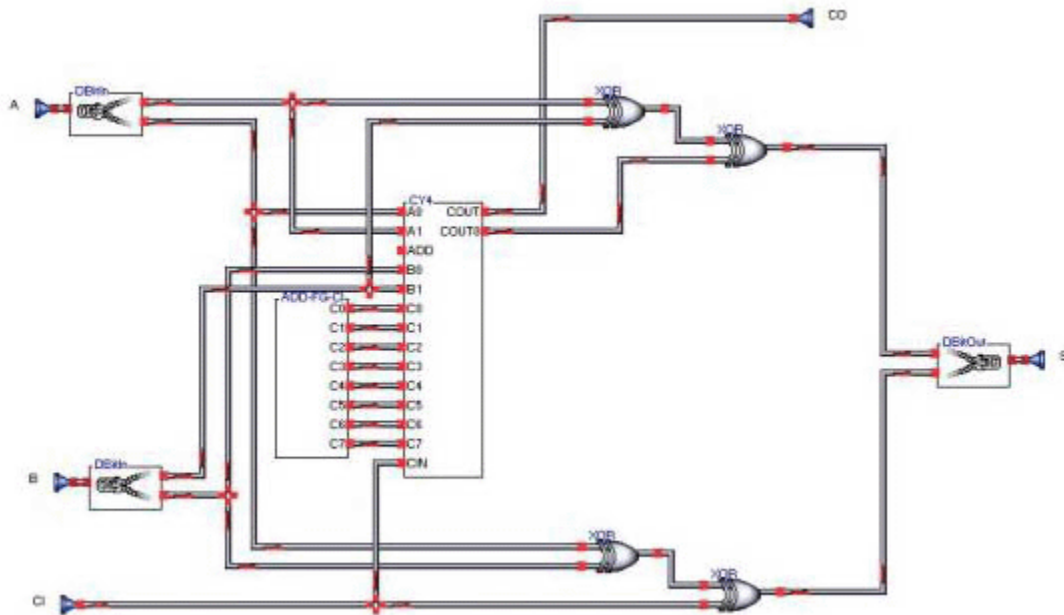


In the FPGA implementation, ADC is not itself recursive, but uses a helper object (think of a recursive helper function in Scheme). The helper object, \$ADC displays the classic recursive footprint. The ADC object itself contains two objects, CarryIn and CarryOut, which respectively start and end the fast-carry propagation chain. This is why the FPGA-specific implementation of ADC is necessary. The CLBs in the FPGAs contain special fast-carry propagation routing resources that you must use for the adders to perform. Technically, the x86 implementation will work, but the carry propagation will use normal routing resources, resulting in an unacceptable number of layers of logic. The \$ADC object has two leaf node versions, a DBit and a single Bit. Each of these leaf nodes represent exactly one CLB, since a CLB is capable of performing a one or two-bit addition. It is necessary to have the single bit implementation to accommodate all possible data sets, but the DBit version represents a more optimal use of resources.

ADC



\$ADC



Fortunately, ambiguous object management is very easy if you understand the basic concept. Since Viva does not allow ambiguous objects to reside in the same project, Viva performs an ambiguous object removal when a sheet or system description containing an object that is ambiguous with one that already exists in the project is opened. You simply need to know how to manage this object removal to effortlessly move between system implementations. Ambiguous object removal is controlled via a user preference located in the Load tab of the Preferences window.

In the **When ambiguous objects are loaded** area, there are three choices, Retain Existing, Replace Existing and Prompt:

- Retain Existing: This option will discard all ambiguous objects in the sheet or system file that is being loaded.
- Replace Existing: This option will discard all ambiguous objects already in the project in favor of those in the file that is being loaded. Use this option for effortless system

switching. This option should also be used when updating libraries, since new objects need to replace their ambiguous predecessors.

- Prompt: This option allows you to choose whether to retain or replace each ambiguous object. This option is often used in library development.

Most users should generally use the Replace Existing option. When the Prompt preference is selected, Viva will present a dialog box with message# 4044 to allow you to choose between the objects. If you check the **Disable this message in the future** box, Viva will apply your choice on this object to all subsequent objects. Therefore, if you choose to replace the existing object and disable the error message, Viva will behave as though you had selected the Replace Existing preference.

Overloading and Underloading

Overloading in Viva is similar in concept to overloading functions in C++. Functions of the same name in C++ are selected by a combination of their name and their parameter types. In Viva, objects with the same name and number of inputs (nodes) are overloaded by changing the input data sets. The object resolver will match the least variant object when deciding which object to build for any given instance.

Underloading in Viva refers to the replacement of objects when you merge two files. For example, when you load a system description, you will want to replace any existing objects of the same name and node count. This is done because many systems vary on their implementation of low-level objects such as registers and adders. Again, you will want to replace such objects when you load a system because the primitives that implement those objects in various hardware and software platforms are fundamentally different. Specifically, if you load CoreLib first and then load the system description, you likely wanted to replace all the CoreLib objects with the system equivalents. If you did not replace ADC, your adders will not work correctly in the hardware. All of the objects overloaded and underloaded by systems are typically instanced in the system descriptions in a sheet called Objects. The objective is to maintain top-level objects that all have the same interface and only replace the low-level system objects, hence, the term underloading.

Updating Library Versions

Note: Star Bridge strongly suggests that you read the following information before updating to a new version of CoreLib.

If your project is working the way you want it to, there is likely not a need to update the library. In the case that you do need to update, use these precautions:

- Do not store your own user objects in the CoreLib tree because you will need to delete the existing tree each time you want to update.
- Avoid using objects from CoreLib that begin with \$ in your project sheets. There is no guarantee such objects will remain in future versions of the library.
- Set your preferences to replace existing before importing the new library.
- Ensure that you replace all objects. That is, do not try to use objects from different versions of a library.
- Remove the previous CoreLib tree group before or after you import. Do not mix objects from an old version with objects from a new version of CoreLib.

If you want to update to a new version of CoreLib, complete the following steps:

1. Back up original project file.
2. Click **View**, then **Show \$Tree Groups** to make sure the \$ tree groups are observable.
3. Delete the old CoreLib sheet.
4. Delete the old CoreLib tree group.
5. Open the new CoreLib sheet.

Sharing designs and libraries among multiple users

Libraries, sheets, projects, and system descriptions cannot be written to simultaneously. To save a file, employ a mechanism to ensure that no one else has changed the file since you read it into Viva.

Chapter 4: Viva Sheets

Viva Sheets

When you start Viva or create a new project, Viva loads a sheet named Corelib into the new project. This sheet contains all objects and data sets associated with Viva library. Use the objects in Corelib to create new sheets, objects and applications. Each sheet loaded into Viva will carry with it all objects and data sets associated with that sheet into the Viva project

You can have multiple sheets within a project. When you create a new sheet, the newly loaded sheet will be added to the project and will become the active sheet in the I2ADL Editor. The project, and sheets in the project, is displayed as a tree (the Work in Progress or WIP tree) located in the upper right frame of Viva.

Viva application worksheets are saved with a file extension of .ipg.

Creating a new sheet

Click the **Sheet** menu and then click **New Sheet**.

A new blank sheet is created and added to the WIP tree.

Opening an existing sheet

1. Click the **Sheet** menu and then click **Open Sheet**.

The Open file dialog opens.

2. Locate and select the sheet file you want to open and click **Open**.

Note: The sheet menu will display a list of the most recently opened/saved sheet files. This is referred to as the sheet history. Clicking on a sheet file in the list opens that file, as if you had opened the sheet using the Open Sheet menu option. You can control the number (0–15) of recently opened sheets displayed on the Files tab of the Preferences dialog; located on the Tools menu.

Saving a sheet

1. Click the **Sheet** menu and then click **Save Sheet** or **Save Sheet As...**

If you use **Save Sheet As...** or if it is the first time you are saving the sheet the save file dialog will open.

2. Navigate to where you want your sheet saved, enter the name of your sheet and click **Save**.

Renaming Sheets


1. From the WIP tree, click a sheet name twice (pausing between clicks).
2. Type the new name of the sheet and push **Enter**.

Converting Sheets to Objects

Once you have created a design on a sheet, you can then convert the sheet into a Viva object. Inputs and outputs of the created object will be of the same order and type as the defined application. For example, if you have 2 inputs, A and B, and a single output, Q, all of which are bit data types, the resulting Viva object has the same inputs and outputs as bit data types.

To convert a sheet to an object:

You can convert sheets into objects using the following procedures:

- Click the Convert Sheet to Object button () from the tool bar to convert the sheet you currently have open.
- From the WIP tree, double-click the sheet you want to convert.
- Drag the sheet you want to convert to the Object Tree frame.

The Convert Sheet to Object dialog is opened. Before saving the object, enter information about the object you have created in the Convert Sheet to Object dialog box. Each of the fields and sections of the Convert Sheet to Object dialog box are explained in the following table.

Convert Sheet to Object dialog box description

Dialog box field/section	Description
Object Name	The name of the object. Name is case sensitive.
Tree Group	The intended object group for this object after conversion. You can select one of the existing groups from the drop-down list or create your own. Create structure (levels) in the tree group by including “\” characters.
Create New Object	Select this radio button if you are creating a new object. Auto-selected for new sheets.
Update Original Object	Select this radio button if you intend to update the original object. Auto-selected if name is changed on existing object.
Leave Sheet Open	By default, Viva closes a sheet after it is converted into an object. Checking this box will keep the sheet open after conversion for additional editing.
Attributes	This section displays a description of all of the sheet’s inputs, outputs, specific attributes, and data types.
Documentation	Use this section to describe the object’s function and how it should be used. This is a free text field.



and how it should be used. This is a free text field. However, some consistency in description format is beneficial.

Icon	Create/edit object icons in your default Windows editor.
Revert	If your object has the same name as one of the .BMP files in the Viva\VivaSystem\Icons folder, it will display that icon by default. Discard renames the .BMP file, causing the icon to be removed from the object. The Revert button will reinstate the .BMP file name.
Discard	Discard renames the .BMP file, causing the icon to be removed from the object.
Refresh	Refresh enables Viva to recognize when an icon is edited. Refresh causes Viva to get updated bitmap files. Viva refreshes automatically when the form is closed.
OK/Cancel	Standard buttons to complete or cancel the intended action. In this case, it is the conversion of a sheet into a Viva object. Cancel will not discard icon changes because changes take effect immediately.

Navigating Sheets

You can navigate through the sheets in your project by clicking the sheet you want to view from the WIP tree. Other ways to quickly navigate sheets in your project are as follows:

Moving from sheet to sheet

You can move forwards and backwards among previously viewed sheets by using the **Back** () and **Next** () commands in the Sheet menu.

Descending into sheets

Viva provides you with the capability to "drill down" into the details of a Viva composite object, essentially giving you an detailed view of the components that make up the object.

To view the details of an object (or descend into a sheet), select the object on your sheet and click the **View** menu and then click **Descend Into Sheet**.

You can also descend into a sheet by double-clicking the object.






When viewing the detailed view of an object, a sheet is added to the project sheet list. This sheet has the same name as the object you are viewing and contains the components of the object.

Note: This command has no effect on primitive objects because such objects have no detail to display. The command is also ineffective if more than one composite object is selected.


Viewing Sheets

You view sheets in Viva by opening them in the I2ADL Editor. Once you have a sheet open there are several tools you can use for viewing sheets.

Zoom

Viva provides pre-set zooming capabilities on sheets open in the I2ADL Editor. This gives you the ability to zoom in () , zoom out () , return to the default zoom () , fit the viewing window to the size of your design () , or to fit the viewing window to the width of your design () . You can access the zoom functionality through the View menu or by using the zoom buttons on the toolbar.

View Sheet in Bitmap Editor

You can capture your sheet and open it in your default bitmap editor. To do so, open the sheet in the I2ADL Editor and click the **View Image in Bitmap Editor** button () .

Chapter 5: Viva Objects

Viva Objects

A Viva object is similar to an object in C++, Java and other object oriented programming (OOP) languages. The primary difference between a Viva object and an OOP-language object is that Viva objects are defined more in terms of decision paths, data flow, and parallelism. An object is responsible for performing some sort of data manipulation, decision making, or to facilitate efficient data flow. Viva objects include input parameters and output values. For more information on Viva objects, the function of the individual objects, see the *Viva Object Reference Guide*.

Adding objects

To drag and drop an object into the application workspace, point to an object in the object tree, then click and drag the object over the workspace and release the mouse button.

If an object is dropped onto the sheet in such a way that one or more of its input or output nodes fall within node snap tolerance range of another node, Viva will automatically connect the nodes if data set types are compatible and if the node functions are appropriate (Input-to-Output or Output-to-Input). If nodes are not connected automatically, you must connect them through the use of transports.

Selecting objects

Left-click an existing object to make it the active object. The color changes to red when activated.

You can also select objects by creating a box. Create a box by clicking and holding the mouse button, then dragging it to another location. This process creates a box, the contents of which are selected.

Use the following options to control selecting and deselecting objects:

- To copy a selected object (or objects), press CTRL+C or CTRL+INSERT.
- To cut a selected object (or objects), press CTRL+X or SHIFT+DELETE.
- To paste a selected object (or objects), press CTRL+V or SHIFT+INSERT.
- To undo cut, copy, paste, and delete operations, select Undo from the Edit menu, click the Undo button on the toolbar, or press CTRL-Z on your keyboard.
- To delete a selected object (or objects), press DELETE.
- CTRL+ box select to toggle the selection of all objects in the box. Objects that were selected are deselected; those that were not selected are selected. The box selects/deselects multiple objects at a time.
- SHIFT+ box select to select additional objects.
- To deselect all objects, click any blank location on the workspace.

- If you want to select only one object click the specified object; all other objects are deselected.
- To select everything on the workspace, press CTRL + A.
- To toggle selection of every object on the workspace, press CTRL + SHIFT + A.

Editing object attributes

To edit an object's attributes, right-click the object to open the **Edit Attributes** dialog box. Double-click an object to show its behavior in a new sheet.

Press ALT while clicking a transport or junction to select or deselect transports and junctions associated with the specified transport or junction. To select or deselect another connection without deselecting other selections, press SHIFT and ALT while clicking a transport or junction.

Moving and copying objects

To move one or more objects, select and drag the object(s) to the preferred location. This option provides freedom to move the selected object horizontally and vertically. Any transport(s) connected to the selected object remain connected and adjust to the selected object's new location.

Use the following options to move objects:

- To restrict object movement to a direction that will not require transports to be broken, press SHIFT while dragging the selected object(s).
- To copy one or more objects and paste them in a different location from the original, press CTRL while dragging the object(s).
- To disconnect and move a selected object or objects, click and hold the mouse button on the selected object(s), then press CTRL and ALT while dragging.

Connecting objects

Transports are links that connect input and output nodes to show data flow. To draw a transport, click an input or output node. The transport attaches from the node to the mouse pointer. Moving the mouse around the node will cycle the transport through 90-degree turns. Click in the Viva workspace to designate an anchor point for the transport and continue dragging and extending the transport. Clicking within node snap tolerance of a node will connect the free end of the transport to the node if connecting nodes have compatible data types and node functions. A transport can also be terminated at the last fixed location by right-clicking in the application workspace.

Transports follow these rules:

- An output node can only connect to an input node.
- An input node can only connect to an output node.

- Only like data types will connect.

To delete a transport, select the transport, then press DELETE.

Double-clicking a transport creates a junction at the clicked location of the transport. A junction consists of four nodes. One node is an input node and the remaining three are output nodes.

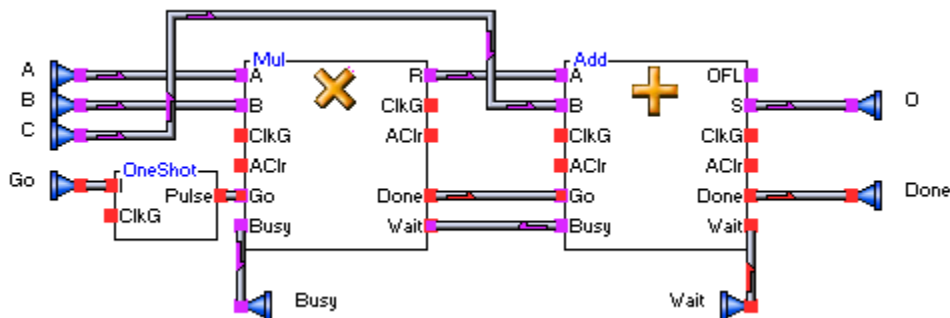
Once a connection is established, double-click at any location on the transport to add a junction. You can also create junctions at a corner of a transport.

If a junction is deleted that connects two transports from opposite sides (an input node and an output node), then the two transports are automatically connected; that is, the junction is removed and the transports merge, or connect. To leave the transports disconnected, press CTRL while deleting the junction.

Object Footprints

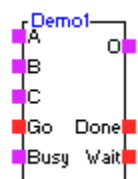
The “footprint” of an object is made up of the object name (case sensitive), the number of inputs and outputs the object contains and their individual names. As an example, the following diagram illustrates a Viva application that contains an object with 3 data inputs, labeled ‘A’, ‘B’, and ‘C’, and a single output value labeled ‘O’. There are also nodes labeled ‘Go’, ‘Busy’, ‘Done’, and ‘Wait’. Go, Done, Busy, Wait nodes have special functionality to assist with timing control (see [Go Done Busy Wait](#)).

Object footprint - example 1



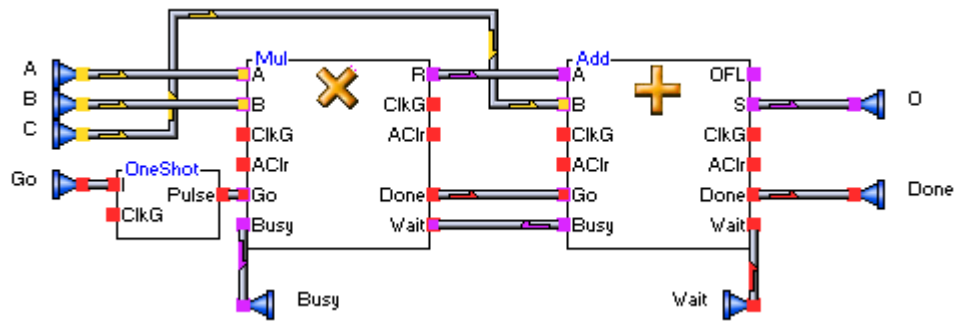
Object footprint - result of example 1

The preceding sheet multiplies the input values of A and B then adds the product to the input value of C. When this sheet is converted to an object, using the name Demo1, the subsequent object has the following footprint:



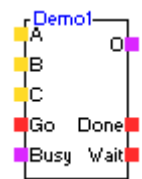
Each data input for the object in the Demo figure became an input node on the left side of the Demo1 object. Each data output on the sheet became an output node on the right side of the Demo1 object. Node data types are maintained from the application sheet to the object footprint upon conversion. If the application in the Demo figure specifically worked with the data type of byte, the object would look similar to the one in the Demo figure, but the nodes would indicate a different type by displaying a different color. This is shown in the following figure.

Object footprint – example 2



The object performs that same function as before and the footprint remains the same. The only difference is the data type being used for the inputs and outputs.

Object footprint - result of example 2



Object Polymorphism

Polymorphism is the ability for an object to handle various data set types in a predefined behavioral design. Polymorphism occurs through a combination of overloading, recursion, and compile-time resolution.

Overloading

Overloading means having different objects with the same footprint handle different behavioral characteristics required by different data set types

An example of overloading would be a 'Convert' object that has the ability to convert an unsigned integer to a signed integer. Then, use a similarly named object, 'Convert', to convert a fixed-point number to a signed integer.

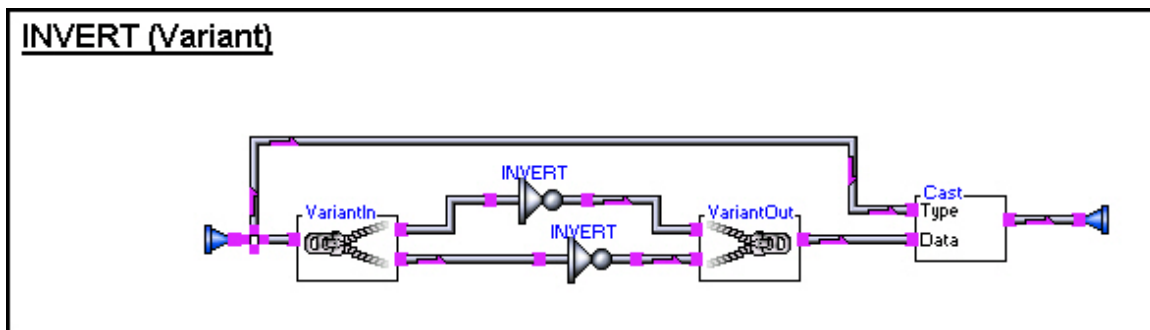
When you create overloaded objects, they should normally be grouped together in the same tree group in the object tree. Viva resolves an overloaded object appropriate to the data sets at compile time. Using this technique, you can create applications that will work on new data sets without having to modify existing objects.

Recursion

Recursion is the ability of a function, or an object, to call itself; to be self-referential. The use of recursion in Viva makes it possible for a single predefined behavioral design to define how to perform addition for all unsigned integers, or all signed integers, and so on.

An object that is depicted with input and output nodes is associated with a series of overloaded functions. By definition, these overloaded functions have the same name yet handle the different implementations of that particular function for different input data set types. Composite objects with variant input types can be recursive as long as the referenced objects operate on simpler data types than the original data type. This rule allows for the creation of recursive decompositions until the object is broken down into an operation on a more fundamental data type. The following table illustrates the hierarchical order of the standard static Viva data sets.

Variant invert object example



Viva data sets

Order	Viva data set	# of Bits	Data set description
Low	Bit	1	The lowest order and most basic type.
	Dbit	2	2-bit cardinal data type
	Nibble	4	4-bit cardinal data type
	Byte	8	8-bit cardinal data type
	Word	16	16-bit cardinal data type
	Int	16	16-bit signed integer
	Fix16	16	16-bit fixed point
	Dword	32	32-bit cardinal data type
	Dint	32	32-bit signed integer
	Fix32	32	32-bit fixed point
	Float	32	32-bit IEEE floating point
	Qword	64	64-bit cardinal data type
High	Double	64	64-bit IEEE floating point

Compile time resolution

Data set polymorphism refers to the ability in Viva to create an appropriate implementation at compile time depending on the data sets. That is, Viva can wait until compile time to determine the precise implementation of your application by resolving variant data sets to specific data sets on the inputs of each object.

Differences between the X86 system and hardware systems

- The X86 system (X86.sd) is composed of four subsystems: (1) X86CPU, (2) X86UI, (3) UIXCPU, and (4) X86ClkImport.
- X86CPU represents the majority of the system. It contains all the objects that the emulator recognizes. The X86UI system comprises the Runtime dialog. The UIXCPU system is the communication between X86CPU and X86UI. X86ClkImport is a communication system from the emulator macro \$Clock to the global signal ClkG. (Most systems have some way of tying the global clock signal to some local resource.) Hardware systems are similar: there is an FPGA system containing all the supported objects and the external tools parameters, a communication system to the host system, and a communication system to the global clock signal.
- System descriptions maintain the idea of an executor. That is, either the system is made to dissolve into other systems, or the system will have a planned execution

environment such as the X86Main thread or an FPGA. See the section titled *Viva Systems* for further information on executors.

Object Attributes

Each Viva object possesses a variable number of attributes. Right-click an object on a sheet to access the attributes dialog box for that object. The attributes on the master objects can be changed by right-clicking on the object in the object tree and selecting **Edit Object Attributes** on the resulting menu. Different objects have different allowable attributes. After specifying attributes for an object, you can move your mouse over the object and a tool-tip will report associated attributes for that object.

The following table provides the definition for each of the attributes found in the drop-down list found on the Edit Attributes dialog. Each Viva object possesses a variable number of attributes that direct its use. The following is a list of possible object attributes:

Viva object attributes

Attribute	Notes
ChildAttribute = "AttName=AttValue", "AttName=AttValue"	<p>Only valid on composite objects.</p> <p>The double quotes in the attribute value are required. The child attribute (AttName=AttValue) is propagated to all descendent objects that do not already have a child attribute. The System attribute does not need to be a child attribute because it is also propagated to all descendent objects. Transports, junctions, exposers and collectors do not receive attributes from their parents.</p> <p>ChildAttribute allows multiple attributes to be passed down to descendant objects. The list of attributes is entered on a single line in the Attribute Editor. Each "Attribute=Value" pair is entered in quotation marks and may be optionally separated by ",". For example:</p> <p>ChildAttribute="System=X86UI", "Resource=TIMESLICE", "Widget=Button"</p>
Constant = ConstantValue *ConstantValue	<p>Only valid on Input horns.</p> <p>Under ordinary circumstances, Input horns become input nodes when a behavior sheet is converted into a composite object. By giving the Input object a Constant attribute, the corresponding input node will be set to the ConstantValue whenever it is left unconnected. In this case, the Input object is given a default value which the user has the option of overloading. If the input node is connected, then the Constant attribute will be ignored.</p> <p>Prefixing a ConstantValue with an asterisk ("*") prevents it from being overloaded. This causes the Input object to not become an input node when the behavior sheet is converted into a composite object. This ensures the Input object always receives the ConstantValue. It cannot be overloaded because it does not appear on the composite object.</p> <p>Whenever constants are used, Viva simplifies the circuitry by propagating the constants and removing unnecessary logic. For example, when a constant 0 is fed into an OR gate, the result is always the value of the other input. The OR gate and the constant</p>

input are removed. When a constant 1 is fed into an OR gate, the result is always 1. The OR gate and the other input are removed.

Constant Propagation Rules

"0" represents a constant zero.

"1" represents a constant one.

"?" represents a random input value (constant or not).

If a primitive bit object has the indicated input value(s), then the object is removed and the indicated output value is propagated down stream.

A constant zero on a Bit Output no longer causes the widget to be removed.

INVERT	
In1	Out1
0	1
1	0

XORCY		
LI	CI	O
0	?	CI
?	0	LI
1	?	Invert(CI)
?	1	Invert(LI)

BUFT_V2		
T	I	O
0	?	I
1	?	Disconnect

\$Select			
#0	#1	S	Out1
?	?	0	#0
?	?	1	#1

MUXCY			
DI	CI	S	O
?	?	0	DI
?	?	1	CI

FDCE				
C	CE	CLR	D	Q
0	?	?	?	0
1	?	?	?	0

?	0	?	?	0
?	?	1	?	0
?	?	?	0	0
?	1	0	1	1

Output Trap

In	
0	Remove widget
1	Remove widget after displaying message

The ConstantValue is displayed in red on the left side of the Input object below the node name. The ConstantValue must be compatible with the Input object's data set. Viva supports the following types of constants:

Numeric Constants: Numbers may be entered in standard numerical format (digits, decimal point, minus sign). Hexadecimal values may be entered using the standard "0x" prefix.

String Constants: String literals may be entered on string data sets.

Random Constants: A ConstantValue of "?" causes Viva to generate a random constant each time the object is compiled. The random number will be uniform across the possible range of the data type.

File Constants: Constant data may be read out of a data file that is in ".ini" file format. The data file must contain a "romdata" section and the entries must have a "D" prefix. Example:

```
[romdata]
```

```
D0=12
```

```
D1=11
```

```
D2=5
```

ConstantValue must be a fully qualified file name or directory prefix code inside two sets of double quotes. (See the EdifFile attribute for more information.) The object must also have an Index attribute. See SampleFileConstants.idl in the Viva\Examples directory.

Macro Constants: Additional constants can be calculated using Viva macros. They have the form "##N9.X". The double quotes, pound signs and "N" are required. Replace "9" with the input node number from the wrapped up object the macro is to be performed on. Replace "X" with one of the following macro function codes:

"A"—Data Set Number; only guaranteed to not change on the following data sets:

	<p>0 BIT</p> <p>1 VARIANT</p> <p>2 VECTOR</p> <p>3 NULL</p> <p>"B"—Data Set Bit Length (65,535 means variant)</p> <p>"H"—Child Data Set Count</p> <p>"T"—Data set Attribute number:</p> <p>1 Unsigned Integer</p> <p>2 Signed Integer</p> <p>Fixed Point</p> <p>8 Floating Point</p> <p>Complex</p> <p>See SampleFileConstants.idl in the Viva\Examples directory.</p>
Documentation = Description	Description is displayed behind the object's name in the project object tree, in the sheet header, and object track information. Very useful in distinguishing between objects with the same footprint.
EdifFile = FileName	<p>Only valid on primitive objects with an FPGA system attribute.</p> <p>The Translator will insert all of the requested FileNames into the FPGAStrings file (before the last two lines of the FPGAStrings file) when the Xilinx ".edn" file is created. FileName may be fully qualified (starts with a drive letter or "\\") or relative to a standard Viva directory by using one of the following FileName prefixes:</p> <p>" " \Build Directory Attribute</p> <p>"\$V\" \Viva</p> <p>"\$S\" \Viva\VivaSystem</p> <p>"\$P\" Current project's directory</p> <p>"\$U\" User directory</p> <p>"\$B\" Build directory</p> <p>"\$F\" FPGA file name</p>
Exclusive = True False	<p>Only valid on \$Spawn object.</p> <p>If True, then the parent Viva program will wait until the spawned child program is complete before continuing.</p> <p>Otherwise, the parent Viva program will run concurrently with the spawned child program.</p>
FileName = FullFileName	<p>Only valid on \$Spawn and objects.</p> <p>FullFileName is the fully qualified directory prefix or disk file name to be used on the object. For \$Spawn objects it is the Viva program to be run as a child spawn process. For Output objects it is the file to record all of the values sent to the Output object.</p>
FPGAProperty =	Only valid on primitive objects with a FPGA system attribute.

String	All object attributes that start with "FPGA" are added into the EDIF net list file as Xilinx cell "properties". The "FPGA" prefix is stripped off. String is included in the Xilinx cell property as the "string" parameter. A String of "<NULL>" is replaced by an empty string.
Icon = FileName	<p>Only valid on non exposer/collector objects.</p> <p>Overrides the object name as the name of the bitmap file containing the object's icon. Entering this attribute on an object will cause all of the objects with this name to use the same icon. Invalid characters in the object name are replaced by "_" to create a valid Windows file name. FileName does not include the path (Viva\VivaSystem\Icons) or the file extension (".bmp").</p> <p>If the right and/or left edge of the bitmap file is not transparent, then transports will be drawn when the icon needs to be made wider to cover an object.</p> <p>Viva rounds the pixel width of object icons to the closest multiple of 5, plus 2. User-generated icons with a different width may encounter an omitted left or right column of pixels. Therefore, only create object icons with a pixel width that is a multiple of 5 plus 0, 1, or 2.</p>
Index = IndexValue	<p>Only valid on Input objects that have a Constant attribute that contains a file name.</p> <p>Constant data may be read out of a data file that is in ".ini" file format. IndexValue must resolve to a data ("D" prefix) entry in the "romdata" section of the data file.</p> <p>See SampleFileConstants.idl in the Viva\Examples directory.</p>
InstanceName	Allows objects to have a specific instance name in the EDIF Viva generates. The attribute is available on primitive objects having a LibRef attribute. For example, if you place an "InstanceName=MyName" on a system primitive object you will see "(instance MyName" in the output EDIF file as opposed to the default auto-assigned N10000+. Be careful to not put the same name on multiple instances of objects.
LibRef = CellRef	<p>Only valid on primitive objects with a FPGA system attribute.</p> <p>CellRef is the EDIF net list "cellRef" name for the primitive FPGA object.</p>
ObjectRef = VectorName	<p>Only valid on Composite objects.</p> <p>Creates a reference to the object's I/O connections in the form of two Vector global labels. VectorName must be a unique identifier (normally an integer) for the object. All of the object's Input/Output nodes are wrapped up into a Vector global label named SheetName.ObjectName + VectorName.In/Out. The Vector global label names are displayed below the object in the I2adl Editor in fuchsia.</p>
PadLoc = PadLocation	<p>Only valid on primitive objects with a FPGA system attribute.</p> <p>PadLocation is the Xilinx pad location that the object will use. It is the "string" parameter of the "LOC" property in the cell definition in</p>

	the EDIF net list file.
PropertyName = FileName	<p>Only valid on Input and Output objects.</p> <p>Same as the Icon attribute except (1) it allows a single Input/Output to have a special icon without giving it to every Input/Output and (2) the bit map file name has a "P_" prefix.</p>
ProvidesResource = ResName	See the Resource attribute.
ResourceQty = 999	<p>Only valid on primitive objects with a System attribute and Input/Output objects.</p> <p>Overrides '1' as the default number of resources required each time an object is instantiated. This is also called the "Reaction Latency" for an object. Because the Resource Editor handles complex resource combining, the resource quantity must be a simple integer value.</p>
RouteEffort<Min, Max, Step>	<p>The RouteEffort<Min, Max, Step> attributes limit how long router connection paths may be. Because FPGAs have so many connections, there are millions of possible communication paths. The RouteEffort<Min, Max, Step> attributes enable shortest paths to be used. The attributes also allow objects to be routed through close systems first (even if a longer connection path has a lower cost). This allows all of the connections that can be made in a single step to be routed first, which reduces the number of multiple-step connections required.</p> <p>RouteEffortMax - Indicates the largest possible connection path limit to be tried. The minimum value is RouteEffortMin (must at least try once). Default value is 50.</p> <p>RouteEffortMin - Establishes a connection path limit for the first object routing pass. The minimum value is 1. The default value is 50.</p> <p>RouteEffortStep - Identifies an incremental step value to increase the route effort by each time a new object routing pass is started. The minimum value is 1. The default value is 2.</p>
System = SysName ..\SysName	<p>SysName is the name of the specific system the object should be implemented in. The default is to the DefaultTargetSystem attribute in the base system. Set SysName to "?" to ignore the DefaultTargetSystem and have the Search Engine Router calculate the lowest cost system. Placing this attribute on a composite object will propagate it to all descendent objects that do not already have a System attribute. If the requested system cannot implement the object then it will be implemented in another system. Giving an I2adl Def a system attribute will move it from the project object tree to the system object tree.</p> <p>The "..\" prefix is only valid on the Input/Output objects on communication system composite transport sheets. It finds the system one level up in the system tree from the composite Transport's system. Enter "..\" to go up two levels in the system tree.</p>

<p>TIMESPEC =</p> <p>TimeProp=TimeString</p>	<p>Only valid on primitive objects with a FPGA System attribute.</p> <p>Allows the user to add Xilinx timing information into the EDIF net list file for the object. The TimeProp is included as a Xilinx cell "property" with the TimeString as its "string" parameter.</p> <p>TIMESPEC=TSCLKG=PERIOD:CLKG:10 is an example of a valid TIMESPEC attribute.</p>
<p>TNM =</p> <p>TimeString</p>	<p>Only valid on primitive objects with a FPGA System attribute.</p> <p>Giving an object a TNM attribute causes all of the transports connected to its output nodes to receive a Xilinx TNM cell "property" with the TimeString as the "string" parameter.</p>
<p>Trap =</p> <p>999</p> <p>ErrorMessage</p> <p>999 ErrorMessage</p>	<p>Only valid on Output objects.</p> <p>Error traps enable the user to detect and report error conditions. The user sets a trap by creating an output object with a Trap attribute. Whenever the input to the trap is non-zero, a message window is displayed. This can occur at compile time (by using constants) or at run time. The text associated with the trap is either a standard error message (999) from the Viva\VivaSystem\ErrorMessage.txt file, a user created error message (ErrorMessage), or both. The input data sets at the time the parent object was expanded are displayed in the error message.</p> <p>The first thirty characters of the error message of an output trap appear on the side of the output horn below the node name in green, the same color used to display User Trap messages in the message window.</p>
<p>WarnOnRemoval =</p> <p>True</p> <p>False</p>	<p>Viva warns the user when a global label input is removed because no matching global label output was found. WarnOnRemoval enables users to disable this feature; the default is "True" and "False" disables the warning.</p> <p>Because there can be multiple input objects with the same global label name, the warning will appear if any input object requests it.</p>
<p>Widget =</p> <p>WidgetType</p>	<p>Only valid on Input/Output objects in a X86UI type system or no system.</p> <p>Overrides the DefaultInputWidget and DefaultOutputWidget attributes from the base system as the widget type.</p> <p>Input objects can have a WidgetType of ScrollBar, TextBox, Button, or SpinEdit.</p> <p>Output objects can have a WidgetType of ScrollBar, TextBox, Memo, Button, or Graph.</p> <p>When the widget type is changed on the widget form, then the source Input/Output object is updated.</p>
<p>WidgetHeight =</p> <p>WidgetLeft =</p> <p>WidgetTop =</p>	<p>Only valid on input/output objects that have a Widget attribute.</p> <p>These attributes override the default Height, Left, Top and Width values that control the size and position of input/output widgets on the widget form.</p>

WidgetWidth = 999	When the widget size or position is changed on the widget form, then source Input/Output object is updated.
WidgetHex = True False	<p>Only valid on input/output objects that have a Widget attribute that is ScrollBar or TextBox.</p> <p>Determines if the numerical data should be displayed in decimal or hexadecimal notation.</p> <p>When the hex display type is toggled on the widget form, then the source Input/Output object is updated.</p>
WidgetMax = WidgetMin = 999	<p>Only valid on input/output objects that have a Widget attribute that is ScrollBar, SpinEdit or Graph.</p> <p>Used to limit the range of the data handled by the Input/Output widget to something less than the full range of the data set.</p> <p>When the widget min/max values are edited on the widget form, then the source Input/Output object is updated.</p>

Object Attributes with Special Meanings

Some object attributes have special meaning at load time, rather than existing as standard object attributes. Their names, valid values, and meanings are listed in the table below.

Object attributes with special meanings

Name	Meaning
WIP Sheet	Identifies WIP sheets saved in the file. Otherwise they would appear as object definitions.
Primitive	Marks the applicable object as primitive (not composite).
CurrentSheet	This behavior will be the WIP sheet that is displayed when the project is opened.
CompileSheet	This is the compiled behavior for this project.
Link	Used by Viva when writing compiled behaviors. Do not edit.
KeepObject	Sets the Keep Object field, seen in the Attribute Editor. Values are connected on "Both sides", "One side", and "Always". You might have an object that has no connected inputs, or you might have an object that has no connected outputs. If such is the case, the compiler removes the sourceless/syncless object unless this attribute directs otherwise.
ProgID	<p>Directive to set this object as a member (function, get/set property, or event dispatcher) for the COM class or static instance, instance generator, or dispatch identifier for a global instance, specified by the value of this attribute. This takes one of 4 forms:</p> <ul style="list-style-type: none"> ▪ Member of static - Name of instance (as defined in a ComObject) + "." + name of member instance. For example, ProgID=Chartfx1.Type ▪ Member of class - Type Library ID + "::" + Name of class + "::" + name of member. For example, ProgID=VIVASTANDARDOBJECTSLib::VivaString::LeftString ▪ COM instance - Type Library ID + "::" + Name of class + "::" + "[Create]" generator. For example, ProgID=VIVASTANDARDOBJECTSLib::VivaString::[Create] ▪ Dispatch identifier - Name of instance + "'s Dispatch" (avoid editing; consider read-only). For


example, ProgID=VivaString1's Dispatch

ChildAttribute

Because the quotation mark character is used to terminate attributes in text base, the child attributes are saved separately; they are combined into one attribute when the file is read back into Viva. Text-based example:

```
//_Attributes ChildAttribute="System=X86UI",  
ChildAttribute="Resource=TIMESLICE",  
ChildAttribute="Widget=Button"
```

Viewing Object Detail

Double-click an object name or an object icon () next to an object name for a detailed view of the object. You can also drag an object onto the WIP tree to display the object in the I2ADL Editor Workspace.

Another way to view an object's detail is to select the object in the tree and then press ENTER to display the object in the I2ADL Editor Workspace or use the arrow keys to select an object and then press ENTER. This feature is available in the I2ADL Editor Workspace and in the System Editor Workspace/Tree.

Note: This is only available for composite objects because primitive objects are not composed of other objects and as such, have no detail.

Chapter 6: System Descriptions

System Descriptions

Viva can be used as the development environment for any hardware platform that can be defined by a system description. A system description defines the physical attributes and resources of a computing platform. Viva system descriptions are a high-level system file containing data for targeting a specific hardware platform. System descriptions provide the bridge between the design you create in Viva and your hardware. The system description contains a number of FPGA primitives, system level primitives, and specialized composite objects to implement the clocking scheme on the FPGA, set up the communication system and communicate to other peripherals on the board. System descriptions also provide access to settings you can configure to adjust the operation of the hardware.

System descriptions use the .sd file extension.

X86 System Description

The X86 system description targets the PC development environment. This system description is used during development to allow for testing the design within the development environment. Synthesizing applications for the X86 system is faster yet Viva still performs design verification and error reporting for quick evaluation of your design. By considering the differences in your target hardware and the X86 system description you can verify your design with the X86 system description, load your target system description and re-synthesize with minimal design changes. See [X86 Versus FPGA Applications](#) for more information.

FPGA Systems

Viva is primarily targeted at Reconfigurable Computing (RC) hardware platforms. RC platforms are typically built using Field Programmable Gate Arrays, or FPGAs. The system description describes the target FPGA-based hardware. These system descriptions represent the resources and structure of the target architecture. Ultimately, all Viva behavior, with the exception of inter-chip transports and transports to the Widget Interface, is mapped into the configurable logic blocks of these FPGAs.

The Viva build process performs a formal recursive synthesis on all objects in an application, during which it expands complex variant objects into gate-level logic. All internal logic, whether it is simple gates, adders, multipliers or objects of higher complexity, is mapped into an FPGA's Configurable Logic Blocks (CLBs). CLBs exist en masse on an FPGA and contain inputs, outputs, function generators, multiplexers and storage elements. The function generators contain programmable look-up tables (LUTs). When the chip is programmed, the LUTs are coded so that their outputs reflect the output of the combinational logic function of the function generators. All of the logic that your Viva program represents will be coded into these LUTs when the FPGA board is programmed.

Optimal speeds are obtainable in FPGAs when application design has as few layers of logic as possible in between CLB storage elements. Most Viva library objects are designed with a single layer of logic where the function generators feed the storage elements within their own CLB.

Each FPGA system has its own set of primitive objects that can be implemented in the FPGA directly. When application behavior is targeted to an FPGA system, the synthesizer recursively expands application behavior and resolves all recursion leaf-nodes in the system object library.

Primitive system objects have no behavior, so you cannot drill into them. To view an FPGA library:

1. Use the Viva menu selection System\Open System to open a system description file.
2. Click the System tab.
3. Drill down to the FPGA level to view the system objects.

Viva Platform Guides

Viva Platform Guides are available for Viva system descriptions. Viva platform guides provide information for programming to target a specific set of hardware. User adjustable hardware settings and system-specific composite objects are documented in the Platform Guide. Also provided are instructions for getting a provided example project running on the target hardware. Contact Starbridge about obtaining the necessary system description file and Platform Guide for the hardware you want to target.

Project Attributes

The Project Attributes dialog allows access to override system attributes. These system attributes are part of the system description file for the target system.

Warning: Many system attributes should not be changed, changing them can cause your application and hardware to function poorly or not at all. Read the Platform Guide for your specific target system provided with your copy of Viva for information about which system attributes you can safely override.

Viewing Project Attributes

To view project attributes for a system or subsystem:

1. Click the System Editor tab.
2. Navigate the system and subsystems in the system tree (located at the right when in the System Editor) to view the system attributes for that system.

Note: The System Editor is read only (contact Starbridge to purchase VivaSD if it is necessary for you to edit or create system descriptions).

Changing Project Attributes

Project attribute changes override the system attribute. Therefore changes to system attributes are made in the Project Attributes dialog.

To create a project attribute system override:

1. In the System Editor system tree, navigate to the system or subsystem where the system attribute you want to override is located.
2. In the System Attributes area of the System Editor, copy the text defining the system attribute.
3. Paste the system attribute text into the Project Attributes dialog and edit the value.
4. Click **OK** to save the override or **Cancel** to close the dialog without saving.

The project attribute is saved and will now override the setting in the base system and all subsystems. You can return to the Project Attributes dialog to delete or modify your project attribute.

X86 Versus FPGA Applications

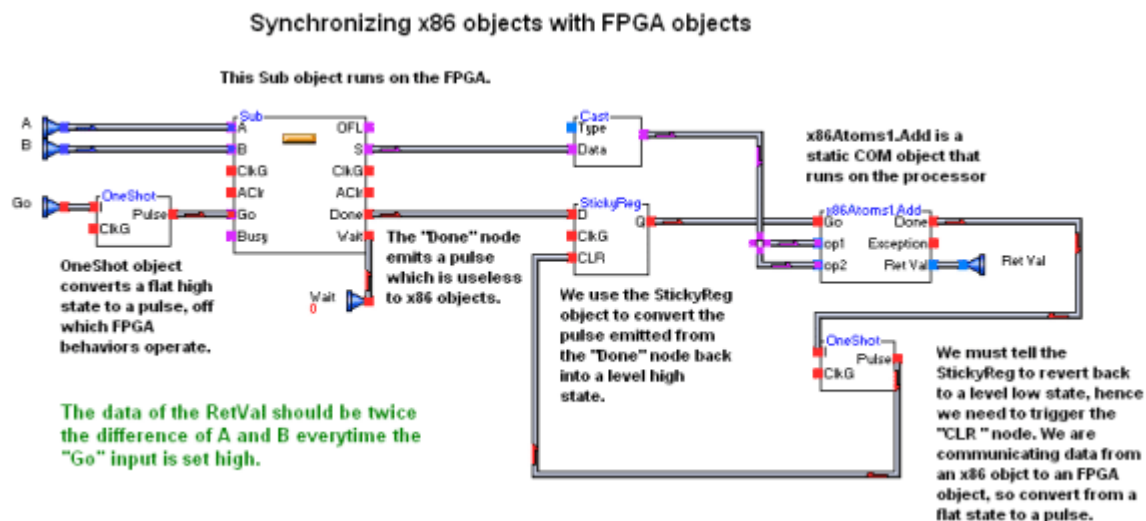
There is an important difference between the execution models of applications that target the X86 processor and those that run on FPGAs. On FPGA-targeted models, the “Go” node should be in a high state for a single machine clock cycle (referred to as a “pulse”); the “Done” node also generates a pulse.

Objects in the X86 system work differently from those targeting an FPGA system. They require the ‘Go’ node to go high and stay high (referred to as a “flat” state), until the system processes it. Likewise, the ‘Done’ node does not merely generate a pulse, but is set and left in a high state, until the ‘Go’ node is set to a low “flat” state; the ‘Done’ node will then be set to a low “flat” state when its events are processed.

To bridge the gap between these two execution models, CoreLib provides two objects: OneShot and StickyReg. The OneShot object generates a pulse through its output node when it receives a high “flat” state in its input node. The StickyReg object emits a “flat” state from its ‘Done’ node when the D node receives a pulse.

An example of the usage of OneShot and StickyReg is shown in the following figure.

x86/FPGA application example



Differences Executing for X86 vs FPGA

The following is a list of differences to keep in mind when executing your design using the X86 system description vs. executing your design for your target FPGA hardware:

- The X86 executor is serial as opposed to parallel. No two things in X86 execute at the exact same time.
- The X86 executor supports COM objects. COM objects never route to an FPGA executor without error.

- FPGA executors have signal propagation delay, whereas the X86 executor does not. The X86 executor runs asynchronous objects as though they took no time at all.
- FPGA clocks have regular oscillations, whereas the X86 executor does not fire another \$Clock signal until the current clock cycle's event queue is free. Hence, the FPGA systems must arrange for timing constraints, which is unnecessary for the X86 executor.
- X86 resources are chopped into TIMESLICES with an indefinite limit based upon the host machine. FPGA systems have real physical resources with known limits on all of them.

Accounting for X86/FPGA Differences

The following are tips for developing and testing projects in the X86 executor that will transfer over to the FPGA system with little conversion effort:

- Minimize logic layers. For example, use binary expansion of recursive gates instead of single bit tiling.
- Put in plenty of data registers. For example, the fast carry chain adders were only designed to do 64 bits of data per clock cycle; for numbers larger than that, use pipelining.
- Use the synchronous (as opposed to the atomic) math operators. Each of these has built-in registers.

The following are tips for developing applications that take advantage of both the X86 executor and the FPGA systems (i.e., File I/O):

- Separate groups of objects for each system into their own objects. Ideally, you will end up with just a couple top-level objects: one would have an X86CPU system attribute, and the others would have a specific FPGA system attribute.
- Any object without a system attribute is targeted to the system of the parent object or the default target system attribute if the parent object does not have a set system. For example, if you put an INVERT gate between two COM objects, it will not get built in the X86CPU system by default, even though that is probably the desired behavior.
- Keep track of transports that will get replaced with communication systems. Sometimes there are synchronization problems between systems that are easier to solve if you know where your communication lines are. The default implementors only ship data between systems in 32-bit chunks, and though this will probably be increased in the future, it can be a stumbling block. For example, if you are sending 32 bits of data and 2 bits of duplex information to the FPGA executor from the X86 system, you may get your duplex information one clock cycle before or after the data, or it may come with 30 bits of data. As a workaround, put all 34 bits in a register in the FPGA system before plugging it in for usage in the FPGA system.
- There is no way to send pulses between systems. Use OneShot and StickyReg in both systems to synchronize pulses.

Chapter 7: Data Sets

Data Sets

The purpose of every Viva application is to manipulate data. It may operate on dynamic input or static information defined within the application itself, or both. In all cases, an application generates output data. Nearly every Viva object has data input and output nodes. The ability to create and use highly structured data sets in a consistent yet flexible fashion is central to Viva.

Data sets are used to define the data types of input and output nodes on an object. As a general rule, inputs and outputs connected together must use identical data sets or connect to a variant.

Fundamental data sets are the building blocks of all other data sets. These data sets are automatically included in each new project. Viva provides the following Fundamental data sets.

Fundamental data sets

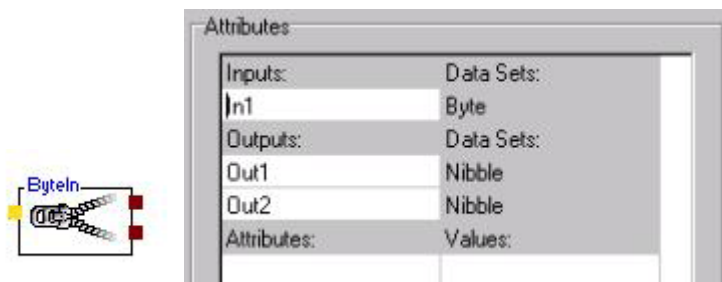
Data set	Definition
Bit	The atomic data set that represents an 'On'/'Off', 0/1, or True/False
Variant	The variant data set is used to represent data when the actual data set is unknown at the time of design and development. This core feature enables Viva to formulate generalized polymorphic objects that work independently of the input data types. The VariantOut (collector) will aggregate one or two data sets into a larger data set defined by the aggregated data sets. The resolution of a Variant data set exposor into the appropriate data set exposor will be performed when the input data sets become known.
List	A special data set consisting of two Variants. The data type does not get elaborated until compile time, when it is resolved into known data sets. More information on list data sets is given later in this section.
Null	The atomic data set that represents no bits. It is what comes out of the top node when a single child data set is exposed. It will not resolve to variant; therefore, objects that receive a null data set must be overloaded to handle the null or the object will be removed without being expanded.

Exposers and Collectors

Data set collectors and exposers are used to compose and decompose data sets. Exposers take a higher order data set as input to expose its lower order constituent child data sets as output. Conversely, collectors get one or more data sets as inputs and compose a higher-level data set that gets exposed in the output. For example, a byte is defined as 2 nibbles. Therefore, a byte exposer accepts a byte as input and exposes two nibbles whereas a byte collector accepts 2 nibbles as input and composes a byte output.

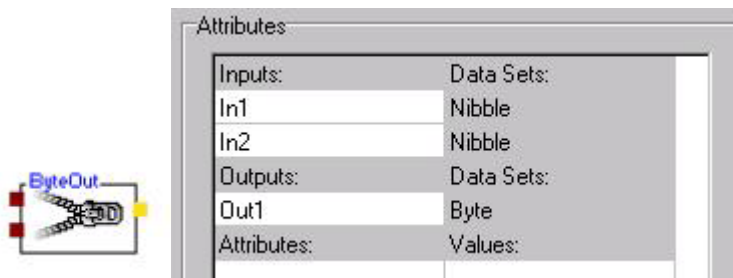
The byte exposer and collector are illustrated in the following figures.

Byte Exposer



The byte exposer is called ByteIn because it takes the byte on the input side.

Byte Collector



The byte collector is called ByteOut because it gives a byte out on the output side.

Bit Pattern Data Sets (MSB, LSB, and BIN)

The VariantIn data set, a primitive object, functions as an exposor. VariantIn takes one data set as an input and then splits the data set into one or two child data sets based on how the data set was defined.

VariantIn, used in conjunction with the MSB (most significant bit), LSB (least significant bit), or BIN (binary tree) objects, determines the exposed data sets. That is, if you run a data set through the MSB object and then VariantIn, you will get the top bit out the top of the splitter, and the rest of the bits out the bottom port, which is what the packaged ExposeMSB(1,2) object does. LSB supplies the opposite. BIN supplies an even number out both the top and the bottom, or, in the case of an odd bitlength, one more out the bottom.

MSB, LSB, and BIN objects are located in the GrammaticalOps tree group of CoreLib.

Patterns are also applicable to lists. In the ExposeCollect tree group of CoreLib, you will find Pack/Unpack objects that do functions to expose and collect items in lists. For example, if you run a list through UnpackBIN you will get two lists out, each containing half the items.

Custom Data Sets and the Data Set Editor

If data sets are required that do not exist in an available library, additional data sets can be defined using the Data Set Editor. To activate the Data Set Editor, click the Data Set Editor tab at the bottom of the main Viva window.

The Data Set Tree displays a hierarchical view of the data sets defined within the current project. It is located on the right side of the Data Set Editor. Some of the operations that are available for use with the data set tree are:

- Double-clicking a data set in the tree loads it into the Editor.
- Highlighting a data set within the data set tree (by pointing to it and left clicking) and pressing DELETE will delete a data set.
- Right-clicking a data set in the tree and selecting **Add to Child Data Set List** adds it to the element list of the currently loaded data set.
- Dragging a data set to a new tree group changes its tree group location.

The data set name displays the name of the data set currently loaded into the editor. This text box is also used to name a new data set. The exposers and collectors created for a data set will share the data set's name, suffixed by the words "In" and "Out". For example, for a data set called 'MyData', Viva will create the exposers 'MyDataIn' and the collectors 'MyDataOut'.

Viva includes the object's input data sets as part of the object hierarchy tracking information. If a node is not connected to an input horn, then the node's data set is used. The word **discon** appears after the data set name for such disconnected nodes.

The data set's children list displays all of the components of the current data set. Data set's children in the list may be removed by clicking them in the data set's children list and pressing DELETE.

Attributes are used to specify how numeric data sets will be displayed on widget forms. When creating custom data sets, you may choose from the following attributes: Unsigned, Signed, Fixed Point, Floating Point, or Complex.

Note: Assigning an attribute to your data set is optional because it defaults to unsigned.

Use the Node Colors control to select the node color applied to the objects that contain your data set. To change the color of a data set, click the **Change** button and select or define a new color with the standard Color dialog. You can also type the color number in the Node Color box.

Note: Although assigning a color to your data set is optional, Star Bridge recommends that you always assign a color or your data set will receive whatever color was last used.

Important: After completing the definition for a new data set, press the 'Save' button.

A built-in feature of Viva is the automatic creation of data set exposers and collectors for each data set created. When working in the Data Set Editor, if you save a data set with more than two components (or child data sets), a warning will recommend a redesign. To create the data set

anyway, you can click **Yes**, but you will have to create an overload to handle this data set for every object that the data set will ever be used on.

List Data Set

Dynamic data sets are employed when you wish to treat an aggregate of data as a single data set. For example, a screen coordinate may consist of a pair of integers and, while Viva would allow you to define a special data set named COORDINATE consisting of a pair of integers, it can be cumbersome to have to define all sorts of special data sets. In the more general case, you may wish to treat a vector of real numbers, and may not know the exact length.

Viva circumvents this difficulty through the use of List. A list consists of the congregation of two or more data sets. Under the data set's children list for the List Data Set, you will find that a list consists of two variant types. Because the input types are Variant, a list can consist of the congregation of two Lists.

To perform a polymorphic interpretation of a Vector, Viva provides the data set exposers and collectors for the List Data Set. They can be found in the data set exposers sub-tree; their names are 'ListIn' and 'ListOut'.

Casting Data

In some cases, data set collectors may not be unique. For example, a 16-bit signed integer (Int) and a 16-bit (unsigned integer) (Word) are both composed from a pair of Bytes. The collector is ambiguous. In this case, Viva will randomly select one of the exposers. To compose two Bytes into an Int type requires that either IntOut be explicitly specified, or that the output of the Variant Collector be cast as an integer using the Cast object.

The Cast object is a special operator designed to convert between various data set types. In all cases, the data, beginning with the lowest significant bit of the source, is transferred to the destination with excess bits being thrown away. If there are insufficient bits to fill the destination, the most significant bits in the destination are filled with zeroes.

One standard and necessary use for the Cast object is to force the data set at the output of objects. In many instances, a VariantOut composes the final result. Viva attempts to resolve the output using the available Data Set Collectors. In some instances, the Data Set Collector may not be unique. As stated earlier, Int and Word are both composed from a pair of Bytes and their respective collectors are structurally ambiguous. To compose two Bytes into an Int requires the output be cast into the correct data set through the Cast operator. Always include a Cast object after a variant collector.

Warning: For cardinal numbers, Cast provides a correct conversion. Use of Cast with integers, fixed point, or more complex types can result in unintended behavior. Cast is intended to be a bit transfer operator, not a conversion operator. For numeric conversions, use the Convert object.

Data Set Recursion

Earlier, it was stated that most data sets were composed of two data sets.

Here is an example without data set recursion:

Assume some object, X, operates on a single Byte input and produces a single, Byte output. You design, develop and test the operator with Byte input and output. If support for any other data sets became a requirement of the operator, you would have to create a new object, probably using X as a template, which supports the new data set. Either way, design gets revisited, code gets edited and a new QA cycle is probably in order.

Here is the same scenario using symmetrical data sets:

Once the Byte operator is created, you create a Variant form of the operator. The footprint of the Variant operator is the same as the byte operator, however its input and output nodes are Variant. This Variant object is defined as two of itself with the input being divided between the two. When Viva compiles your application, it asks what data sets its top-level inputs and outputs should be. For example, if you chose Byte, Viva would synthesize only the Byte operator. If you choose Word, Viva will enter a Recursive loop in which it applies the provided Data Set (Word) to the Variant form of the operator. Since there is no form of X that explicitly supports Word Data Sets, Viva chooses to recurse on the Variant form of X with a Word Data Set as input. The Variant X operator splits the data in half, one Byte each and presents these inputs to two instances of itself. Viva now searches for a form of X that operates on Byte Data Sets. This form of X is defined and Viva synthesizes two instances of the Byte operator. The outputs of the two Byte operators are aggregated by Variant collectors and exposed as the originally defined Data Set of Word.

Static Data Sets in Viva

The purpose of every Viva application is to process data. The operatory mediums may be statically predefined, dynamically determined, or both. Nearly every Viva object has data input and output nodes. The ability to create and utilize highly structured data types in a consistent, yet flexible fashion is central to Viva.

Static data sets have definitive size, structure, and attributes. The system-generated static data sets are located in the System\Static and System\ComPredefined tree groups. The Data Set Editor allows the user to create additional static data sets. Static data sets must be used to interface with COM and with Widgets. Widget and COM objects must use Static data sets because their size and context are predefined.

CoreLib contains objects for converting to/from static data sets; DynamicOut and StaticOut.

DynamicOut—Takes a static data set as the input and converts it into a contextual data set of the same type and size. This object is normally used after widget/COM input objects to convert the data for use by CoreLib's contextual objects. Contextual data sets must be used on almost all Viva objects.

StaticOut—Takes a contextual data set as the Data input and converts it to the static data set on the Type input. This object is normally used before widget/COM output objects to convert the data for output. The Type input is normally the static data set fed input to the source DynamicOut object.

This object is only meant to do same-type conversions. Therefore, a Type input of Float and a Data input of Floating is legal, but a Type input of Byte and a Data input of Floating are not legal.

Consistent Structure and Design

Data sets are used to define the data types of input and output nodes on an object. As a general rule, input and outputs connected together must expect identical data sets. Almost all Viva data sets are composed of two other data sets. This is due primarily to application design considerations and is not a limitation of Viva itself. Actually, Viva allows for the combination of any defined data sets into a higher data set. However, you can take advantage of some of the most powerful features of Viva by using a symmetrical data set design approach.

Viva defines the following “static” data sets:

Static data sets

Data set	Definition
Dbit	Double Bit. 2-Bit Cardinal. Pair of Bits. Half a Nibble.
Nibble	4-Bit Cardinal. Two DBits.
Byte	8-Bit Cardinal. Two Nibbles.
Word	16-Bit Cardinal. Two Bytes.
DWord	Double Word. 32-Bit Cardinal. Two Words.
QWord	Quad Word. 64-Bit Cardinal. Two DWords.
Int	16-Bit Signed Integer. Two Bytes.
DInt	32-Bit Signed Integer. Two Words.
Fix16	16-Bit Fixed Point. Two Bytes.
Fix32	32-Bit Fixed Point. Two Words.
Float	32-Bit IEEE Floating Point.
Double	64-Bit IEEE Floating Point.
Qint	64-Bit Signed Integer. Two Dwords.

The following data sets are additionally defined for standard interaction with COM components:

Data sets for interaction with COM components

Data set	Definition
BinaryString	Pointer to a holder of unicode character strings, the COM standard.

Bool	Standard 32-bit Dataset used to represent a true or false state; false = 0; true = -1 (FFFFFFFF)
ComInt	Standard signed integer type; length is presently 32 bits.
ComUInt	The unsigned equivalent of ComInt.
ComVariant	Contains 8 bytes of variant data, and a holder for the data's context. Any COM Dataset can be fed into a ComVariant input node. In general, do not use this directly .
Currency	A rarely used COM standard data type. 8 bytes of fixed-point data. Included in Viva for purposes of compliance with the COM standard.
Date	8 bytes representing a date and time. Included in Viva for purposes of compliance with the COM standard.
Enumeration	32-Bit Signed Integer for support of enumerations defined in COM Type Libraries.
Hresult	Standard 32-bit unsigned type for a status code return type. Rarely seen in user-end COM interfacing.
IDispatch*	This abstract 32-bit unsigned data type holds the address of the interface for a COM component, through which all of its programmatic functionality is invoked.
Signed Byte	8-bit signed integer.
Unsupported	Used in Viva as a fallback to cover the rarely-used COM-supported Datasets that Viva does not presently support, or cannot resolve.
IUnknown*	Pointer to an unknown integer data set.
Pointer	32-bit data set used to define data sets that point to other data sets.
WideChar	COM standard 2-byte character data set.
WideString*	COM standard wide string (composed of wide characters).

Chapter 8: Synthesizing Your Application

Synthesizing Your Application

Synthesize a sheet

When you synthesize a sheet you compile the sheet and the application you have created on that sheet. If you select the **Automatically Run after Compile** preference (see [Viva Preferences](#)), the sheet will execute after compiling.

Click the **Sheet** menu, then click **Start/Cancel Synthesis** to compile the current application worksheet.

Selecting this menu command during the compilation of an application worksheet will stop the compilation process and return to the worksheet editor.

After you have compiled an application worksheet, you can choose to execute it at any time by clicking the **Sheet** menu and then clicking **Play/Stop Sheet**.

Warning: It is recommended that you create separate folders for each project. When you synthesize a sheet, the output is stored in the project build directory. Subsequent builds will overwrite the data in that build directory. Therefore, if you store multiple projects in the same directory you risk overwriting one project's build output with another.

Creating Executables

To create an executable, you must first use the I2ADL editor to create an application, successfully synthesize it, and then save it as an executable file. Compiled Viva executables have a file extension of .vex.

The status bar will indicate the compile status. When compiling, two numbers appear. The number on the left indicates the number of objects expanded during a compile. The number on the right is an indicator of the number of objects processed after a compile. During a compile, the number of expanding objects that increments, is the number of objects and the number of connections between objects that are expanding. The number of objects processed is the number of primitives and the number of connections between primitives after a compile has completed.

Running Saved Viva Executables

Click **Executable**, then **Run Executable** to locate and load an executable file. A saved executable runs independently from the Viva editor, hence, you can also double-click the file name from within Windows Explorer. The executable will load and execute in the same manner as when it was compiled within the Viva editor.

Resource Usage

To see what resources were used in the last compile, perform the following steps:

1. After synthesizing a sheet, click the **Resource Editor** tab that is located at the bottom of the workspace.
2. Click **Display Usage**. Resources used by each system are displayed.

Note: For exact resources used, refer to the files created by Xilinx during the place and route (PAR) stage. The .PAR file is located in the Builddir directory, which defaults to the user directory.

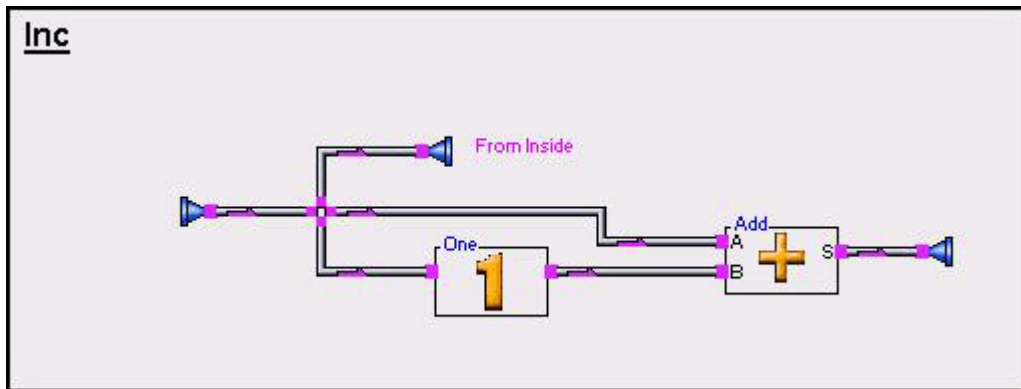
Chapter 9: Advanced Tasks

Creating and Removing a Debugging Horn

If there is data you want to carefully analyze, you may want to create a debugging horn. To do so, send the data that you want to analyze to an output horn. Assign the output horn an attribute of Resource and a value of Global. That output is the defining source of a global label. Wherever you want to see the data, create an input with the same name as the global label. Assign the input an attribute of Resource and a value of Global or *Global.

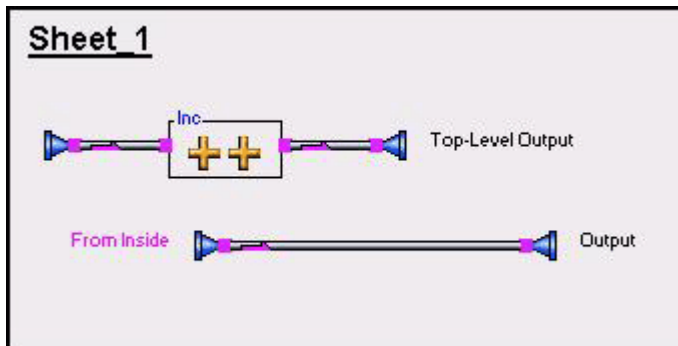
The following two figures illustrate the process of creating a debugging horn:

Creating a debugging horn – second level of an Inc object



This Inc object has an added output called From Inside. The attribute for the output is Resource and the value is Global.

Creating a debugging horn – top level of an Inc object

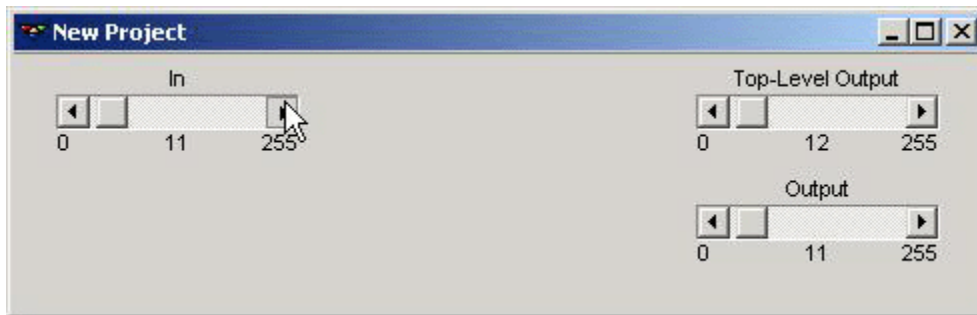


On the top-level sheet of the Inc object, there is an input and output to enable the data to display in the **Runtime** dialog box. The input has the same name as the defining global source (the output previously described—"From Inside") and a resource with a value of Global.

Note: There are inputs and outputs in implementors that are automatically promoted to the Runtime dialog box. If you prefer not to see the inputs and outputs in the Runtime dialog box, you may want to manually remove them from the implementor.

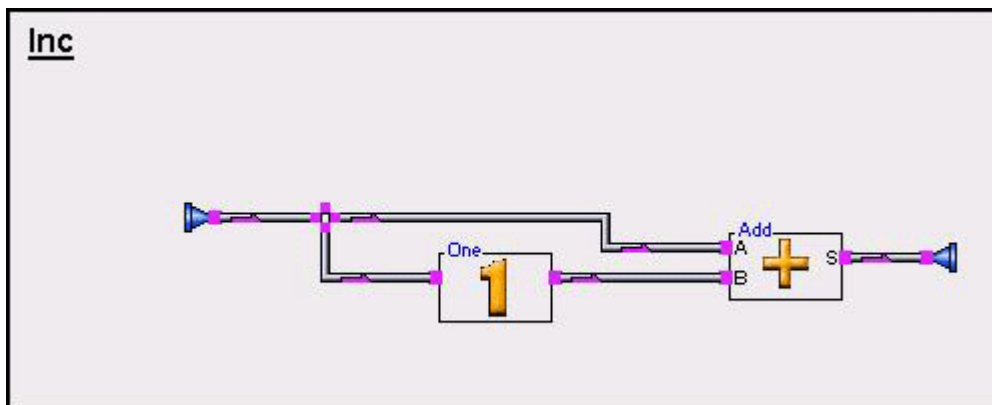
The following Runtime dialog box illustrates the synthesis of the specified Inc object.

Runtime dialog displays synthesis

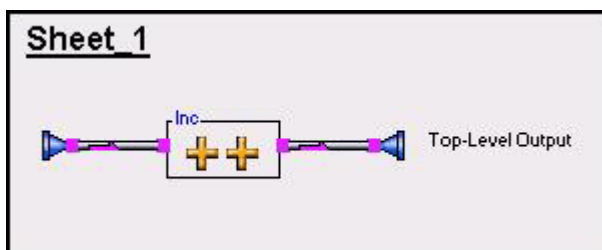


The primary reason to remove a debugging horn is to clean up the **Runtime** dialog box. To remove inputs and outputs from the Runtime dialog box, remove the global defining source and the inputs that use it. For example, the following two illustrations depict removal of the global defining source and the input that used it.

Removing the global defining source

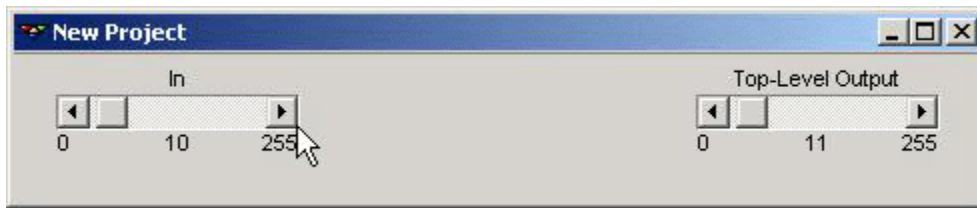


Removing the input



The resulting synthesis is illustrated in the following Runtime dialog box.

Runtime dialog after removal



Specifying Runtime Preferences

You can specify whether the **Runtime** dialog box will automatically launch after a compile. This option enables you to choose to just compile the sheet or to compile and run the sheet, which means you could compile without programming FPGAs.

To specify launch preferences for the **Runtime** dialog box, do the following:

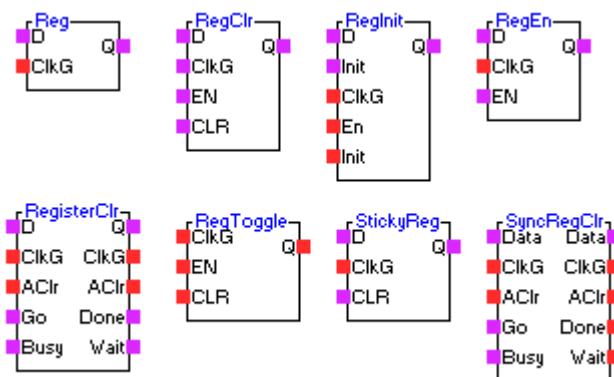
1. From the **Tools** menu, choose **Preferences**.
2. Click the **Compiler** tab.
3. Select the **Automatically Run after compile** box to enable/disable automatically launching the **Runtime** dialog box after compiling.

If you want to manually launch the **Runtime** dialog box after compiling, from the **Sheet** menu, select **Play/Stop Sheet**.

Disabling the intermediate data results display

It is possible to disable the intermediate data results display in the Runtime dialog box by adding a Register object before an Output object to achieve the desired effect. Placing a Register object before an Output object causes the register to produce a result when it receives a Go signal. The following illustration depicts Register objects you could use to disable the intermediate data results display in the Runtime dialog box.

Register objects for disabling intermediate data



Another way to optimize data output speed is to adjust the system attribute, **EmulatorSpeed**. Increasing the value for **EmulatorSpeed** increases the speed of the data output.

Note: Increasing the **EmulatorSpeed** value may cause a decrease in performance quality (response time) of the Runtime dialog box. The default **EmulatorSpeed** value is 50.

To adjust the **EmulatorSpeed** value, open the System Editor by selecting the System Editor tab. In the System Attributes field, type **EmulatorSpeed=<preferred value>**. The change is automatically updated.

Troubleshooting Compile Issues

The following are potential causes for compile issues:

Go, Done, Busy, and Wait are control nodes that have special handling. Therefore, if you have duplicate control node names, the router will fail. Variations on the names are fine.

You cannot rely on output horn data type settings to enforce data types. That is, don't change the data set on an output horn; use a cast object. This is especially evident in feedback loops on registers because it is possible that the variant gets resolved all the way around a loop and back into the original object it came from.

If you don't have the ClkG plugged in correctly to carry through to the other side of the sheet, underlying layers of logic may get ripped out. Watch the Synth graphic and see if, as it drills into objects, most things end up getting ripped out.

The \$Select objects are pre-compiler multiplexers that affect what code actually gets built. The \$Select objects will always be resolved before their down-stream objects. They wait for their S parameter, which must receive a constant. Use \$Select pairs to improve compile speed. The objects between the two \$Selects will be discarded based on the S input.

System primitives only support bit inputs and outputs. Any attempt at something else will throw the Can't Route error listing the number of plugged inputs and outputs on the offending primitive where one of the plugs is larger than a bit.

Another cause of not being able to route is related to the assigned systems of objects. The parent object's system attribute will not override a hard-coded system in a child object. If you get a large number of things that cannot route, it is likely you have a child object assigned to the wrong system somewhere.

System descriptions contain overloaded objects that are necessary for the hardware. The \$Emulator \$ADC/\$ADSU will not function properly in the FPGA-based system description. If such overloaded objects are not written when you open the system description (because you forgot to set your preferences to replace existing), your timing constraints will likely fail. Also, if you open the system description before Corelib, you will want to retain the existing modules.

There is potential for too many bits in the communication system. Viva currently supports 512 virtual I/O lines.

Directory System Attributes

“BuildDir” system attribute

When an FPGA project is compiled, the Xilinx tools create a number of disk files. The new system attribute “BuildDir=<directory>” allows the user to override the user directory (C:\Documents and Settings\<username>) as the directory for these files. This attribute is entered on the base system in the System Editor. It allows simultaneous Xilinx compiles by placing the files created by Xilinx in different directories. Most users do one compile at a time. You can use the default user directory without conflict to do multiple Xilinx compiles at the same time. Change the build directory to be the project directory and put every project in its own file.

The <directory> part of the attribute may be a fully qualified name (starts with a drive letter or backslash), or it may be relative to one of the following directories:

“ ”	C:\Program Files\Starbridge\Viva\VivaSystem
“\$V\”	C:\Program Files\Starbridge\Viva
“\$S\”	C:\Program Files\Starbridge\Viva\VivaSystem
“\$P\”	Current project’s directory
“\$F\”	FPGA file name (See the following note.)
“\$B\”	Build directory
“\$U\”	User settings directory (C:\Documents and Settings\<username>)

Note: Because the FPGA file name requires an object assigned to a system, “\$F\” is only available on object attributes.

Viva.ini file

Use the Preferences option on the Tools drop-down menu to select a number of settings that customize Viva. These preferences are stored in the Viva.ini file. This file is located in the user’s settings directory (C:\Documents and Settings\<username>). Thus, each person on a shared use machine should have a distinct user name to avoid overwriting someone else’s preferences. In the event that the user settings directory does not exist, Viva will store preference changes in the shared Viva.ini file located in the system directory (C:\Program Files\Starbridge\Viva\VivaSystem).

User settings directory

The user settings directory (C:\Documents and Settings\<username>) is the default build directory and also the default new project directory.

Project directory

You can save projects in any location; the location of a saved project becomes the project directory. By default, Viva will save a new project in the directory of the last project you saved, unless you specify otherwise. Viva will always place the ErrorLog.txt file in the associated project directory.

Reconfiguring FPGAs with the \$Spawn Object

The \$Spawn object, located in the X86 CPU system, allows you to execute another previously saved Viva file. The Save File can either be a Viva Executable file (.vex) or a Viva Project file (.idl) as specified by the \$Spawn object's FileName attribute.

\$Spawn calls VivaRun.exe if a .vex file is specified for its FileName attribute, or Viva.exe if an .idl file is specified. If the Viva Project file specified is not compiled, then Viva will compile it before attempting to execute it.

\$Spawn returns a double word value from the called process once it has terminated, via its "Result" node.

An attribute of "Exclusive=true" on a Spawn object will cause the calling process to halt and wait for the termination of the child process.

All \$Spawn'd processes self-terminate when their caller terminates.

Note: Viva recognizes if a project was compiled before it was saved. This allows the project to be \$Spawn'd much faster (without recompiling).

Go

The Go object is a notification mechanism that a Viva execution process has begun. A compiled behavior can have any number of these objects. The Send node issues whatever data was passed to this process on the command line, if applicable; 0 otherwise. The Done node sends out a 0 value, then a 1 value, when the process begins. Use it to trigger any initialization functionality that is warranted by any behavior.

Stop

The inverse of the Go object is the Stop object, which has three purposes:

- To communicate to the behavior that it has received a termination request
- For the behavior to communicate to Viva that it is ready to be terminated
- To allow the behavior to relay data to the calling process. Data on the Stop of the executed behavior appears on the Result of the calling \$Spawn object.

The "Stopping" node sends out a 0 value, then a 1 value, when the process has received a termination request. The process will not actually terminate until *all* Stop objects have received a high state into their "Go" nodes. The data given the "Stop" node of the last Stop object (that has data given to it) to receive this signal will be the data passed back to the calling process.

Implementing \$Spawn

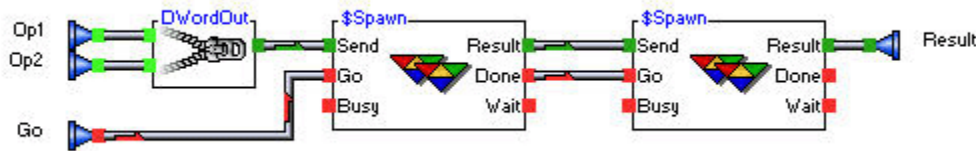
In this example, the \$Spawn objects call two different Viva “executable” files (.vex) in sequence, which both target the same Programming Element (PE). Data is passed to the called process through the “Send” node, and data is received from the called process through the “Result” node.

In this case, the “Result” output should be five plus the product of “Op1” and “Op2”.

“config1.vex”, called by the first \$Spawn, will program PE7 with a behavior having a multiply object, then split the parameter (given through the “Send” node) back into Words and multiply the two of them.

“config2.vex”, called by the second \$Spawn, reconfigures PE7 with a behavior having an Add object, and then adds five to the number passed, which is the product of “Op1” and “Op2”.

Implementing \$Spawn example



In this process, two different FPGA behaviors run at different times on the same chip, without having any FPGA-targeting objects in this behavior, and without having to compile anything of substance.

For a simple test case, put a value of 3 on the Op1 input, and 5 on the Op2 input, and set the Go input high. After two processes return, you will have to press the “Bring Widget Form to Front” button. The Result output should have a value of 20.

Note: Regarding the maximum number of Spawns possible: Under Windows NT, there should be no limit, except as dictated by memory.

Chip targeting

The chip targeted by the called Viva executable project is specified at compile time by the user, so you must make different versions of the same executable to target different chips; however, you only need one source “.idl” file.

It is possible to reuse data from the calling process even though the \$Spawn object targets the same chip. You can use a Viva or COM process to save the necessary data before calling the process that will re-program the targeted chip.

Precompiling Portions of a Viva Project

If the **Synthesize to EDIF cell only** option (available through the **Sheet** menu) is checked, your compile will build the current sheet to an EDIF (Electronic Data Interchange Format) cell without attempting to execute. This is done to support "incremental compile," allowing Viva to export EDIF. Viva continues to reference exported objects as black boxes as it has always done. That is, Viva does not import any other languages, but you can reference precompiled objects as black boxes. The following conditions apply to this option:

- The compile will fail if the number of FPGA systems being used is not exactly 1.
- Having an X86CPU, X86UI, and UIXCPU system is required, though they will only support Exposer, Collector, Input, Output, and Transport.
- A reference to the EdifFile attribute will be added to the top of the newly built EDIF file.
- The name of the object will be the sheet name with the input data sets appended.
- The export file will end up in the BuildDir, which defaults to the user directory.
- The new object will automatically end up in the System Objects tree in the FPGA system.
- All the ClkG (clocks) signals and other globals must be overloaded and plugged into actual input and output horns. This is because all the implementors (always-built objects) and transports will not get built when targeting a single EDIF cell. This is so the cell can later be plugged into the global clock for a project.

EDIF File Support

The object attribute “EdifFile” supports EDIF file imports into EDIF NetFiles built by Viva. Xilinx cell properties can be included in an EDIF netlist by using the “FPGA” object attributes. For more information, see Importing EDIF into Viva in Examples.

Chapter 10: Saved Files (text based)

Saved Files (text based)

When Viva saves a file, it groups the following components sequentially:

1. Header
2. COM dependency references
3. DataSet definitions
4. Object prototypes
5. Object definitions
6. System definitions.

The components within each group are listed in the file in the order in which they were created. Therefore, objects added to a Viva project/sheet/system will always appear at the end of the save file.

Comparing or merging Viva save files is similar to comparing or merging text files for any other programming language since the listed components can be defined in any order, with the exception of the header.

Star Bridge recommends observing the following precautions regarding saved files:

- Be sure not to interject return characters where they do not belong. For example, do not include a closing parenthesis on a line separate from the last list item that it enclosed.
- Identifier names should not contain non-alphanumeric characters other than “_” and “\$”.
- Always make backups before editing saved files.

Header

The first component of the file must be / Viva (and a text base version number). Other parts of the header identify the type of file and at what date/time it was created. The version number is important because it controls how files are read in.

COM Dependency Reference

ComLibrary

ComLibrary “[file name]”

Example: ComLibrary \$\$\VivaAtom.dll"

[file name] The fully qualified path and file name of the COM Type Library. Directory prefix codes are allowed.

ComForm

ComForm "[file name]"

Example: ComForm \$U\Tests\Edit.cfm

[file name] The fully qualified path and file name of the COM Form file (*.cfm). Directory prefix codes are allowed.

ComObject

ComObject [Prog ID] [object name]

Example: ComObject VivaAtomLib.x86Atoms x86Atoms1

[Prog ID] The Viva-defined (not Windows-defined) programmatic ID of the COM class to instantiate. Consists of the name of the containing library and the COM class name, separated by ".".

[object name] The alphanumeric tag by which this instance will be referenced within Viva.

Data Set Definitions

DataSet [name] = ([child1], [child2]...); //_Attributes [context type], [color], [tree group], [COM variable type];

Example 1: DataSet Fix32 = (Word , Word); //_Attributes 4,12632256,System\Static

Example 2: DataSet "DWord*" = (Word , Word); //_Attributes 1,11141375,System\Static,16403

[name] The name of the DataSet. Can contain any printable character other than a tab or return; if it contains characters other than a through z, 0 through 9, or underscore (_), then it must be enclosed in double quotes.

[child] The name of a data set that is a component of this data set. Can have any number of "children", or component data sets, but do not define a data set that has less than 1 or more than 200 children.

[context type] Numeric value corresponding to one of the values listed in the Widget Handling Attribute box of the Dataset Editor.

\$=unsigned

2=signed

4=fixed point

8=floating point

16=complex

This information is used by widgets to display data in the correct format.

[color] The 24-bit numeric code for the display color for the current data set on nodes.

[treegroup] Optional. The name of the data set tree group in which this object will be displayed. Consists of the name of the parent group on each level separated by “\”.

[COM variable type] Optional.

Object Definition and Object Prototype

The Object definition is as follows:

```
Object ([outputs]) [name]([inputs])
; //_Attributes [Attribute list]
{
    [documentation]
    //_ Object Prototypes
    [object prototype list]
    // Behavior Topology
    [node connection listing]
}
```

The portion enclosed in “{” and “}” is the behavior definition, and does not apply to primitive objects or to object prototypes.

Example:

```
Object ( Bit O) Mux( Bit A, Bit B, Bit S)
//_Attributes System=X86CPU,Documentation=Bit
{
// Mux - Variant Select Case
//
//_ Object Prototypes
Object ( Bit A) Input; //_GUI 9,12
Object ( Bit B) Input:A; //_GUI 9,21
Object ( Bit S) Input:B; //_GUI 9,24
Object Output( Bit O) ; //_GUI 78,14
Object ( Bit Out1) INVERT( Bit In1) ; //_GUI 22,15
Object ( Bit Out) AND( Bit In1, Bit In2) ; //_GUI 45,11
Object ( Bit Out) AND:A( Bit In1, Bit In2) ; //_GUI 46,20
Object ( Bit Out1, Bit Out2, Bit Out3) Junction
    ( Bit In0) ; //_GUI 19,24
Object ( Bit Out) OR( Bit In1, Bit In2) ; //_GUI 61,12
// Behavior Topology
Output = OR;
INVERT = Junction.Out1; //_GUI 20,18
```

```

AND.In1 = Input;
AND.In2 = INVERT;  //_GUI 40,16, 40,18
AND:A.In1 = Input:A;
AND:A.In2 = Junction.Out2;
Junction = Input:B;
OR.In1 = AND;
OR.In2 = AND:A;  //_GUI 58,17
}

```

Object Definition

[outputs]: A comma-separated list of output parameters, each consisting of the data set of the output, followed by its name.

[name]: The name of the object being prototyped. If it contains white space, then it must be enclosed in quotes.

[inputs]: A comma-separated list of input parameters, each consisting of the data set of the input, followed by its name.

[Attribute list]: A comma-separated list of attributes for the object, in the standard format of [name]=[value], with no intervening white space.

Behavior Definition

[documentation]: Used to populate the Documentation field of the object, as seen on the right-hand side of the Attribute Editor. Can span any number of lines; each signified by “//” prefix.

[object prototype list]: The newline-delimited list of object prototypes (as explained earlier) of objects used in the behavior of this object. In Viva, you reference and prototype an object using the same statement. Each object name in this listing must be unique for this behavior. For each object having a name ambiguous with that of another object in the same behavior, Viva adds a suffix beginning with “:”; such suffix does not become part of the actual object name once the file is loaded.

[node connection listing]: The newline-delimited list of node connections, taking the form *[object name].[input node name] = [object name].[output node name]*. The [node name] can be omitted for a node if it is the only one in its list (inputs or outputs).

System Definition

The System definition is as follows:

```

System X86
  //_Attributes [attribute list]
  {
    [nested system definitions]
  }

```

Example:

```

System X86
//_Attributes
1,Resource=Default,DefaultInputWidget=ScrollBar,DefaultOutputWidget=
ScrollBar,
WidgetSystem=X86UI,WidgetResource=TIMESLICE,WidgetLocation=?,
DefaultTargetSystem=X86CPU
{
System X86CPU
//_Attributes 2,Resource=X86Main
{
}
System X86UI
//_Attributes 3,Resource=X86UI
{
}
System UIXCPU
//_Attributes 7,Resource=UIBus
{
}
System X86ClkImport

```

```

//_Attributes 6,Resource=ClockImport
{
}
}

```

[attribute list] List of attributes in the same format as in Object definitions. The first attribute listed must be the 1-based ordinal value of the system type as defined in the SystemType (Executor) combo box in the System Editor. The second attribute must be "Resource=" + the name of the system's resource prototype as listed in the Resource Prototype combo box in the System Editor

Chapter 11: Go Done Busy Wait

Go Done Busy Wait

Go Done Busy Wait (GDBW) is the primary state control abstraction in Viva. GDBW allows you to create a distributed one-hot state machine throughout any design. GDBW also permits complex synchronizing between objects with the ability to stall the pipe at any point without losing data or commands. More sophisticated techniques allow for synthesizing recursive state machines with very simple parameterization.

It is important to note the node reversal inherent to GDBW. Wait is an input, and Busy is an output; node placement might imply the opposite. Go->Done and Wait->Busy defines a complete control handshake between two objects. This handshake defines a protocol both objects must follow.

Usage Guidelines

- The Go input is ignored while the Busy output is high.
- The sending object should not send a Done output when the Wait input is high.
- Any time a Busy output is low, a Go input can be received.
- Incoming associated data is valid only when the Go input is high.
- If the Wait input goes high during the same clock cycle that the Done output goes high, then the Done output will go low and the associated data will not latch; this requires that the object hold the data until its Wait input goes low, signaling that it can send the data.

Go – Input

Go tells an object that data presented to it is valid and the object should begin processing data. Generally, Go must be a single clock cycle pulse because most objects cannot accept data every clock cycle, and their Go instantiates an initialization state. Fully pipelined objects are an exception to this rule. Go is usually obtained from an upstream object's Done. If you provide a Go signal to an object from the widget interface, you will generally want to filter that signal through a OneShot object to ensure the object receives a single pulse.

Done – Output

The Done signal indicates that data an object is presenting is valid. Done is generated as a single clock cycle pulse to conform with the expectations of a downstream object's Go.

Busy – Output

Busy is an object's means of communicating that it is not currently able to accept new data and a Go. Busy will propagate from a downstream object to an upstream object's Wait. Busy tells an upstream object that it is responsible for maintaining the data that it is currently providing. Busy will go high on the same clock cycle that a Go is received and should remain high through the clock cycle that the object in question produces a Done.

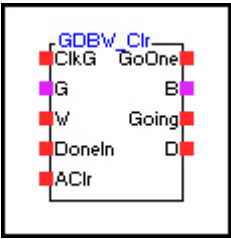
Wait – Input

Wait is generally received from a downstream object's Busy output. When an object's Wait input is high, that object is responsible for maintaining the state of all data that it is providing. It is also responsible for not generating another Done signal, as that signal will most likely be ignored by the downstream object.

GDBW_Clr Object

Star Bridge recommends using the GDBW_Clr object when creating a Go Done Busy Wait object to ensure proper functionality.

GDBW_Clr object



GDBW_Clr object control signals and descriptions

Control signals	Control signal description
GoOne	Goes high with G and not B. Drops with first clock cycle.
Going	Rises when GoOne and DoneIn are low and stays high until DoneIn goes high.
DoneIn	Causes Going to go low on the next clock cycle. Will cause B to drop and D to rise as long as W is low.

Chapter 12: Recursion

Recursion

To understand recursion, it is recommended that you review the *Data set recursion* section because recursion is the process of deconstructing a data set, performing some operation, and then reconstructing the data set.

The following list provides an overview to help you develop a conceptual framework of how recursion works within Viva:

- Recursion is a compile time mechanism; it is not a run-time mechanism.
- Recursion mechanisms depend on data type, not data value.
- Recursion allows you to program independent of bit widths.
- Use data set overloading to avoid an infinite loop during the recursion process.
- When performing recursion in Viva, it is important to note the following:
- On a sheet, all exposers and collectors display the least significant element on the bottom.
- On a sheet, when you split an MSB, the exposed bit (or MSB) is on the top, and everything else is on the bottom.
- On a sheet, when you split an LSB, the exposed bit (or LSB) is on the bottom, and everything else is on the top.

Variant Overloading

Variant overloading is the backbone of Viva's recursive synthesis. It also forms the basis for data set polymorphism and the resolution of high-level objects into primitive objects. Unlike many other graphical programming interfaces, where objects represent cores that are linked together by the tool, every object in Viva is formally and recursively synthesized from input data sets down to primitive objects. These primitive objects could be from the Primitive Objects Tree (AND, OR, INVERT, and so on) or they could be primitive objects found in a system library.

To understand the process represented by Variant overloading, you must first understand object footprints. An object's footprint is a combination of the name of the object and the number of inputs and outputs it contains. In the following diagram, the two AND gates have the same footprint, even though the object on the left has two Variant inputs and one Variant output, and the other has all Bit inputs and outputs.

Same footprint



The following two AND gates do not have the same footprint, even though they both have Variant inputs and outputs in common. The object on the left has two inputs and the object on the right has three inputs.

Different footprints



On the next set of two objects, there are differing data sets. These objects still have the same number of I/O however. What is important is only the case-sensitive name of the object and the number of inputs and outputs.

Same footprint – example 2



When writing a Viva program, one will usually use the most variant version of an object. Of the three AND gates with identical footprints that follow, the one on the left is the most variant.

Most variant footprint

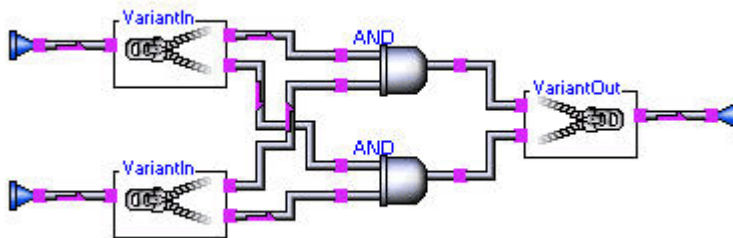


The recursive synthesis process takes this most variant object and recurses around the data sets that are input to the object, breaks them down into smaller data sets until all that is left is a set of primitive objects. This is possible because at each step of the recursive process Viva re-saves to the least-variant version of an object.

Recursive Footprints and Data Set Polymorphism

The following is the recursive behavior contained in the most variant AND object.

Variant AND Object

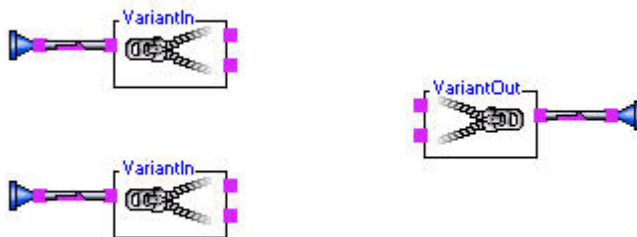


Both inputs are piped into Variant Exposers that split the incoming data sets into their two children data sets. Two Variant AND objects take the high-order and low-order children data sets respectively. These AND objects are self-references. If you drill into them, you will see the same recursive behavior. Coming out of these self-referential AND objects, the high-order and low-order data sets are collected back into the original data set that was input to the object.

The following instructions indicate how to build self-referential recursive footprints.

1. Open a fresh copy of Viva and do not load any systems or library files. At this point, there are only the objects in the Primitive objects group.
2. Drag two inputs and one output onto the worksheet. Attach a VariantIN to each of the inputs and a VariantOut to the output. Your sheet should now look like the following diagram:

Self-referential recursive footprints example



3. Now that the object's interface is defined in terms of the number and type of inputs and outputs, wrap this behavior sheet up as an object. You can do this by selecting **Convert Sheet to Object** from the sheet menu, pressing the **Convert Sheet to Object** button, or by pressing F8.

4. Give the object a name. In the Object Name field, type AND (case sensitive). Click **OK**. You should see your AND object in the Composite Objects Tree Group.
5. Double-click the object.
6. Drag two copies of the object you just created onto the sheet. This is the self-referencing step.
7. Route the high-order output nodes of the VariantIns to the top AND and the low-order to the bottom AND. Also pipe the outputs of the ANDs to the VariantOut. Your behavior sheet should now look like the Variant AND object:
8. Repeat step 3 to update the object. Make sure the **Update Original Object** radio button is selected. This is the default.

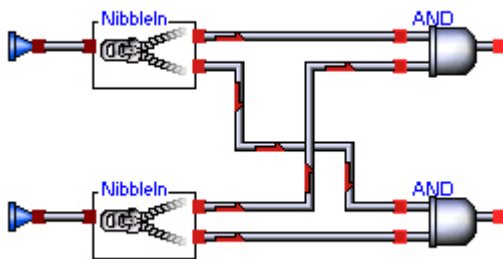
You have created a Variant AND object, that when coupled with the bit-level AND from Primitive Objects will handle any input data sets, as long as the two inputs carry the same data set.

The following steps provide more detail about how this is done. Remember that you currently have a variant and bit AND objects with the same footprint.

1. Starting with a new behavior sheet, drag the Variant AND object out and attach inputs and outputs to it.
2. Assign the data set of both inputs to be nibbles.
3. When you hit play, the behavior will be recursively expanded until all four bits that make up either of the nibbles go to four separate Primitive (Bit) ANDs. The next section explores each step of the recursion in more detail.

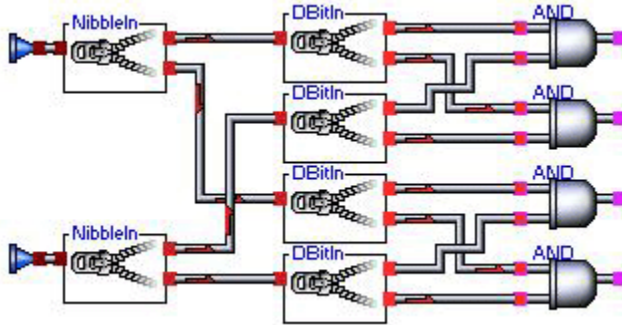
In the first step, the AND gate receives two nibbles that go directly to the VariantIns. NibbleIns will replace these VariantIns through selective synthesis. So the recursion looks like this:

Variant AND Object – recursion



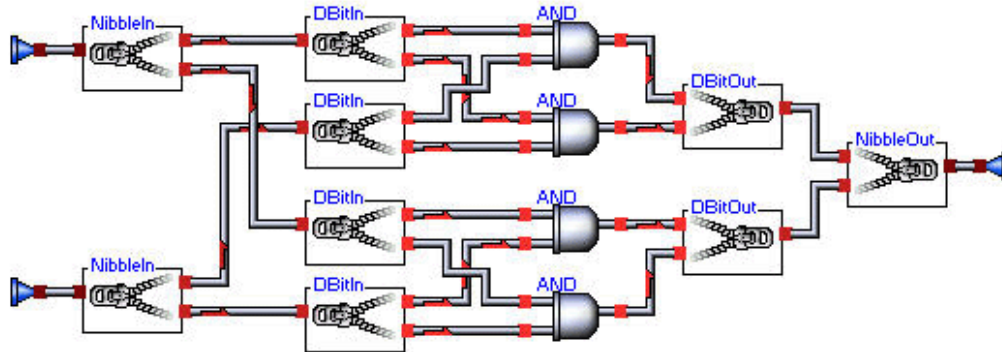
Nibbles then get broken into DBits and result in two ANDs with DBit inputs. The Variant AND is still the least variant AND that can handle DBit inputs. In this step, you export the variant ANDs again with DBitIns.

Variant AND Object – export variant ANDs



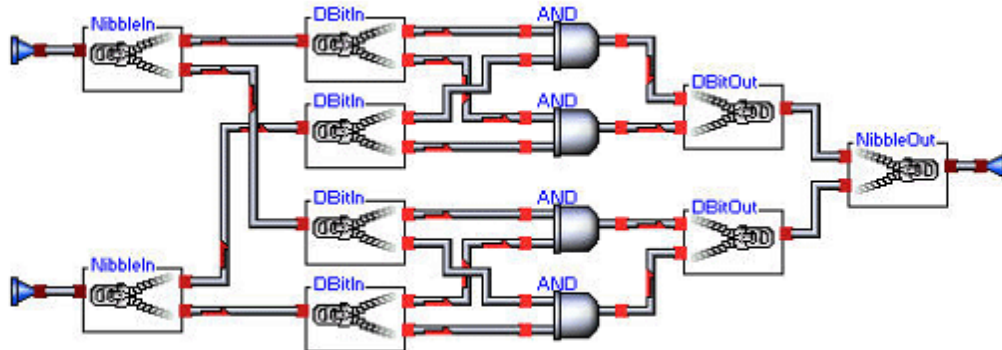
There are four AND objects with matched Bit inputs. The variant AND is no longer the least Variant, and these four ANDs will be immediately replaced with bit-wise ANDs from the primitive object tree. The Primitive ANDs act as the *Base Case* and terminate the recursion.

Variant AND Object – terminate recursion



All that is left to do now is return from the recursive calls and rebuild the incoming data set. The final expansion of the objects looks like the following figure:

Variant AND Object – final expansion of objects

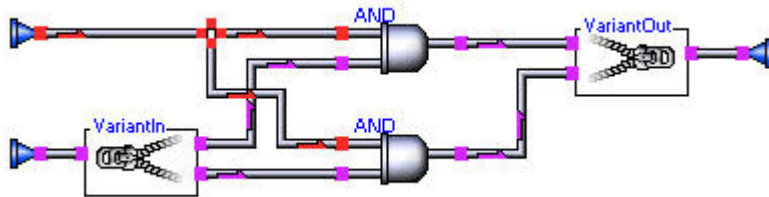


These two objects are all that is needed to cover AND of any two matching data sets. This process will work identically for any data set as long as the two incoming data sets match.

Consider the case where the incoming data sets do not match. If one input to the AND is a Bit, and the other is an arbitrary Data Set (Variant), you will have the ability to gate the data on the Variant input. You will need to build another version of the AND gate to accomplish this. Make the top input the bit and the bottom the Variant.

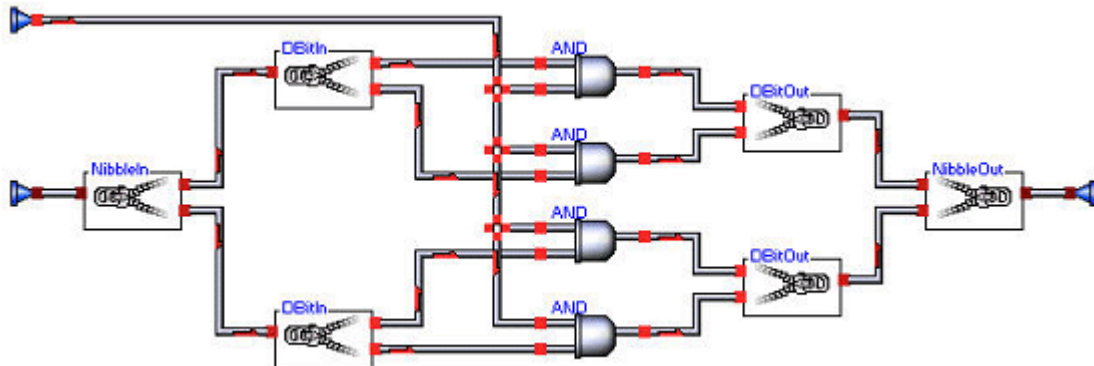
Drill into the Variant AND object. Delete the top Variant input and the corresponding VariantIn. Replace with a Bit input as follows:

Polymorphism when incoming data sets do not match



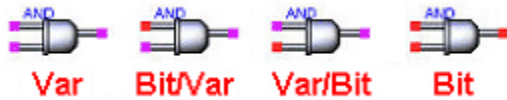
When you wrap this behavior up as an object, select the **Create New Object** radio button so that the behavior of the original object is not overwritten. This will create a third AND object with an interchangeable footprint. In this object, only recurse on the bottom input, and simply propagate the top input through the recursion. If you were to place a Bit input and a Nibble input on the new AND, the following structure would be synthesized:

Synthesized structure with incoming data sets that do not match



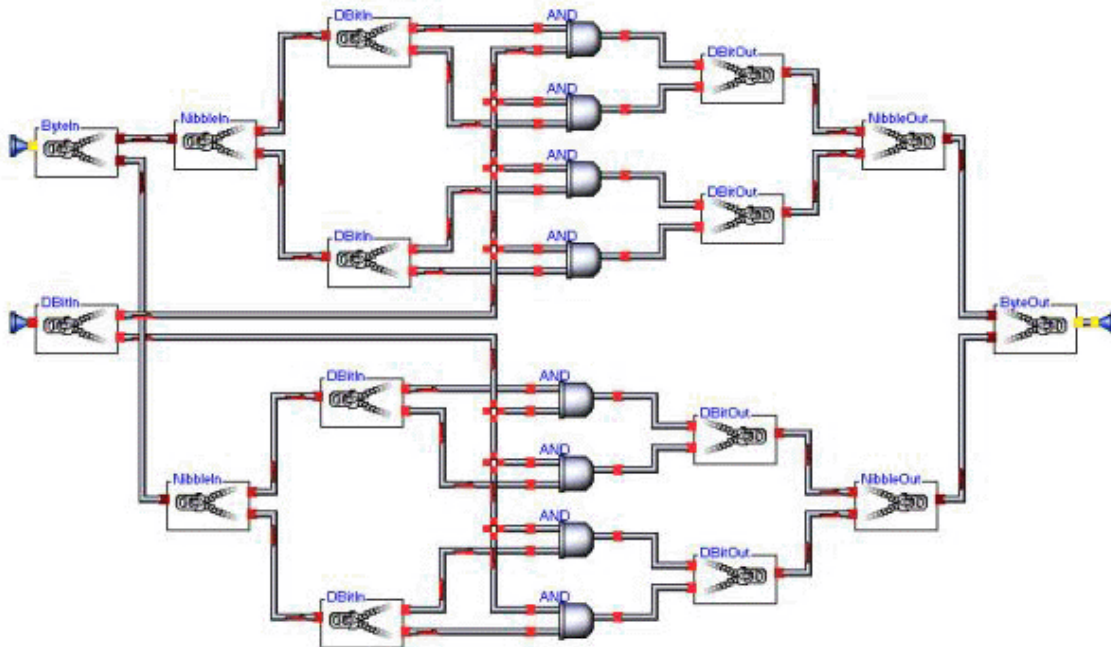
It is preferable to use the most variant (two variant inputs) version of the AND. If used in the preceding case, the most variant AND would immediately be replaced in the first stage of recursion. If you make another AND object with the Bit and Variant inputs reversed, you may use the most variant AND object and do not have to remember whether the Bit goes on the top or the bottom. If you switched the inputs and created a fourth AND object, you would have the following ANDs that share the same footprint:

ANDs that share same footprint



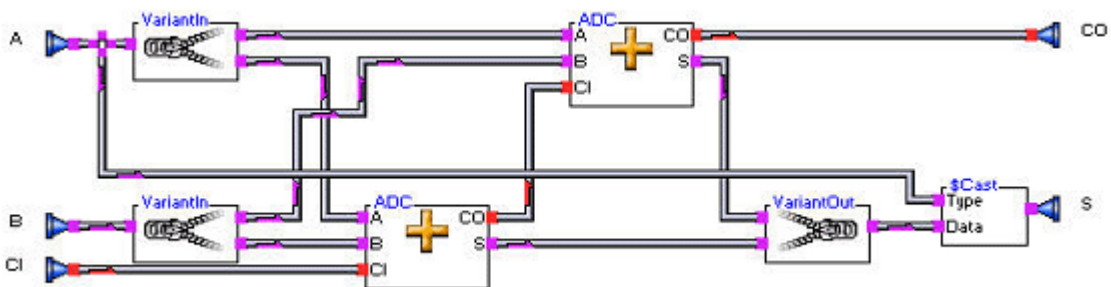
If you use the most variant AND, you no longer have to worry about the inputs carrying the same data set. If you placed a Byte and a DBit on the inputs of the most variant AND, it would generate the following structure:

Addition of a Byte and a DBit

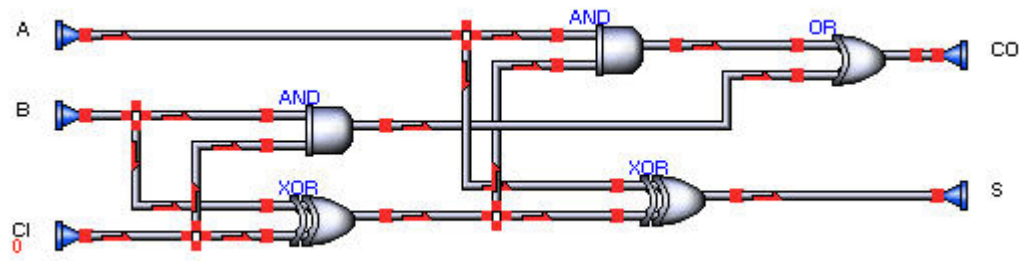


Usually when you are creating data set polymorphic objects, you will program a variant version and a bit-level resolution version. In this example, the recursion terminating leaf-node is a Primitive Object and did not need to be explicitly programmed. By programming a variant object and a bit-level object you may generally cover all the cases you need. Below are the two objects necessary to synthesize a data set polymorphic ADC (add with carry):

Variant ADC



Bit ADC



Chapter 13: Synthesizer Graphics Display

Synthesizer Graphics Display

The Synthesizer Graphics Display is presented during the compilation phase of a Viva program. The items depicted in the display consist of the individual components of the portion of the program being compiled. Pressing the arrows in the control bar with the mouse can alter the size of the objects.

Note: By reducing the size of the objects or hiding the synthesizer graphics display, you can decrease the compilation time.

The colors of these Objects are as follows:

Object colors in the Synthesizer Graphics Display

Object	Color
High Level Objects	Yellow
Primitive Objects:	
Transport	Red
AND	Blue
OR	Lime
INVERT	White
ASSIGN	Aqua
INPUT	Teal
OUTPUT	Maroon
Other	Olive

Chapter 14: Syncing

Syncing

As is common in hardware systems, most sub-modules do not complete on the same clock cycle. Therefore, if two modules (A and B) run parallel so that both output to a common third module (C), you must ensure A and B complete before attempting C. Do not AND the Done signals from A and B since they have little chance of happening on the same cycle; in that situation, the AND will block all Done signals. Instead, use the Sync object.

The SyncClr object will take as an input a list of Done signals and produce from those a single Done pulse when all the Done signals included in the list have fired. It returns a list of Busy signals formatted to the same list structure of Done signals coming in. You can use the ListOut objects to combine your signals for the Done and the ListIn objects to split the Busy signals.

Syncs are built into what is called a SyncRegClr, and SyncRegPair. SyncRegClrs are built into most top-level operators such as Add and Mul, which is why the variant Done and Busy nodes are on most top-level objects.

Sync and SyncRegister

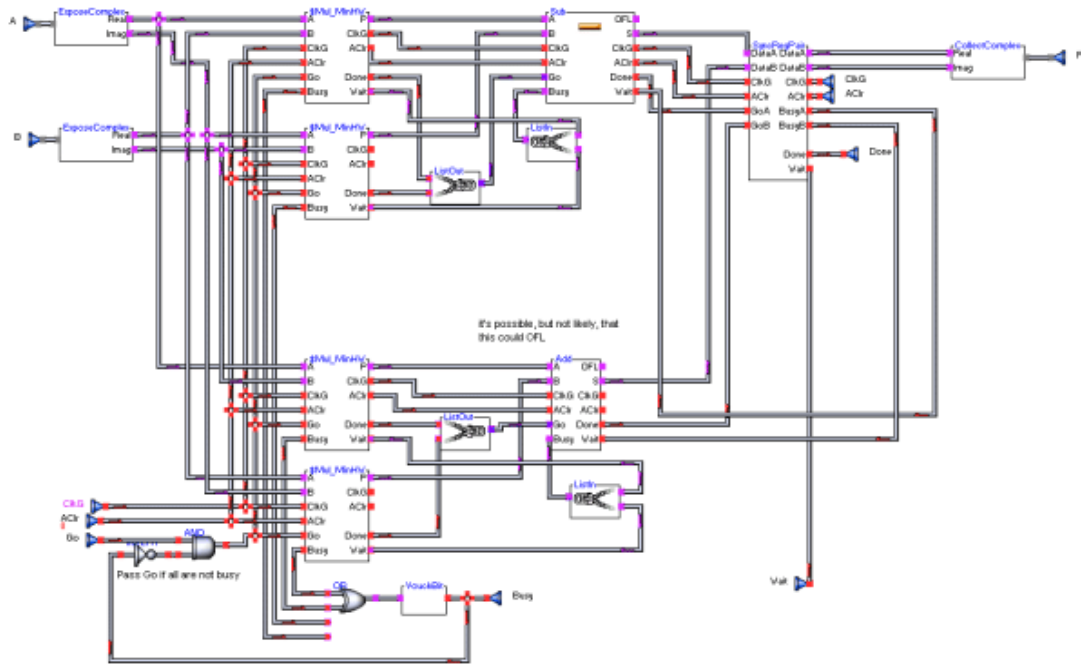


The \$Mul_MinHW (Complex) provides a good example of Synchronizing objects.

Synchronizing objects example

\$Mul_MinHW (Complex)

$$A[6] * B[6] = A[6] + A[6] + A[6] + A[6] + A[6] + A[6] + A[6] + A[6]$$



Note the following:

- The Busy signals coming off the first group of parallel objects are OR'ed together and before output. That lets objects upstream know that at least one \$Mul object is busy; therefore, you don't want their data at the moment. It's tempting to pack those into a list but you've only got one Go signal; there is no value in individual Busy signals in this case.
- Remember the Add and Sub have Sync objects built into them, which is why you can plug lists into the Go signals on them.
- Notice the Busy signals are split the same way the Done signals are combined.
- You will see the Sync object at the end. This forces both the Add and Sub to have completed before the data at P is ready. The incoming wait is plugged in there as well, forcing the Add and Sub to output the same data when high and forcing the whole object to propagate a stall.

Meeting Timing Constraints

If you have five or six GDBW objects connected together, timing constraints may not be met. On long GDBW chains Busy-to-Wait signals are asynchronous and not registered. Use the RegOnBusy object to register the Busy-to-Wait signal every 5-6 stages of the computational pipeline, or when not meeting timing constraints. The RegOnBusy object will pass through the Wait-to-Busy signal unless Waiting and receive a Go signal at which time it will register the data and propagate the Wait signal to a Busy.

Chapter 15: Memory Objects

On-Chip Memory Objects

Objects you would use to implement on-chip memory are located in the Memory sub tree of CoreLib and include the RAM, Queue, and ROM objects.

RAM

A RAM object implements the on-chip memory. In the x86 system, this object uses the Viva Standard COM objects to allocate system memory.

Inputs

A RAM object has the following inputs: Data, Address, Read, Write, Go, and Wait.

RAM Inputs	Description
Data and Address	The allocation size in bytes is the bytelength of the data times two to the bitlength of the address.
Read and Write	Read and Write are mutually exclusive flags that determine the operation you will perform. They both cannot be set; that is, they are not bi-directional.
Go	Go does not need to be a pulse. As long as a Go is held high, RAM is enabled for Reading or Writing.
Wait	If Wait is set, the signal passes through directly to Busy.

Outputs

A RAM object has the following outputs: Data Output (O) and Done.

RAM Outputs	Description
Data Output (O)	Data Output is the data stored at the address location when Read is asserted.
Done	Done is always kept high as long as Read and Go are asserted and Wait is not asserted. For example, if you assert Read and then assert Go, Done will be asserted one clock cycle later.
OFL	Set to 1 if queue fills up (overflows).

Queue

If the Queue contains data, it triggers the dequeue off a not Wait state. That is, for each clock cycle, a data element is pulled off the Queue in the order it was put in the Queue, as long as Wait is low. Then, a Done is fired for each valid data element pulled off the Queue.

Inputs

A Queue object has the following inputs: DataIn, Size, Reset, Go, and Wait.

Queue Inputs	Description
DataIn and Size	The allocation size of the Queue in bytes is the bytelength of the data (DataIn) times two to the bitlength of the Size.
Reset	Reset clears the Queue and resets the Queue to an empty state.
Go	Data is put on the Queue on every clock cycle that Go is held high.
Wait	For each clock cycle, a data element is pulled off the Queue in the order it was put in the Queue, as long as Wait is low.

Outputs

A Queue object has the following outputs: DataOut and Done.

Queue Outputs	Description
DataOut	DataOut is the currently de-queued data element.
Done	Done is fired for each valid data element pulled off the Queue.

Examples of using on-chip memory objects

There are documented examples of using on-chip memory objects. The examples are located in the Memory subtree of CoreLib and include the exRAM, exQueue, and exROM objects. To see the examples, open the Examples folder on the Memory subtree of CoreLib and double-click the Memory_Examples and follow the instructions on the sheet.

Emulated RAM Objects

Emulated RAM objects work within the confines of the Go, Done, Busy, Wait protocol. However, unlike hardware block RAM, RAM objects are not single-clock latent. Every command to an emulated RAM object depends on the dispatch and return from one or more COM objects. Therefore, you must use the Busy signal when using RAM objects.

While you can issue read or write commands every clock cycle in hardware, when emulating, you can only issue a subsequent command after the Busy signal from the first command has gone low.

Chapter 16: Host to FPGA Communication

Behavioral Communications System (BCS)

Communication between systems occurs through a series of communications systems called the Behavioral Communication System (BCS). A simple example would be an input horn and an output horn connected to an INVERT object that is in system PE1. The input and output horns on the top-level sheet automatically get assigned to the host GUI system, and widgets are placed on a runtime form for them. The input widget data goes through the BCS to the hardware (PE1 system), which does the invert, then back through the BCS to the output widget. Any time you use a system that resides on the host CPU, this requires Viva.exe at runtime.

Chapter 17: COM

COM

Component Object Model (COM) is an operating system-wide convention for code exposure. It is a sort of "universal interface." Code written in any language can be exposed using COM and used in any language that uses the COM interface. OLE, ActiveX, COM+, and DCOM are all derived from COM.

Popular programming languages have proprietary systems for using COM components via the COM interface and for creating COM components. While Viva presently does not have the ability to create COM components, it does enable you to use existing COM components, including OLE and ActiveX controls.

The functionality of COM components is implemented in Dynamic Link Libraries (DLL files) and ActiveX Control files (.OCX files). These files contain a "type library." A type library contains one or more COM objects (perhaps better known as Classes), ActiveX controls, data type definitions, and/or enumerations. To use one of these files, it must be registered on your system, whether you are developing or executing them.

Note: For further information regarding COM, Star Bridge recommends you visit <http://msdn.microsoft.com/library/default.asp> and review the material under the heading Component Development.

Registering DLL/OCX

To make a type library usable, you must register it on your system. To register a DLL or OCX file on your system, you must run the program "Regsvr32.exe" located in the Windows/System directory. The path and filename of the file that you want registered is passed a parameter on the command line. For example:

```
C:\regsvr32.exe c:\viva\vivasystem\VivaAtom.dll
```

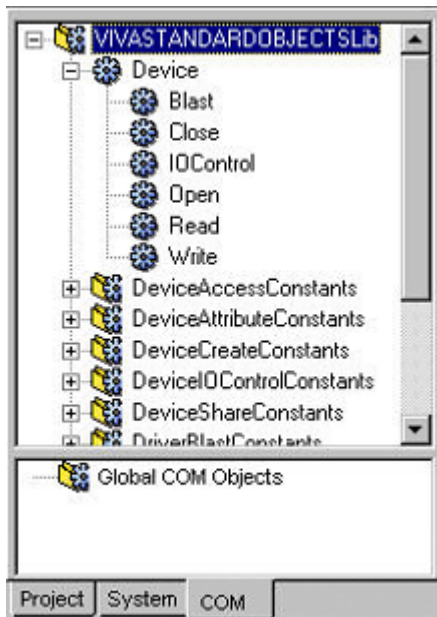
You can accomplish the same thing by dragging the icon of the DLL or OCX file over the Regsvr32 icon.

COM and Viva

To view a list of existing type libraries on your system and the components and definitions that they contain, select the COM component from the Object Browser. To activate the Object Browser, from the **Tools** menu, select **Object Browser**. For example, check the box in front of Viva Standard Objects I/O Type Library.

You can include any of these libraries for use in Viva through the Object Browser window. Once you have included the type libraries containing the components that you want to use, you can instantiate these components and access their member variables, properties, functions, and events. Use of the included COM libraries is done in the same manner as other objects: use the class and object trees found on the COM tab of the object tree pane. These libraries are also loaded by default when creating a new project.

COM Tab/Object Tree



The COM class tree, shown in the top part of the COM Tab/Object Tree figure above, contains the Viva standard object library. Each class and enumeration within that library is shown as a branch of the tree. For example, Device and DeviceAttributeConstants are both branches (classes) of the Viva standard object library, while Open and Read are both enumerations of the Device class.

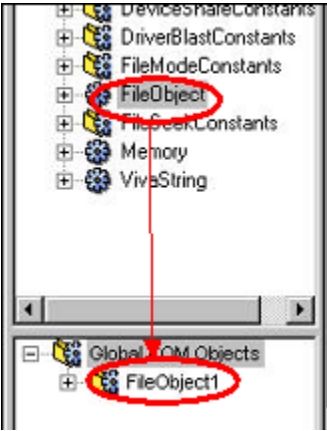
Each branch that represents an enumeration can be used by dragging and dropping it onto an application sheet. Placing an enumeration on an application sheet will instantiate an input having a name and constant value appropriate to the enumeration entry represented, as illustrated in the following example.

COM Object Enumeration

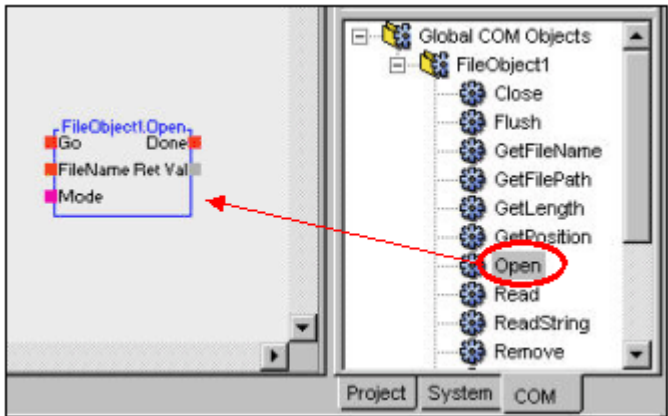


Dragging an object that represents a class onto the COM object tree (bottom part of the right pane) will add an object to the COM object tree that represents a global instance of that class (see the following COM Class Global Instance figure). This new object will contain detailed object members for each of the class's properties, functions, and events. Dragging one of these object members onto the application workspace creates a reference to that member. This is illustrated in the COM Class Member Reference figure. The COM object tree also contains objects for all COM forms and their controls.

COM Class Global Instance



COM Class Member Reference



At runtime, except for events, these members can be accessed by sending a high signal to the 'Go' node of the member. The behavior of these members is listed in the following table.

COM Class Member Behaviors

Member	Behavior
Properties	The present value of the variable is sent out the output node named "Output". A signal sent to the node named

node named "Current". Any value sent to the node named "New" will then become the new value of the variable.

Functions Each input node, other than Go, if any, serves as a parameter to the function. If the function has a return value, then it will be sent out the second output node before the Done node fires. If any node corresponding to a non-optional parameter is not hooked up, then an exception will be thrown when the function is invoked.

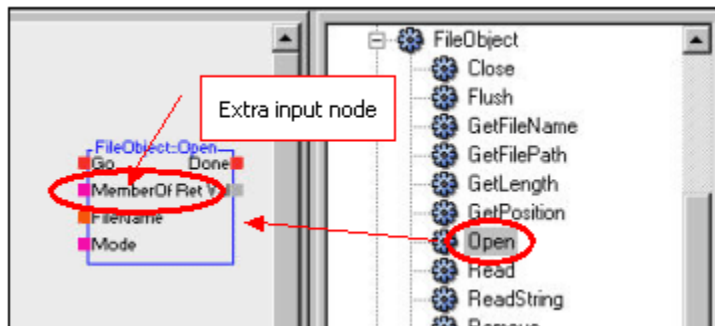
Events Each output node, other than the Done node, is a parameter passed with the event. When a COM event fires in Viva, all events generated on attached transports are forced to process immediately. Awareness of this information will enable you to avoid infinite loops.

Once a member has completed its execution, the 'Done' node sends out a high state. Also, sending a low state to the 'Go' node will cause the member to send a low state from the 'Done' node.

Dynamic COM Objects

You can refer to dynamic COM members by dragging an object representing a member from the COM class tree (upper part of the right pane) rather than the COM object tree. Dynamic COM members function in the same manner as regular object members, except that they are not associated with a specific instance of a class. These objects will have an extra input node called 'Member Of' that is used to specify to which object they apply.

Dynamic COM class object

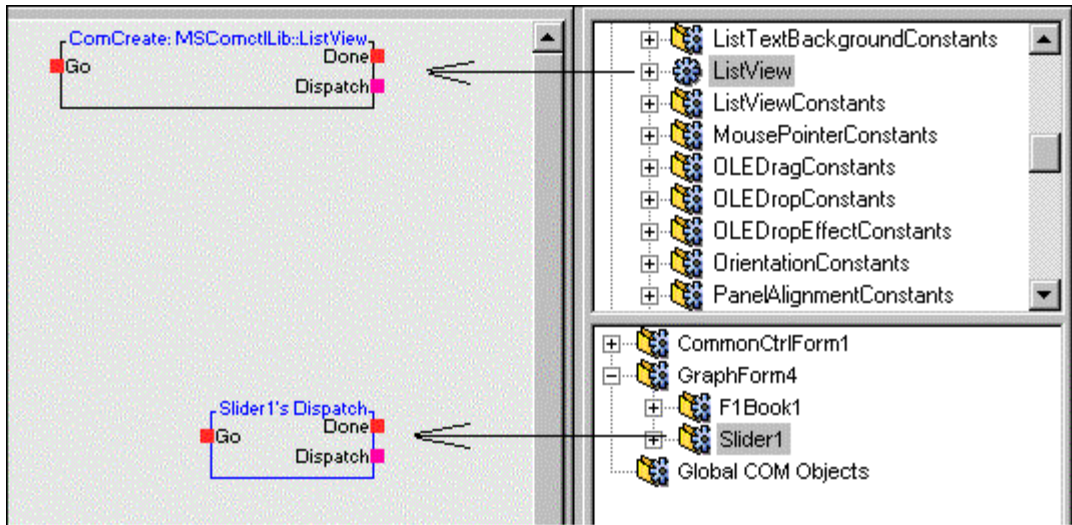


The input to this type of node is a pointer to a class specific 'Dispatch' pertaining to the class to which the object belongs.

To acquire the dispatch pointer of an existing COM object, use a Dispatch object, which you can generate by dragging a node representing a COM object in the COM object tree onto the application editor. Once given a 'Go' signal, it will send the dispatch pointer out the node named 'Dispatch'.

You can also instantiate COM objects at runtime using a ComCreate object. Dragging a class object representing a COM class from the COM class tree onto the application editor can generate a ComCreate object. Note the name of the newly created object: "ComCreate: ", then the name or Library ID for the library that the class resides in, then "::", followed by the class name. The following figure illustrates how the class object will appear on a Viva sheet.

COM class objects example



COM objects created at runtime are not automatically released. They can outlive their calling program, and even be used in other processes. You must use a Release object to destroy them. This object is located in the Primitive Objects branch of the Project objects tree. Simply pass in the dispatch pointer and send it a 'Go' signal.

COM Object Destruction



Object Browser

The Object Browser, available from the Tools menu, contains a list of existing type libraries on your system. These libraries also contain lists of the components and definitions included in each library.

The 'Libraries' list box contains the list of type libraries registered on your system. If you select one of these items, the list of its components and definitions appears in the 'Classes and Types' list box. If you select one of these, the list of its members appears in the 'Members' list box.

You can include a Library for use in Viva by checking its entry in the Libraries list box. Simply click the check box next to the desired library to include it. The library can likewise be un-included by clearing the check box.

Note: You must click OK to apply any changes you make to the selection, or de-selection, of any type libraries.

The caption immediately below the Libraries list box is the path and filename of the actual file that contains the Library, and the caption below that is the Library ID, if it exists, for the selected library. It is used in Viva save files, prefixed to a class name, to indicate that it is located in this library.

The caption immediately below the Classes and Types list box is simply the type of the selected entry, followed by its name while the caption below that is the Windows Programmatic ID (not to be confused with Viva Programmatic ID, used in save files), if it exists, for any class selected. If you are not already familiar with programmatic IDs, know that any class that does not have a Windows Programmatic ID cannot be directly instantiated via the COM interface.

COM Form Designer

The COM Form Designer, available from the Tools menu, is a utility that is used to create forms that host ActiveX Controls. The COM Form Designer File menu contains New, Open, Save, and Save As commands, along with a recently used file history.

The forms tree in the top pane contains a list of all forms in the current project. **By default, there are none.** Selecting the File/New menu command will create a new form. Clicking on a branch in this tree will display its corresponding form. Pressing the delete key will close the form corresponding to the selected branch.

The ActiveX controls tree in the bottom pane contains a list of available ActiveX controls for the libraries that have been selected for use in this project using the Object Browser. Dragging one of these branches onto an ActiveX form will instantiate that control at the location specified. The control can then be resized or deleted. To delete the control, simply select it by clicking it and then pressing the delete key.

Right-clicking a control brings up an available actions list for the selected control. This list may or may not contain entries.

When a Viva file is saved, any forms that are dependant on that file are also saved, even if they did not originate from the current project.

All forms and controls are listed in and can be used from the COM object tree.

Creating COM Components with Microsoft Visual C++ 6

Creating Simple COM Components

This section explains how to create a COM library (DLL), create a simple COM component within that library, and add functions and an event to the COM component.

1. Open a project for generating the DLL. In Microsoft Visual C++, click **File**, then **New....**
2. On the Projects tab, click **ATL COM AppWizard**.
3. Name the project SampleProj1, then click **OK** to invoke the AppWizard for the project.
4. On the interface form, check the **Support MFC** checkbox.
5. Click **Finish**. A dialog box appears that indicates what files it will create for the project. Click **OK**.

You have now created the framework for creating your COM Library and any components that you want to add to it.

6. To add a COM component, switch to the Class View tab in the workspace window.
7. Right-click the top node. From the resulting menu, select **New ATL Object....**
8. From the dialog that appears, select **Simple Object** in the right pane and click **Next**.
9. In the Short Name: field, type MathObject so future references to it will fit in the context of your project.
10. Click the **Attributes** tab. Select the **Support Connection Points** option, which will create a Dispatch Interface that you will use to create an Event for this component. Click **OK**.

One new class and two new “interfaces” are created in your project to support the new component. An interface in this context is the specification and mechanism by which other applications can interact with the component, via COM. This interface can contain functions, properties, and events that you can use to interact with the component.

CMathObject is the class in which the implementation code for this component will be placed, a process discussed later.

IMathObject is the interface used by other applications to invoke functionality of this component. This is known as an “Incoming” interface. Functions and properties can be placed in this interface.

`_IMathObjectEvents` is the interface that can trigger events, which are special functions that are used to talk to other applications. This is known as a “Outgoing” interface. A function can be set up in another application that “listens” for an event from a component, and is called whenever the component triggers the event.

To add a function to the component, expand the `CMathObject` node. Notice that it has a node named “`ImathObject`,” which is the same name as the node for the `IMathObject` interface documented previously. The `CMathObject` node, subordinate to the `CMathObject` class node, is for the Interface implementation. The other `IMathObject` node is for the interface definition. The meaning and use of this is described later.

Adding methods to COM components

To add a method, right-click the `ImathObject` node that is subordinate to the `CMathObject` class node, and select **Add Method...** from the menu that appears.

There is an important difference between doing this and adding a method to the `CMathObject` class: only methods added to the interface implementation for this class will actually be exposed via COM.

A dialog appears asking you to enter a function name and a parameter list. Note the following about functions on interfaces:

- Do not give the function a name that is ambiguous with any identifier in a C++ library that you have included.
- The valid parameter types for COM in Visual C++ are shown in the following table.

Parameter types for COM in Visual C++

Data Type	Description	Default sign
<code>VARIANT_BOOL</code>	COM standard 32-bit Boolean type. false = 0; true = -1	unsigned
<code>byte</code>	8 bits	(not applicable)
<code>double</code>	64-bit floating point number	(not applicable)
<code>error_status_t</code>	32-bit unsigned integer for returning status values for error handling.	unsigned
<code>float</code>	32-bit floating point number	(not applicable)
<code>handle_t</code>	primitive handle type for binding	(not applicable)
<code>hyper</code>	64-bit integer	signed
<code>int</code>	32-bit integer. On 16-bit platforms, cannot appear in remote functions without a size qualifier such as short, small, long or hyper.	signed

long	32-bit integer	signed
short	16-bit integer	signed
small	8-bit integer	signed
wchar_t	16-bit predefined type for wide characters.	unsigned
BSTR	COM standard string type. Pointer to type wchar_t.	(not applicable)
VARIANT	Container of any COM supported type. Wrapper class is _variant_t	(not applicable)

Parameter types can also be a pointer to any of the types listed in the previous table.

For example, type *Add* in the Method Name field and enter the Parameters as shown below:

```
int param1, int param2, [out, retval] int *RetVal
```

The [out, retval] prefix to the last parameter is the marker that it is the return value for the function. This can only be applied to the last parameter for a method, and must be a pointer to the type that you want to return from the method.

Click **OK**. To get the Visual C++ IDE to update the workspace window to reflect the changes, save all files, close your workspace, and re-open it.

After forcing an update, notice that new nodes have been added subordinate to both the interface implementation node and the interface definition node. Double-click the new node that is subordinate to the IMathObject interface implementation node to view the code that implements the new function.

The function body has an AFX_MANAGE_STATE macro and a statement to return the status code S_OK.

Do not remove the AFX_MANAGE_STATE macro; it performs an action that must be completed at the beginning of this function call. Also, it should always be the first line of code in this function body.

Do not remove the other line either; there is no need to return any code other than S_OK unless an error occurred. There are many status codes that can be returned, but as a general rule, if you encounter an error condition, call the Error function, which is a member of CMathObject, which passes it a string that describes the problem and returns DISP_E_EXCEPTION. Most COM-utilizing programming environments throw an exception when this code is returned with the message passed in as a parameter to the Error function.

For example, add the following line in place of the “TODO” comment:

```
*RetVal = param1 + param2;
```

The RetVal parameter passed in by reference holds the value that will be returned from this function to program using this function. Do not confuse this with the status code returned. The S_OK is for the OLE automation system to notify you to proceed as intended. The program calling this function does not see this. However, the calling program would be notified, if you were to return some error code, such as DISP_E_EXCEPTION.

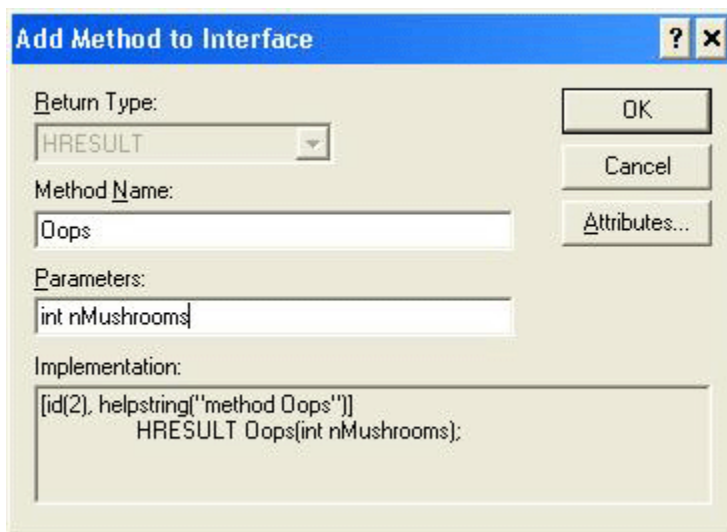
The function body should now read as follows:

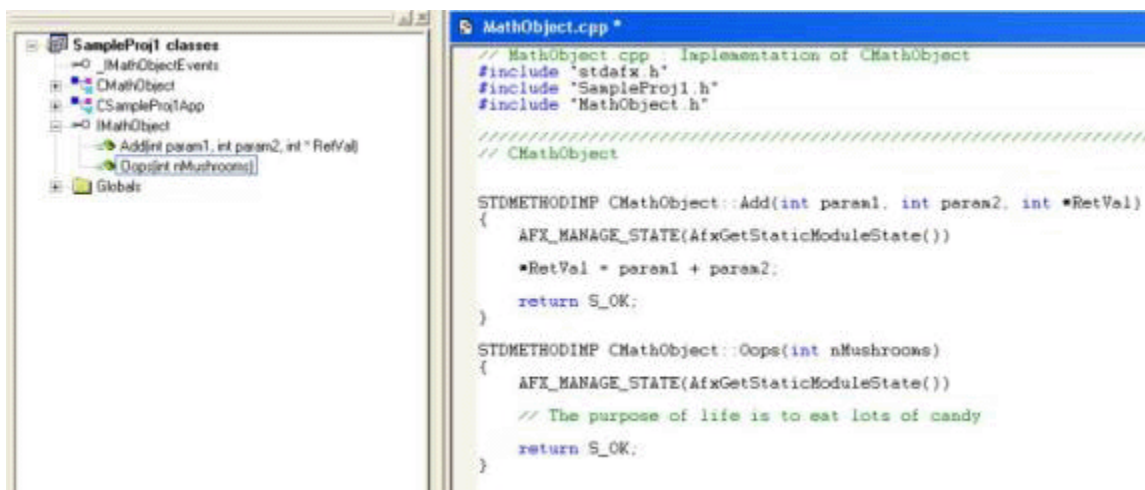
```
STDMETHODIMP CMathObject::Add(int param1, int param2, int *RetVal)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState())
    *RetVal = param1 + param2;
    return S_OK;
}
```

This function will, when called, simply add the numbers passed in by the first two parameters and set the value pointed to by RetVal to it, returning it to the calling process.

The following figures illustrate adding a new function to the IMathObject interface of this component.

Adding a new function to the IMathObject interface





If, in the process of changing the name of the function, you alter the entire project beyond recognition of the Visual C++ AppWizard, delete the bad function and start over. Use the following procedure to remove a function from a component.

Removing a Function

Right-click the node under the interface implementation (not definition), and select **Delete** from the menu that appears.

A dialog will appear to explain that the function body will be commented out, rather than deleted, and you will be asked to confirm the deletion of the function. Click **Yes**. You must manually delete the function body to permanently remove it.

Note that the function still exists under the interface definition and there is no option to delete it.

The interface definition links to the IDL file associated with the COM Type Library rather than the C++ code that implements its functionality. An IDL file describes the Type Library, including all of its dependencies, components, type definitions, and interfaces, including each of their members (functions, properties, and events).

When you need to change an interface beyond what the AppWizard handles, you must edit the IDL file. Double-click the incorrect node to bring up the IDL file. You can also access the IDL file by clicking the **File View** tab in the Project window.

To remove the incorrect function from the interface definition, simply locate and remove the line of code defining it. The IMathObject interface definition should now read as follows:

```
Interface IMathObject : IDispatch
{
    [id(1), helpstring(*method Add")] HRESULT Add(int param1, int
param2, [out, retval] int
*RetVal);
};
```

Save all files before the tree in the Project window will be updated to reflect this change. Then, select **Build**, then **Rebuild All** to compile the Type Library. This will both build and register the DLL and the COM Type Library will be complete.

Building an Events

Any function that is added to the `_IMathObjectEvents` interface can be called by this component, not by a program using this component. The program that this component resides in will experience and event when this component calls such a function.

Events are normally used to notify the process hosting a component that a pre-determined condition has been met on behalf of the component.

The following example shows how to build an event that can call into another program that is set up to deal with it, and actually get results back from that program. This assumes that the program does its job.

`IMathObject` has both an interface definition and an interface implementation, whereas `_IMathObjectEvents` has only the former because the `CMathObject` class hasn't been told to implement this interface yet. This process was done automatically for `ImathObject` but must be done manually for `_IMathObjectEvents`.

1. To create an implementation for this interface in `CMathObject`, right-click its node in the Project window, and select **Implement Connection Point** from the resulting menu.
2. Check the **_ImathObjectEvents** box, then click **OK**.

This creates a new class: `CProxy_IMathObjectEvents`. This class will contain the implementation code for the `IMathObjectEvents` interface, and `CMathObject` can directly use its members because `CMathObject` now inherits additionally from it.

You must repeat this process every time that you change the `IMathObjectEvents` interface to get the implementation to update.

3. Add a function to the `_IMathObjectEvents` interface in the same manner as you added the `Add` function to the `IMathObject` interface.
4. Type *ItsYourTurn* in the Method Name field and enter the Parameters as shown below:

```
int *Data
```

The parameter is passed by reference. This is so that the process that receives this event can modify the argument passed, which allows you to call into the hosting application and receive data back from it. You will then have two-way communication with the host.

Tell `CMathObject` to re-implement `_IMathObjectEvents`.

This creates a now have a `Fire_ItsYourTurn(INT *Data)` node subordinate to the `CProxy_IMathObjectEvents` class node. It represents the implementation of the `ItsYourTurn` event.

5. Add another function to `IMathObject` to invoke the event. Define the function by typing *Interact* in the Method Name field and entering the Parameters as shown below:

```
int Data, [out, retval] int *RetVal
```

Go to the function's C++ implementation code by, once again, double-clicking the node that represents it, subordinate to the `IMathObject` node.

6. Change the function body of `Interact` to look like this, then rebuild.

```
STDMETHODIMP CMathObject::Interact(int Data, int *RetVal)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState())
    // Invoke the event!
    Fire_ItsYourTurn(&Data);
    // Let's display Data after our hosting application has had a
    // chance to process it. What did it do to Data? It could do
    // anything; the important thing is that it's two-way
    interaction.
    char msg[20];
    itoa(Data, msg, 10);
    AfxMessageBox(msg);
    *RetVal = Data;
    return S_OK;
}
```

The warning indicates that the parameter you passed as a pointer is going to be converted to a `[bool]` so the address of `Data` will be either `0x00000000` or `0x00000001`. Where the code goes to stuff `Data` into a variant, to conform to COM dispatch protocol, there is no `[int*]` override in the Variant class. Thus, the compiler do coerces your `[int*]` into a `[bool]`.

7. Change the cast circled below, then rebuild.

```

template <class T>
class CProxy_IMathObjectEvents : public IConnectionPointImpl<T, &DIID__IMathObjectEvents, T>
{
    //Warning this class may be recreated by the wizard.
public:
    HRESULT Fire_ItsYourTurn(INT * Data)
    {
        CComVariant varResult;
        T* pT = static_cast<T*>(this);
        int nConnectionIndex;
        CComVariant* pvars = new CComVariant[1];
        int nConnections = m_vec.GetSize();

        for (nConnectionIndex = 0; nConnectionIndex < nConnections; nConnectionIndex++)
        {
            pT->Lock();
            CComPtr<IUnknown> sp = m_vec.GetAt(nConnectionIndex);
            pT->Unlock();
            IDispatch* pDispatch = reinterpret_cast<IDispatch*>(sp.p);
            if (pDispatch != NULL)
            {
                VariantClear(&varResult);
                pvars[0] = ((int))Data;
                DISPPARAMS disp = { pvars, NULL, 1, 0 };
                pDispatch->Invoke(0x1, IID_NULL, LOCALE_USER_DEFAULT, DISPATCH_METHOD, &varResult, &disp, NULL, NULL);
            }
        }
        delete[] pvars;
        return varResult.scode;
    }
};

```

This time, you should not receive any warnings. Do not worry that the type cast to is signed, whereas an address is unsigned. All the bits will still be there.

The DLL that you have created cannot be used on another machine that does not also have Microsoft Visual C++ 6 installed because the active build configuration is set to Win32 Debug, which is the default. To fix this problem, select **Build**, then **Set Active Configuration**. Select **SampleProj1 – Win32 Release MinDependency**, then click **OK**.

When you rebuild, you will have a self-sufficient DLL that can run on any machine that has Windows 95 or later running on it.

For a listing of tutorials on creating COM/ActiveX components in Visual C++ and other languages, visit <http://www.guruischool.com/index.jsp>. For help on creating ActiveX controls in Visual Basic, visit <http://msdn.microsoft.com/vbasic/techinfo/training/activex.asp>.

Index

A

Adding methods.....	136
Ambiguous objects	20

B

BCS	124
Behavior definition	90
Behavioral Communication System.....	124
BIN.....	64
Bit pattern data sets.....	64
Building events	140
Busy	100

C

Casting data.....	68
Collectors.....	63

COM

Adding methods	136
Building events.....	140
Components Creating	135
Dynamic objects.....	131
Form Designer	134
Registering DLL/OCX	127
Removing a function	139
COM	126, 128
Compile issues	82

Consistent structure	71
----------------------------	----

Constraints.....	118
------------------	-----

Converting sheets	28
-------------------------	----

Creating

COM components	135
Executables.....	75
Custom data sets	65

D

Data casting	68
--------------------	----

Data Set

Creating.....	65
definition (text base file)	90
Editor	65
Polymorphism.....	107
Recursion	69

Debugging horn	82
----------------------	----

design	71
--------------	----

Directory System Attributes	83
-----------------------------------	----

DLL/OCX

Registering	127
DLL/OCX.....	127

Done	99
------------	----

Dynamic COM objects	131
---------------------------	-----

E

EDIF File support.....88

Emulated RAM objects122

Executables

 Creating.....75

Executables75

Exposers.....63

F

FPGA

 X86 executor differences59

FPGA59

G

GDBW_Clr Object.....102

Go98

Go Done Busy Wait96, 97

I

Implementing \$Spawn.....86

L

Library

 Library Management19

 Updating versions24

List67

LSB64

M

Memory objects120, 122

MSB 64

N

Navigating sheets 30

O

Object

 Attributes 42

 Attributes with special meanings..... 51

 Browser 133

 Definition (text base file)..... 90

 Footprints 37

 On-chip memory..... 120

 Polymorphism..... 39

 Prototype 90

 Viewing detail 53

On-chip memory object..... 120

Overloading..... 23, 105

P

Preferences..... 5

R

Recursion..... 104

Recursive footprints 107

Resource usage..... 76

Runtime dialog preferences..... 81

S

Sample project 9

Sheets		Tree structures.....	15
Converting to objects	28	Troubleshooting	78, 82
Navigating	30	U	
Viewing.....	31	Underloading.....	23
Static data sets	70	V	
Synching	116	Viva projects	
Synthesizer graphics display	114	Attributes	58
System definition (text base file).....	90	W	
T		Wait.....	96, 101
Timing constraints.....	118	X	
Tree structures		X86 versus FPGA applications.....	59
Managing Viva Trees	15		

