VISTA: A Visualization Tool for Computer Architects

by

Aaron D. Mihalik

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 30, 2004

Author_____

Department of Electrical Engineering and Computer Science
January 30, 2004

Certified by_____

Krste Asanović
Thesis Supervisor

Accepted by_____

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

VISTA: A Visualization Tool for Computer Architects

by

Aaron D. Mihalik

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and Computer Science

Abstract

As computer architectures continue to grow in complexity, software developers and hardware engineers cope with the increasing complexity by developing proprietary applications, simulations and tool sets to understand the behavior of these complex systems. Although the field of information visualization is leading to powerful applications in many areas, information visualization applications for computer architecture development are either tightly coupled with a specific architecture or target a wide range of computer system data.

This thesis introduces the Visualization Tool for Computer Architects (VISTA) Environment. The VISTA Environment is an extensible and modular information visualization environment for hardware engineers, software developers and educators to visualize data from a variety of computer architecture simulations at different levels of abstraction. The VISTA Environment leverages common attributes in simulation data, computer architecture visualizations, and computer architecture development methods to create a powerful information visualization environment to aid in designing, understanding and communicating complex computer architectures.

Thesis Supervisor: Krste Asanović
Title: Associate Professor of Computer Science

Acknowledgments

The VISTA project was initially created by Mathew Jack and Chris Batten in 2002. Mathew Jack, a student at Cambridge University, created a prototype version of VISTA while he studied at MIT during the 2002-2003 academic year. Chris Batten continued to work on VISTA until Krste Asanović invited me to work on the project in the Fall of 2003. Thanks to Krste, I was able to work on this unique project in computer architecture.

Throughout the development of VISTA, Chris Batten was always there to offer me advice, encouragement and inspiration. Although the "firehose" metaphor is overused, I cannot think of anything more appropriate: getting some good ideas from Chris Batten is like trying to get a drink of water from a firehose.

There are also a two friends I'd like to thank. Throughout my time at MIT, I have run into Goutam Reddy numerous times and he's always been a great friend. This past year and half was no exception as dragged me through my classes and kept me line whenever I wandered. Also, I am indebt to Karen Robinson both financially and emotionally. Whenever Goutam left off, Karen Robinson picked up.

Finally, I'd like to thank my family. My parents have provided me with numerous unique opportunities throughout my life, and they have always been there to support my endeavors. My younger brother, Adam, remains my best friend and a hell of a role model. And last, I thank my little sister Emily (*Em Emers Emilinie*). She is so wise and so funny.

Table of Contents

Table of Figures

Chapter 1:

Introduction

The internal workings of modern computer architectures are well abstracted away from the typical computer user. While computer users chat in real-time over the Internet, listen to a set of their favorite songs, and watch recorded television episodes on their desktop machine, the computer processor, which took thousands of man-years to design, is quietly orchestrating the complex set of operations required to seamlessly multitask these applications.

Although the typical computer user might not be aware of the complexities in the modern processor, hardware engineers and software designers are thoroughly acquainted with these issues. These professionals have to design prospective architectures, understand the internal workings of the processor at some level, and discuss with others various issues regarding the processor.

Hardware engineers and software designers have developed an enormous set of applications, tool sets and simulations in order to cope with the complexity of modern computer architectures. This set of tools cover a wide range of aspects in computer architecture development, but fails to provide a common framework to design, understand and communicate computer architecture related material

This is the basis for the Visualization Tool for Computer Architects (VISTA) Environment. The VISTA Environment is an extensible information visualization environment for hardware engineers, software developers and educators to visualize data from a variety of computer architecture simulations at different levels of abstraction. The VISTA Environment leverages common attributes in simulation data, computer architecture visualizations, and computer architecture development methods to create a

flexible and powerful information visualization environment to aid in designing, understanding and communicating complex computer architectures.

1.1 The Microprocessor: A Device of Exponentially Increasing Complexity

The birth of the microprocessor can be traced back to 1958 at a laboratory in Dallas, Texas. It was there that Jack Kilby, a Texas Instruments engineer, borrowed and improvised equipment to build the first two-transistor integrated circuit. This seemingly small piece of work by Kilby laid the foundation for the entire field of microprocessors and microelectronics. Forty-five years later, the microelectronics field that sprung from Kilby's work has evolved to a $200 billion industry that drives a $1 trillion electronic end-equipment market. [1]

Since 1958, the size, complexity and computational power of microprocessors have grown at an exponential rate. In a 1965 article published in Electronics, Gordon Moore, one of the founders of Fairchild Semiconductor and the Intel Corporation, noted that the number of transistors in an integrated circuit was growing at an exponential rate and saw "no reason to believe [that this growth] will not remain nearly constant for at least 10 years" [2]. This observation was immediately termed "Moore's Law" by the press and the name has stuck.

Although Moore was only willing to commit to the exponential growth trend until 1975, his observation has held true for nearly forty years. The Intel x86 family of processors, the predominate computer architecture in the desktop computing world, can provide numerous examples of Moore's Law in action. Two notable processors are the Intel 8088 processor released in 1982 and the Intel Pentium 4 processor released in 2000.

Twenty years after the creation of the first integrated circuit, Intel and IBM launched one of the first lines of home computers. This machine, termed the IBM PC, was released in 1982 and came standard with a monochrome monitor, floppy disk system, and an Intel 8088 processor. Although the machine was fairly primitive compared to today's machines—it could only handle simple word processing applications and business management tasks—the Intel 8088 contained nearly 29,000 transistors [3].

In November of 2000, approximately twenty years after the release of the IBM PC, Intel introduced the Pentium 4 processor. The Pentium 4 was designed to tackle the increasingly complex set of applications that desktop and entry level workstation users demanded. Users wanted a processor capable of multi-tasking an assortment of applications including software to communicate over the Internet with real-time audio and video, video games that rendered sophisticated 3D graphics on the fly, and multimedia applications that incorporated computationally intensive video and audio compression formats. To accomplish this, the Pentium 4 delivered users with a dramatic increase of processing power, but required an enormous amount of processor complexity. The Pentium 4 could execute instructions 5,000 times faster than as its Intel 8088 ancestor [4], but it had grown to use 42 million transistors [3].

The increase in the number of transistors suggests that there are considerable differences between the two processors. As the x86 series of processors evolved from the Intel 8088, Intel hardware engineers implemented a set of complex techniques and sophisticated internal operations to improve the performance of their processors. These techniques and operations, each one involving numerous interactions between components of the processor, included branch prediction and speculative execution of instructions, concurrent out-of-order execution of instructions on multiple functional units and in-order commitment of the results to memory, dynamically renaming registers to remove artificial data and control dependencies, and complex interactions between the processor and memory system. Although these techniques dramatically increase the performance of the processors, the resulting complexity is clear: thousands of man-years of effort are now required to develop a new generation of these sophisticated processors.

1.2 Designing, Understanding and Communicating Complex Computer Architectures

The task of effectively designing, understanding and communicating complex computer architectures becomes more difficult as the complexity of the system increases. In order to cope with the increasing complexity of computer architectures, hardware engineers, software designers and academia develop software simulations that model the architecture at various levels of abstraction.

1.2.1 Simulation Applications: Hardware and Software Development

Simulation applications for hardware and software development are used throughout the design process to cope with the complexity in computer architecture and software design. Simulation users enjoy the freedom to choose or develop simulations that take into account the time they have to run the simulation, the type of information they want to obtain about the system, and the computing environment in which they want to execute the simulation.

Simulations that model the behavior of the architecture are created during the first stages of the development process and are used throughout the life of the processor. In the early stages of development, hardware engineers use behavioral simulations to explore different options in an architecture design space, validate design decisions, and analyze the performance of existing applications on new architectures.

At the same time, software developers use system level behavioral simulations to create and debug software for the prospective architecture. Simulations based on behavioral models allow software designers to optimize their code on a new architecture without having to know the details of the underlying system. Also, these simulations cost less than the actual hardware and are available before the hardware is produced.

Simulation Level	Simulation Components	Design Use	Simulation rate on host machine
Instruction-set architecture (ISA)	Processor, memory	Hardware Engineers: explore design space, validate decisions, and analyze existing applications	$> 10^6$ cycles/second
Complete Machine	Processor, memory, operating system, other hardware components	Software Developers: creating and debugging software without the need for actual hardware	$> 10^3$ cycles/second
Register Transfer (RTL)	Registers, Combinational Circuits	Hardware Engineers: examine data flow between the registers and components	> 10 cycles/second
Gate/Circuit	Gates, Transistors	Hardware Engineers: detailed modeling of for circuit timing and energy consumption	> 1 cycles/second

Figure 1-1: Table of Simulation Applications

Hardware and software developers use simulation applications throughout the design of a new architecture to cope with the complexity in hardware and software design.

After developing an architecture design using the behavioral simulators, hardware engineers implement the functional aspects of the processor in register transfer level

(RTL) simulations. Hardware engineers use a Hardware Descriptive Language (HDL) to specify how the data flows between the registers and how the architecture processes the data. At this stage, the hardware engineers implement and debug complex functional techniques that increase the computational power of the architecture.

Finally, there are instances when a hardware engineer needs more detailed information about the simulation of a processor. For instance, an engineer could require highly accurate energy consumption information about the processor. In these cases, the hardware engineer can use general purpose analog circuit simulators like SPICE to obtain such information. However, the SPICE simulations of a circuit are orders of magnitude slower than a RTL level simulation.

1.2.2 Simulation Applications: Teaching and Communicating Fundamental Ideas and Research

The underlying concepts behind computer architecture designs are difficult to describe, communicate and understand using traditional methods of teaching. Introductory computer architecture classes are typically taught using static visual and textural representations such as diagrams and drawings in lectures, papers, and textbooks. These traditional methods of teaching computer architecture severely limit one's ability to grasp complicated design implementations, computational behavioral patters, and component interactions that take place in modern processors.

These complex interactions require dynamic and interactive simulations in which one can monitor the progression of instructions and data through a system and understand the effects of design changes on a system. Dynamic and interactive simulations allow users to understand the functional components in a design, the interaction between software and the processor, and how hardware and software design decisions are made.

The set of simulation tools used by hardware engineers and software designers are usually not appropriate for students to learn about architecture designs. These feature rich simulation tools are frequently designed for optimum simulation speed of complex design projects and require special training to use. Ease of use, excellent visualization tools and portability are not normally goals for these simulator applications.

The demand for dynamic and interactive teaching material has spurred the development of simulation and visualization environments that are useful for interactive learning of computer architecture concepts. Tools such as RaVi [6], JCachesim [8] and HASE [10] are examples of such tools. RaVi and HASE are general purpose computer architecture environments that allow for simulations and dynamic presentations to be created for web based learning environments. JCachesim provides many of the same functions, but is focused on observing the interaction between a processor and the cache during the execution of a program. JCachesim also includes logging capabilities for use in an online learning environment.

1.3 Traditional Approaches to Simulations and Visualizations

Although the set of applications for simulating and visualizing computer architectures is quite large, most of these applications can be grouped in three different categories: (a) high performance proprietary or customizable simulation engines with text-based trace output, (b) simulation engines coupled with visualization environments or (c) visualization environments that accept data from a variety of different sources.

1.3.1 The SimpleScalar Tool Set

The SimpleScalar Tool Set is the most widely used superscalar processor performance simulator and uses a high performance customizable simulation engine with text-based trace output. Typically, hardware engineers will turn to performance simulators like the SimpleScalar Tool Set in order to explore various architecture designs because these simulators tend to run several orders of magnitude faster than RTL simulations. The SimpleScalar Tool Set is publicly available and extensible.

The SimpleScalar Tool Set allows users to choose from highly detailed (and hence relatively slower) simulations or faster simulations that provide less detail. On a 200-MHz Pentium Pro, the highly detailed version of SimpleScalar can simulate 150,000 machine cycles per second. On the same 200-MHz Pentium Pro, the less detailed version of SimpleScalar can simulate four million machine cycles per second. [9]

The drawback of using the SimpleScalar Tool Set is that the text-based trace output can be a large and complex dataset that is difficult to navigate. In this situation, users typically reduce the data into a more manageable dataset, or spend large amounts of time tediously navigating the dataset. Although reduction can be useful for confirming a hypothesis about an existing system and navigation of a large dataset is possible for a well known system, these techniques are less useful for debugging or understanding a novel system.

1.3.2 The Hierarchical Computer Architecture Design and Simulation Environment

The Hierarchical Computer Architecture Design and Simulation Environment (HASE) is a computer system design, simulation and visualization environment developed at the University of Edinburgh. HASE allows for rapid development and exploration of computer architectures at various levels of abstraction. The environment is designed for both hardware and software designers to create a system, simulate software execution on the system and examine the simulation results in an animation and visualization environment. [10]

Although the HASE simulation system was originally written in C++, it has since been ported to Java and spurred the creation of a number of other tool sets; SimJava, JavaHASE and WebHASE have been created as a result of the HASE Java port. The Java implementation of HASE has allowed for applet development to share simulations and visualizations of systems over the web (see Figure 1-2). Current applets include interactive animations and simulations of the DLX pipeline, scoreboarding, predication, and Tomasulo's Algorithm.

The drawback of the HASE simulation environment is that it is tightly coupled with the visualization system. Although HASE provides users with a number of tools to design, simulate, and share their systems over the Internet, these users are always bound to the discrete-event simulation engine incorporated into HASE.

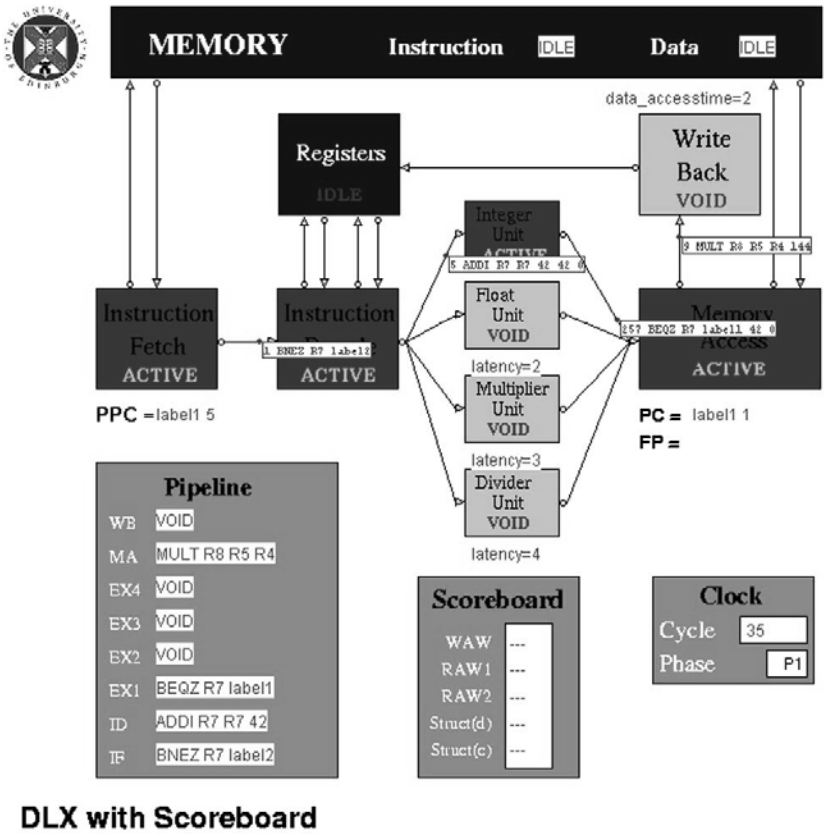


Figure 1-2: HASE Applet Used To Demonstrate A DLX Processor with Scoreboarding

> Currently the Hierarchical Computer Architecture Design and Simulation Environment (HASE) project provides a number of educational simulations available on the web as applets. These simulations include interactive animations and simulations of the DLX pipeline, scoreboarding, predication, and Tomasulo's Algorithm. [11]

1.3.3 The Rivet Visualization Environment

The Rivet Visualization Environment was developed by Robert P. Bosch at Stanford for his PhD work. Rivet can accept data from a variety of data sources and present this information in various different visual displays. Rivet is extensible and allows for rapid prototyping of new visualizations.

In Bosch's dissertation, he details a number of computer related systems that Rivet was used to visualize. During the course of this research, Bosch used Rivet to create visualizations for an interactive parallelizing compiler, a detailed memory system, an application running on superscalar processors (see Figure 1-3), and a real-time

performance of computer systems and clusters [12]. Since then, Rivet has gone beyond computer related systems to visualize multi-dimensional relational databases.

As Bosch showed in his dissertation, Rivet can visualize an assortment of different computer system related information, including computer architecture simulation data. However, the work on Rivet has focused visualizing diverse datasets, not visualizing datasets with common characteristics. For instance, the "PipeCleaner" visualization was created to visualize data from a specific processor simulator and it is not reused to visualize data from other processor simulators.

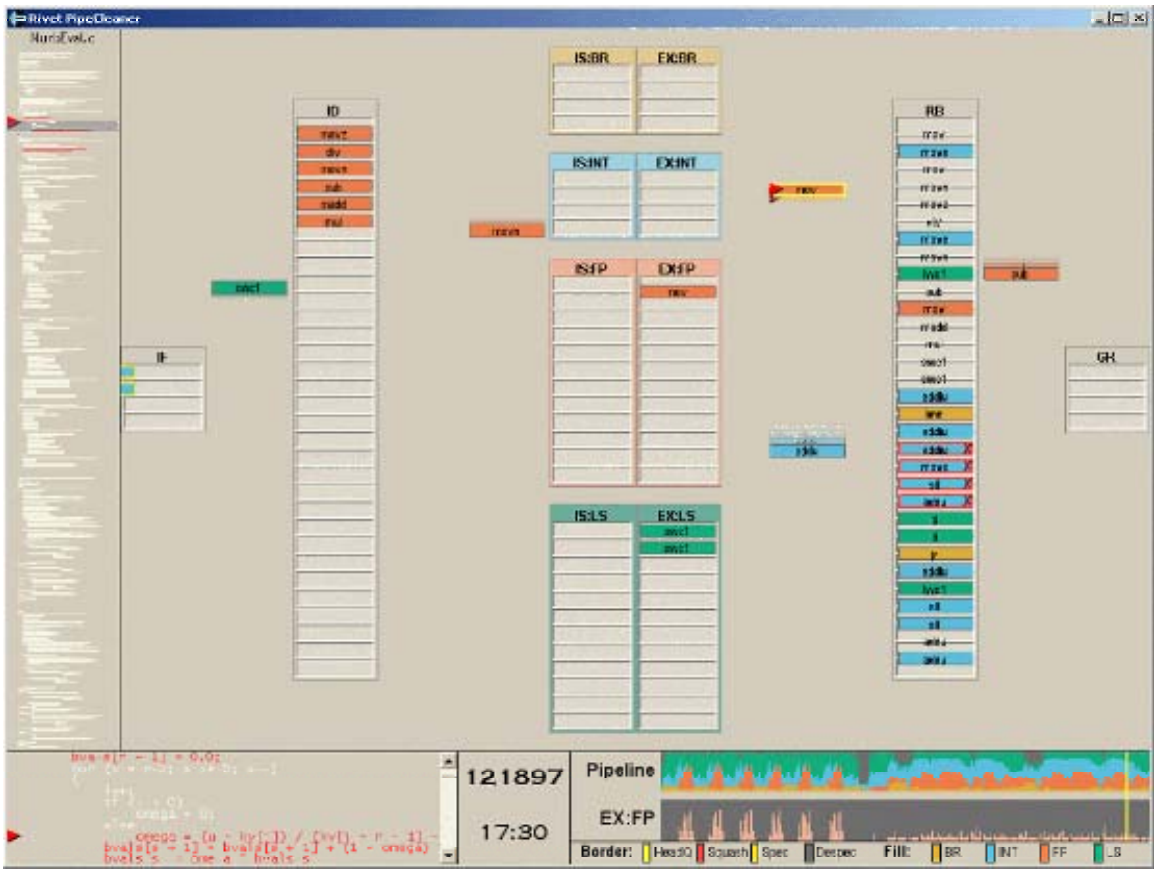


Figure 1-3: Rivet Visualization from the "PipeCleaner" Project

The Rivet Visualization Environment was developed by Robert P. Bosch at Stanford for his PhD work. Rivet is an visualization environment that can accept data from a variety of data sources and present this information in various different visual displays. Rivet is extensible and allows for rapid prototyping of new visualizations. The "PiperCleaner" Project is one such visualization that displays application performance on superscalar processors. [13]

Chapter 2:

The Vista Environment

This chapter presents an overview of the VISTA Environment. The first section enumerates the goals of the VISTA Environment. The next section develops a specification of the abstract data objects from a widely used simulation data format.

Finally, this chapter ends with a design overview of VISTA Environment. This design overview introduces the three sets of components that make up the VISTA Environment and the interfaces that these components implement. Also, this section provides a design overview for each of these three components.

2.1 The VISTA Approach

The VISTA Environment, in its broadest definition, is a software package that enables users to transform rich and detailed computer architecture simulation datasets into manageable visualizations. Since the first implementation of the VISTA Environment by Mathew Jack in 2002, the VISTA Environment has evolved to meet several specific goals.

The specific goals of the VISTA Environment are to provide users with:

Visualizations for Effective Navigation of Rich and Complex Datasets. The VISTA Environment allows users to create visual representations of rich and complex datasets. While retaining the richness of the original dataset, data visualizations provide users with simple elements to explore the data, allow users to view large amounts of data on a single screen, and present the datasets at various levels of abstraction.

Dynamic and Interactive Presentations of Architecture Concepts. Visualizing of the dynamic behavior of complex architectures improves the dissemination of the ideas and concepts incorporated into these systems. Research into visualization tools for

computer architecture students have noted that "navigating through the screens of interesting, colorful visualizations maintains students' interest and can keep their brains active" [8] and that "students really like the [visualization] units, appreciate their availability … [and are] highly motivated [to try] these units at home." [6]

Access to Varied Computer Architecture Simulation Datasets without Significant Data Storage or Processing Overhead. The VISTA Environment leverages common attributes in data created by architecture simulations and allows users to quickly adapt their pre-defined visualizations to different sets of data, different architecture simulations, and different iterations of the same simulation. With the VISTA Environment, users have the freedom to choose or develop their own simulation package while continuing to use the VISTA Environment for their data visualization needs.

Common Visual Framework to Understand and Communicate Simulation Results. The VISTA Environment provides a common visual language to compare and contrast architecture simulation datasets. This common framework allows users to focus on understanding the architecture under study, rather than focusing on learning a new visualization environment or software application for each simulation dataset.

Customizable and Extensible Data Visualization Toolset. Since it is impossible to provide users with a complete set of visualizations, the VISTA Environment provides the framework for users to customize and create new visualizations. Users can rearrange, transform, manipulate and reformat the visualization environment. Users can create new and unique visualizations of their datasets and reuse these visualizations on other datasets.

2.2 Abstract Simulation Data Objects

The abstract data structures used to encapsulate simulation data in the VISTA Environment are inspired by the specifications of the Value Change Dump format. The Value Change Dump format is a common format for data from low-level hardware simulations and is specified in the IEEE Standard 1364 [14].

Notable aspects of the Value Change Dump format are:

- The metadata for all of the signals in the system is declared at the start of the file. This metadata is separate from any signal data.

- The signal is assigned a unique case-sensitive string identifier and is referred to by that key throughout the Value Change Dump file.
- The signal data is stored as value changes in discrete simulation time intervals or "simulation frames".
- The values are real numbers, strings or sets of bits.

These aspects of the Value Change Dump format are representative of computer architecture simulation data, and for the basis for data types in the VISTA Environment.

2.2.1 Signal Metadata Object

This data object contains signal metadata from the simulation. It is based on the $VAR declarations in the Value Change Dump specification in IEEE Standard 1364 [14]. During the Metadata declaration, the following information is given:

- `var_type` and `size`: This metadata can be used to determine the type of object that be used to store the signal data.
- `reference`: The name of the signal as a string of characters. This name does not have to be unique, and also provides insight about the hierarchical structure of the set of signals.
- `identifier_code`: The signal's unique case-sensitive string identifier.

15.2.3.8 $var

The **$var** section prints the names and identifier codes of the variables being dumped.

Syntax:

$var var_type size identifier_code reference **$end**

var_type ::=
event | **integer** | **parameter** | **real** | **reg** | **supply0** | **supply1** | **time**
| **tri** | **triand** | **trior** | **trireg** | **tri0** | **tri1** | **wand** | **wire** | **wor**

size ::= decimal_number

reference ::=
identifier
| identifier [*bit_select*_index]
| identifier [*msb*_index : *lsb*_index]

index ::= decimal_number

Size specifies how many bits are in the variable.

The identifier code specifies the name of the variable using printable ASCII characters, as previously described.

a) The msb index indicates the most significant index; the lsb index indicates the least significant index.

b) More than one reference name may be mapped to the same identifier code. For example, net10 and net15 may be interconnected in the circuit and therefore may have the same identifier code.

c) The individual bits of vector nets may be dumped individually.

d) The identifier is the name of the variable being dumped in the model.

Example:

```
$var
      integer 32 (2 index
$end
```

Figure 2-1: $VAR Specification from the IEEE 1364 Standard

The metadata information stored in the Structure can be extracted from the $VAR declarations in a Value Change Dump file.

2.2.2 Signal Value Object

This data object contains a signal's value tagged with the signal key. It is based on the Vector Value Change format in the Value Change Dump specification in IEEE Standard 1364 [14].

A value change is represented by a single line that contains the signal's `identifier_code` followed by the signal's new value. The value of the signal is a number, string or bitset. This bitset data can be of arbitrary length and contain unknown bits ('x') and high impedance bits ('z').

14.1.1.4 Unknown and high impedance values

When the result of an expression contains an unknown or high impedance value, the following rules apply to displaying that value.

In decimal (`%d`) format:

- If all bits are at the unknown value, a single lowercase "x" character is displayed.
- If all bits are at the high impedance value, a single lowercase "z" character is displayed.
- If some, but not all, bits are at the unknown value, the uppercase "X" character is displayed.
- If some, but not all, bits are at the high impedance value, the uppercase "Z" character is displayed.
- Decimal numerals always appear right-justified in a fixed-width field.

In hexadecimal (`%h`) and octal (`%o`) formats:

- Each group of 4 bits is represented as a single hexadecimal digit; each group of 3 bits is represented as a single octal digit.
- If all bits in a group are at the unknown value, a lowercase "x" is displayed for that digit.
- If all bits in a group are at a high impedance state, a lowercase "z" is printed for that digit.
- If some, but not all, bits in a group are unknown, an uppercase "X" is displayed for that digit.
- If some, but not all, bits in a group are at a high impedance state, then an uppercase "Z" is displayed for that digit.

In binary (`%b`) format, each bit is printed separately using the characters 0, 1, x, and z.

Figure 2-2: Rules for Unknown and High Impedance Values from the IEEE 1364 Standard

The bitset data often found in simulation data files contain bitset data of arbitrary length with unknown and high impedance bits.

2.2.3 Discrete Simulation Frame Objects

This data object contains a frame number, a set of all of the keys that change state during that frame, a set of new Signal Value Objects and a set of previous Signal Value Objects. It is based on the Value Change section in the Value Change Dump specification in IEEE Standard 1364 [14].

In computer architecture simulations, signal values change at discrete simulation time increments. In a Value Change Dump file, there is a special marker—the "#" sign—that signifies a time increment. After each marker, there is a new frame number and a list of signal identifier and signal value pairs.

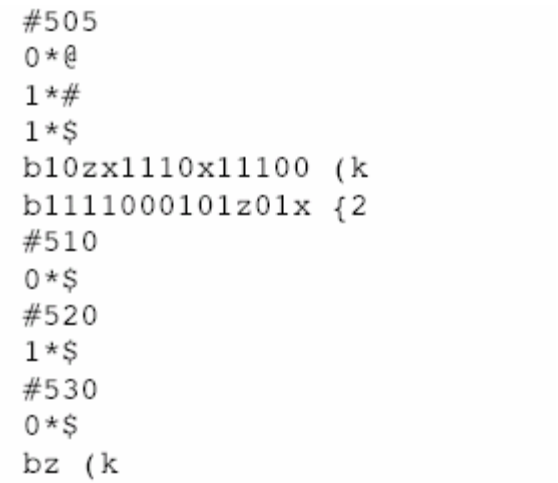


Figure 2-3: Sample Value Change Section from a Value Change Dump File

In computer architecture simulations, signal values change at discrete simulation time increments. In a Value Change Dump file, there is a special marker—the "#" sign—that signifies a time increment. After each marker, there is a new frame number and a list of signal identifier and signal value pairs.

Since the previous value of the signal is not preserved, the information contained in a Value Change Dump can only be used to determine the next state of the system. For instance, if someone knew the values of all of the signals at frame #520, he could determine the value of all of the signals at frame #530. However, he could not determine the value of all of the signals at frame #510.

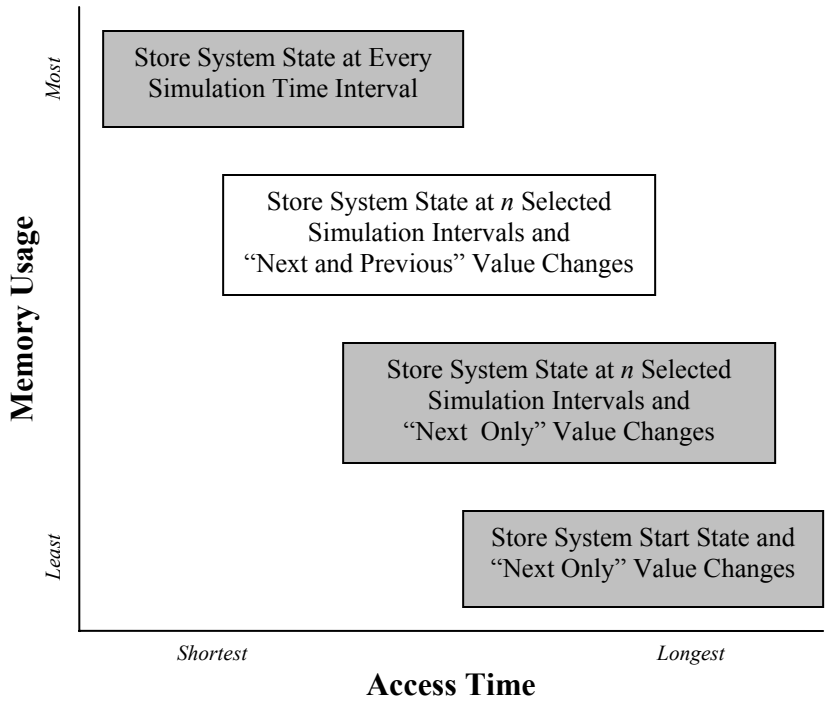


Figure 2-4: The Memory and Access Time Cost of Various Value Change Schemes

The VISTA Environment implements a comfortable medium between memory usage and access time. The VISTA Environment stores both the previous value and next value change information when a signals value changes and it stores the complete system state at regular simulation time intervals.

The technique of storing only the next value change information is useful to reduce the storage of the data. However, this scheme requires the most time to derive the system state at an arbitrary simulation time. On the other hand, storing the system state at every simulation time interval would minimize the time required to derive a system state, but required an enormous amount of data storage.

The VISTA Environment implements a comfortable medium between these two techniques that balances memory usage and access time. The VISTA Environment stores both the previous value and next value change information when a signals value changes and it stores the complete system state at regular simulation time intervals.

Since the VISTA Environment stores both the previous value and next value change information, the VISTA Environment can derive the system state at a desired simulation time from a system state at any arbitrary simulation time. In the technique

used in the VCD file, a user can derive the system state for a desired simulation time only from a system state that is from a previous simulation time. Typically, the technique used by the VISTA Environment will require slightly more memory than other schemes, but it allows the VISTA Environment to quickly derive a system state at any given simulation time.

2.3 Design Overview

The VISTA Environment is composed of three sets of components that interface with each other. These components are responsible for responding to requests from the other components, and, in the case of the Graphical User Interface, responsible for responding to requests from the user. Although the VISTA Environment is implemented in Java, these components are open to many different implementations.

In the first section, there is a brief explanation of the components and their functions. Next, there is an overview of how these components interface with each other. Finally, this section provides a design overview for each component.

2.3.1 Three Components of the VISTA Environment

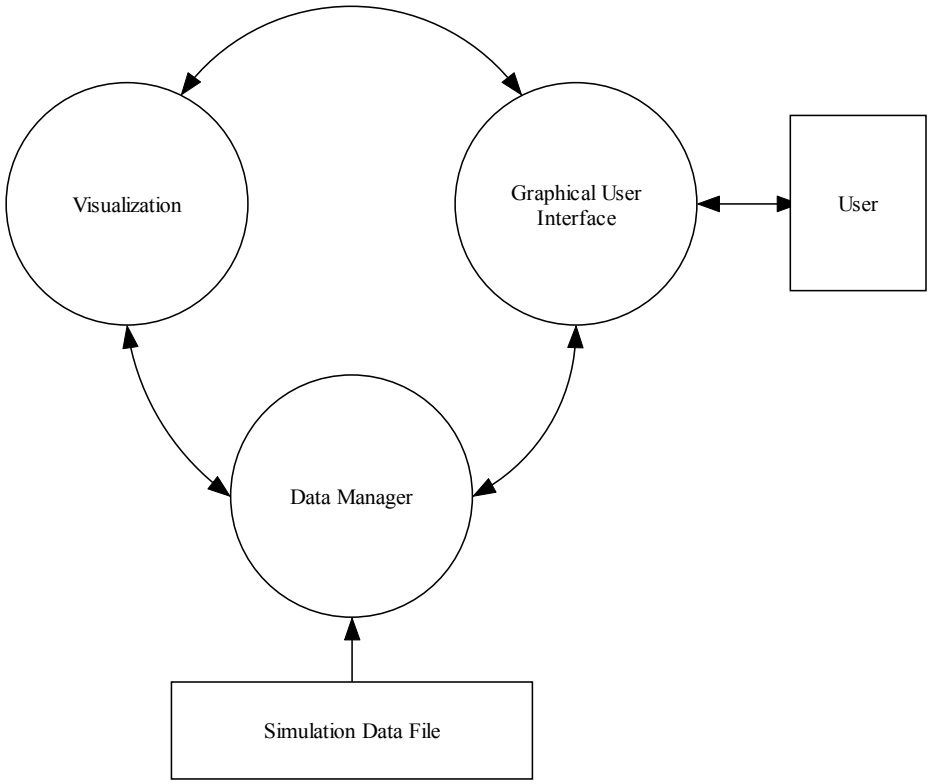


Figure 2-5: Conceptual Component Model of the VISTA Environment

The VISTA Environment is composed of three sets of components that interface with each other. Although the VISTA Environment is implemented in Java, these interfaces are simple and are open to many different implementations.

Data Manager: This component is responsible for importing the simulation data at the request of the GUI and for translating the simulation data into objects that are usable by the Visualization component.

Visualization: This component is responsible for retrieving simulation data from the Data Manager and providing the GUI with visual representations of the simulation data.

Graphical User Interface: This component is responsible for presenting the Visualization components to the user and for presenting the status of the importing process from the Data Manger. Also, this component handles the user's interaction with the VISTA Environment and relays requests to the other components when required.

2.3.2 Interfacing the VISTA Components

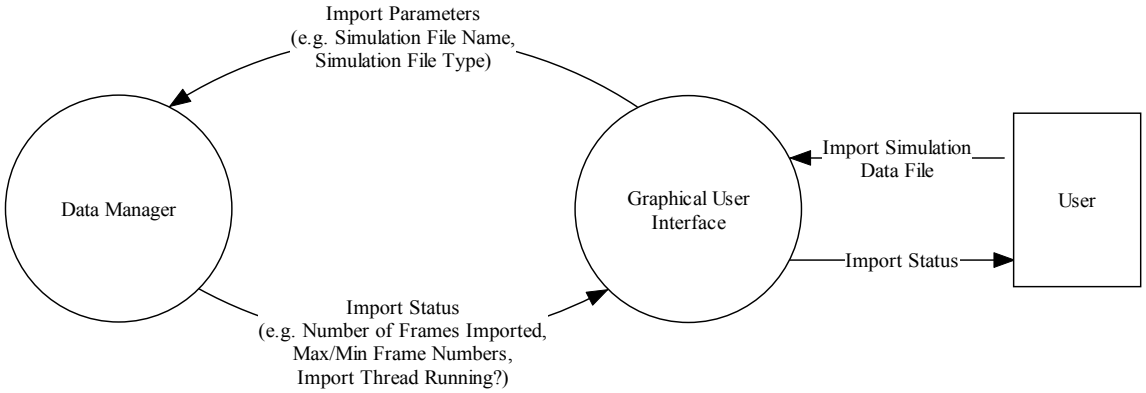


Figure 2-6: Interface Between GUI and Importing and Managing Simulation Data Components

Through the Graphical User Interface, the user can specify the simulation data file that he wants to import and can monitor the status of the simulation data file import process.

The GUI takes the simulation data file specifications from the user and provides the Data Manger with simulation file parameters (i.e. file location, and simulation file type). The Data Manager then initiates the simulation data importing process.

While the Data Manager is importing simulation data, it provides the GUI with status information about the state of the data import. The GUI presents this status information to the user so that he can monitor the status of the simulation data file import process.

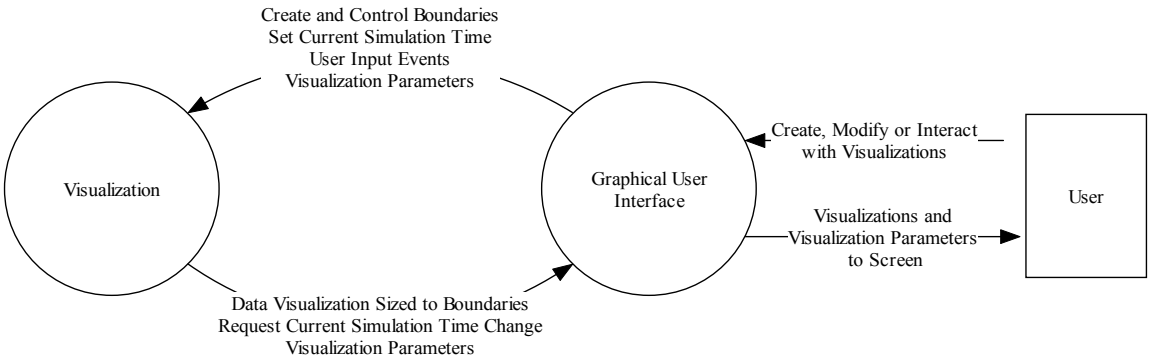


Figure 2-7: Interface Between the GUI and the Visualization

Through the Graphical User Interface, the user can create, modify or interact with visualizations.

The GUI is responsible for creating the boundaries for the visualizations, notifying visualizations of changes of the current simulation time, passing relevant user input events to visualizations, and allowing the user to adjust parameters in the of the visualizations.

The Visualization component is responsible for providing the GUI with visualizations in response to the GUI's request, fitting the visualizations to boundaries imposed by the GUI, sending requests to the GUI when a visualization wants to change the current simulation time, and providing the GUI with editable parameters for a visualization.

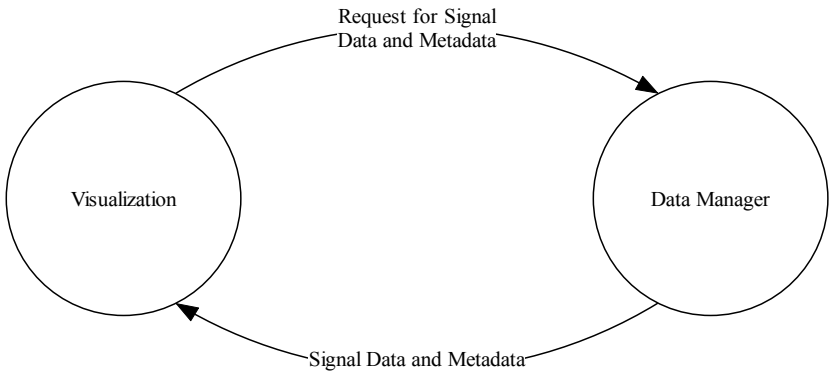


Figure 2-8: Interface Between the Visualizations and Data Components

The Visualization component contains visualizations of simulation data and requests the simulation data from the Data Manager. The Data Manager provides the visualizations with simulation data and metadata.

2.3.3 Importing and Managing Simulation Data Component Overview

In order to interface with the visualizations and the GUI, the Data Manager implements several sub components to orchestrate and delegate the tasks.

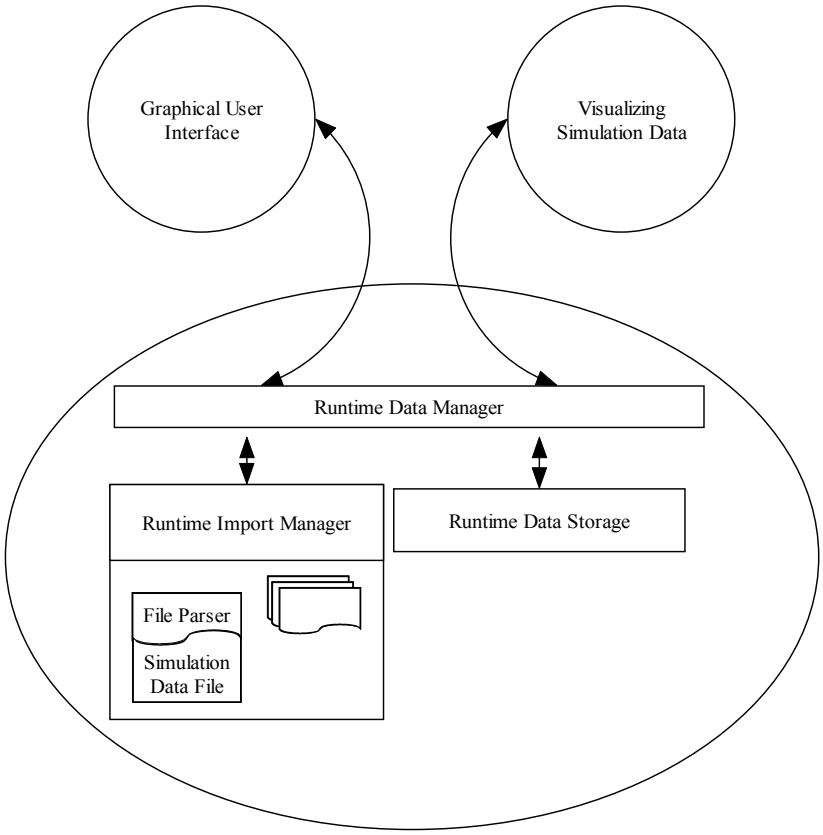


Figure 2-9: Conceptual Model of the Components of Data Components

Runtime Data Manager: The Runtime Data Manager is the bridge between the Data Manager and the rest of the VISTA Environment. The Runtime Data Manager orchestrates the other processes in the Data Manager, provides the visualizations with methods to retrieve simulation data, provides the GUI with methods to initiate the simulation importing process, and provides the GUI with methods to retrieve the status of the simulation importing process.

Simulation Import: The Simulation Import components include a Runtime Import Manager and a set of simulation file parsers. The Runtime Import Manager controls the simulation parser on a separate program thread, allowing the simulation file parsing and creation of data objects to be run in the background. The Runtime Import Manager has methods that allow the Runtime Data Manager to initiate the simulation importing process and to retrieve the status of the simulation importing process.

Runtime Data Storage: The Runtime Data Storage is used to store simulation data objects that can be transferred to other storage methods. It is reasonable to assume that alternative storage methods for simulation data objects will be required to store the

data objects created in the simulation importing process. Simulation data files can be quite large—on the order of hundreds of megabytes or gigabytes—and many operating system will not allow the Java Virtual Machine this much memory.

2.3.4 Visualizing Simulation Data Component Overview

In order to interface with the Data Manager and the Graphical User Interface, the Visualization Simulation Data component implements two sub components and adheres to a model to communicate "current simulation time". The two components are View Layout and View Object and the current simulation time model is a Global Time Event.

Global Time Event Listener: Simulation data is stored in discrete time intervals and a visualization can represent the data at any time within the simulation. However, it is useful to have all of the visualizations presented by the GUI represent a consistent time. This gives rise to a "current simulation time" to which all of the visualizations are synchronized. When this current simulation time changes, the visualization must change to represent this different current simulation time. Each component that implements the Global Time Event Listener model can "hear" changes in the current simulation time represented in the VISTA Environment.

View Object: The View Object is the basic building block of the visualizations. View Objects are used to visualize a single signal's simulation data over a range of simulation time. A View Object is placed in a View Layout. View Objects request data from the Data Manager, and requests a change in the current simulation time from the Graphical User Interface. View Objects also respond to changes in the current simulation time from the GUI, allows the GUI to access to view and modify some of its fields, and can respond directly to user input events.

View Layouts: The View Layout is container for a set of View Objects. A View Layout does not present simulation data, and creates a data visualization composed of View Objects in the boundaries set by the GUIs. View Layouts, like View Objects, respond to changes in the current simulation time from the GUI and allow the GUI to access to view and modify some of their fields, and can respond directly to user input events.

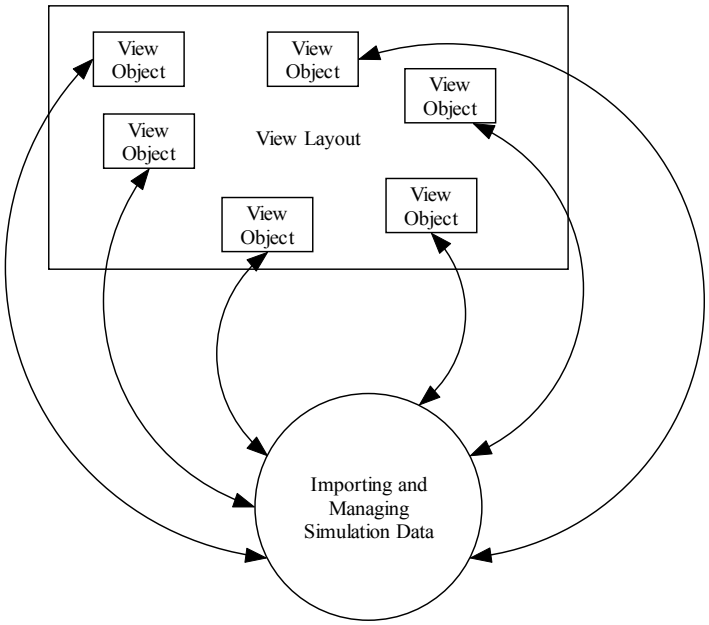


Figure 2-10: Conceptual Model of View Objects in a View Layout Accessing Simulation Data From the Importing and Managing Simulation Data Component

2.3.5 Graphical User Interface Component Overview

In order to interface with the User, the Data Manager and the Visualization Simulation Data component, the Graphical Interface component implements several sub components.

Visualization Containers: The GUI has visualization containers to control the boundaries of visualizations from the visualizations. Also, these containers route user input events directly to the visualizations.

Global Time Event Manager: The GUI controls the Global Time Event Manager and has an interface receiving requests to change current simulation time from the visualizations. Also, the Global Time Event Manager announces Global Time Events that Global Time Event Listeners can "hear."

Modifying and Examining the VISTA Environment: The GUI has tools that allow the user to initiate a simulation import process and to receive feedback about the status of this process. Also, there are tools that allow users to examine and modify information of visualizations.

Chapter 3:

Data Manager Implementation

This chapter covers the implementation of the mechanisms that are responsible for translating the simulation data into objects that are usable in the visualization process. These mechanisms read the structure and state data contained in the simulation files, encapsulate these values into data objects and provides a standard interface for the visualization process to access the data. Although the details of this process are abstracted from typical users, some users will be required to create new simulation parsers for new data sources.

The first section in this chapter introduces the **Vista Data Structures** that are used to store the information contained in the simulation files. These **Vista Data Structures** implement the Abstract Simulation Data Objects (see 2.2).

Next, the **Simulation Import** section details the simulations import components that are used to parse the simulation data files and create the **Vista Data Structures**. Since some users will be required to create new simulation parsers, these components include templates and tools to assist in creating new parsers.

The third section details the **Runtime Data Manager** mechanism that serves as the bridge between these components and the rest of the VISTA Environment. The **Runtime Data Manager** controls the data importing process, manipulates the data once it is imported, and provides VISTA with a simple interface to the data.

The final section explains the **Runtime Data Storage** system that the **Runtime Data Manager** uses to store simulation data. This system opens up the possibility for users to store the **Vista Data Structures** outside of the software heap and onto disk or network storage.

3.1 Vista Data Structures

The VISTA Environment uses four different objects to encapsulate simulation data. These objects implement the Abstract Simulation Data Objects defined in Section 2.2.

The **Structure** object is the implementation of the Signal Metadata Object and contains signal metadata from the simulation. Each piece of metadata is tagged with a key—a unique case-sensitive string identifier—that corresponds to the signal.

A **VisValue** is an implementation of the Signal Value Object and holds the value of a signal tagged with the same signal key used in the metadata contained in the **Structure**. The value of the signal is either a number or string.

A **TraceFrame** is an implementation of the Discrete Simulation Frame Object and contains a frame number, a set of keys that correspond to signals that change value during the frame, a set of the new **VisValues** for the signals that change, and a set of the previous **VisValues** for the signals that change.

An **Image** contains the values and metadata of all of the signals in a system at a specific simulation time. The specific simulation time that an **Image** represents can be changed by applying a particular set of **TraceFrames** in a particular order.

3.1.1 Structure

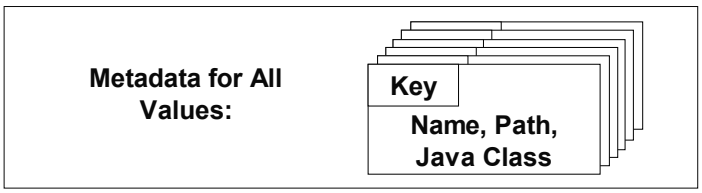


Figure 3-1: Diagram of a Structure Object

The Structure contains a set of metadata objects that correspond to signals in the simulation. These metadata objects are tagged with a key—a unique case-sensitive string identifier—that corresponds to the signal.

The Structure contains a set of metadata objects that correspond to signals in the simulation. The metadata includes the name of the signal, the path of the signal in the simulation structure hierarchy, and the Java class that is used to encapsulate the value of the signal.

The name of the signal is stored as a string of characters. This name does not have to be unique and the length is not limited.

The path of the signal is stored as an array of strings. The lowest element in the array (i.e. String[0]) is the furthest ancestor from the signal and the highest element (i.e. String[length-1]) is the parent of the signal. The root of the structure hierarchy is not contained in the path array (instead, the root is stored in the Structure name field). A path of length zero means that the signal is a direct descendent from the root.

These metadata objects are tagged with a key—a unique case-sensitive string identifier—that corresponds to the signal. The key is a string of characters that is not limited in length but it is required to be unique within the VISTA Environment. The key is used throughout the VISTA Environment for retrieving signal data, and identical keys could lead to undesired results.

3.1.2 VisValue

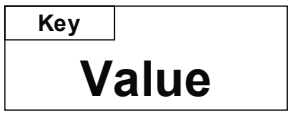


Figure 3-2: Diagram of a VisValue Object

A VisValue holds the value of a signal and a key that uniquely identifies the signal.

A VisValue stores the state data for a signal in either a String (java.lang.String) or a concrete subclass of the Number (java.lang.Number) abstract class. This allows VISTA to store both strings and numbers in a single data object that can be operated on by the visualizations. Also, a VisValue is tagged with a key that corresponds to the metadata stored in the Structure (see 3.1.1).

The VISTA Environment contains VCDBitSet, a subclass of java.lang.Number that can handle simulation data bitsets. Currently, VCDBitSet can handle arbitrary length bitsets, and sets unknown and high impendence bits to zero.

3.1.3 TraceFrame

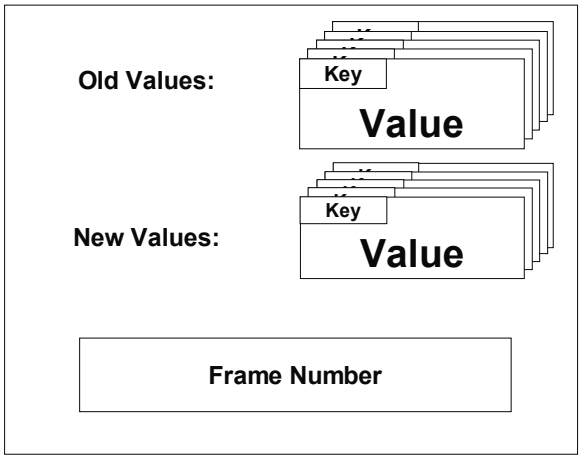


Figure 3-3: Diagram of a TraceFrame Object

A TraceFrame contains a frame number, a set of all of the keys that change state during that frame, a set of the new VisValues for the signals that change, and a set of the previous VisValues for the signals that change.

A TraceFrame contains a frame number, a set of the keys of the signals that change value during that frame, a set of the new VisValues for the signals that change, and a set of the previous VisValues for the signals that change. For each key in the set of keys, there is a corresponding new VisValue and a corresponding old VisValue.

3.1.4 Image

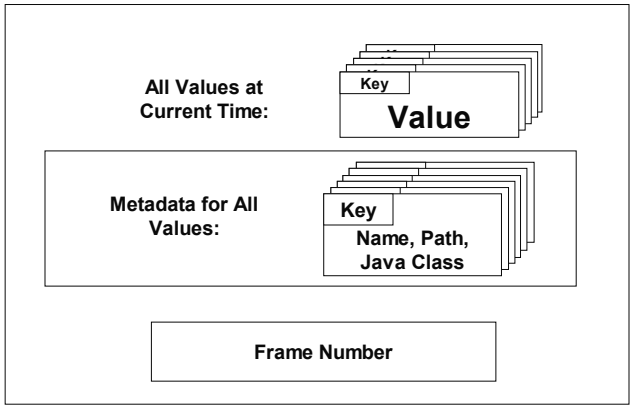


Figure 3-4: Diagram of an Image Object

An Image contains the values and metadata of all of the signals in a system at a specific simulation time. The specific simulation time that an Image represents can be changed by applying a particular set of TraceFrames in a particular order.

An Image contains the values and metadata of all of the signals in a system at a specific simulation time. The specific simulation time that an Image represents can be changed by applying a particular set of TraceFrames in a particular order. Although only one Image is visible to the user in the VISTA Environment, the Runtime Data Manager caches Images at regular simulation time intervals in order to minimize the access time required for a given simulation time (see 3.3).

TraceFrames can be applied to an Image to change the given simulation time that the Image represents. However, care needs to be taken to apply the correct trace frame, or the Image will represent incorrect data. The process of applying TraceFrames is performed by the Runtime Data Manager (see 3.3).

3.2 Simulation Import

The VISTA Environment contains two components for importing simulation data. These components are used to parse the simulation data files and create the Vista Data Structures. Since some users will be required to create new simulation parsers, these components include templates and tools to assist in creating new parsers.

SimParser is an abstract class that developers can subclass in order to create a new parser. The abstract **SimParser** class takes care of many of the details of writing a new simulation parser and allows developers to create add parsers without much effort. The VISTA Environment has a VCDPaser, a subclass of **SimParser,** which can be used to parse Value Change Dump files.

The **Runtime Import Manager** controls the simulation parser on a separate program thread, allowing the simulation file parsing and Vista Data Structure creation to be run in the background. Developers do not need to understand the details of **Runtime Import Manager** in order to write new parsers.

3.2.1 SimParser

Users that want to import a new simulation file format into the VISTA Environment are required to write a new simulation parser. Since most of these simulation data files are based on a proprietary format, it is reasonable to assume that

users will be writing parsers. To make this process easier, users subclass the abstract SimParser class to create a new simulation parser.

In order to create a new parser, the simulation parser developer will need to implement several abstract methods from the SimParser class. The developer will need to write a constructor that initializes the Structure of the simulation data. Also, since the SimParser implements the iterator interface, the simulation parser developer will need to implement the `hasNextFrame` and `nextFrame` methods. The `hasNextFrame` method returns a `true` value if there is another frame in the simulation file and the `nextFrame` method returns the next Trace Frame.

3.2.2 Runtime Import Manager

The Runtime Import Manager creates and controls an instance of the SimParser class to import the user's data. The Runtime Import Manager runs on a separate program thread, allowing the simulation file parsing and Vista Data Structure creation to be run in the background. The Runtime Import Manager is used by the Runtime Data Manger to control an implemented simulation parser and provide feedback on the status of the parser.

Since the Runtime Import Manager runs on a separate thread, it shares processor time with other threads. Users of the VISTA Environment can set the priority of the Runtime Import Manager thread, or even stop the thread completely. While the Runtime Import Manager is running, the Runtime Data Manager periodically checks on its status and the Vista Data Structures it has created. If the Runtime Data Manager detects new Vista Data Structures, it brings them into the VISTA Environment.

3.3 Runtime Data Manager

The Runtime Data Manager is the bridge between the Importing and Managing Simulation Data set of components and the rest of the VISTA Environment. The Runtime Data Manager orchestrates the Importing and Managing Simulation Data processes and provides the rest of the VISTA Environment with simple methods to import and retrieve simulation data.

The VISTA Environment can to initiate a simulation data import by calling the `importTraceFile` method in the Runtime Data Manager. This method requires a file path and a string identifier of the parser to be used. The Runtime Data Manager will open the file, package the file data into a Buffered Input Stream, create the corresponding simulation parser, and start the Runtime Import Manager process to import the data.

The VISTA Environment can retrieve a VisValue of a signal at a specific simulation time calling the `getVisValue` method in the Runtime Data Manger. This method requires the signal's key and an integer corresponding to the desired simulation time.

The simplicity of these commands masks the complexity of the processes. For instance, the process to retrieve a VisValue requires the Runtime Data Manager to perform a number of operations to derive the desired simulation state of the system.

When the VISTA Environment retrieves a VisValue from the Runtime Data Manager, a number of operations occur. Although the Runtime Data Manager only has one active Image, it stores a number of inactive Images in the Runtime Data Storage. These inactive Images are snapshots of the simulation state of the system at certain intervals in the simulation time. If an inactive Image is closer to the desired simulation time than the active image, the active image is replaced by the inactive one.

Then the Runtime Data Manager determines the TraceFrames that are required to get the active Image to the desired simulation time and applies these TraceFrames correctly. If the active Image crosses a certain interval in the simulation time during the process of applying these TraceFrames, this active Image will be cloned and stored in the Runtime Data Storage.

3.4 Runtime Data Storage

In the development of the VISTA Environment, a major limitation has been the amount of memory that an operating system will allocate to the Java Virtual Machine. Simulation data files can be quite large—on the order of hundreds of megabytes or gigabytes—and many operating system will not allow the Java Virtual Machine this much memory. The Runtime Data Storage is used to store data objects that can be transferred to other storage methods.

The Runtime Data Storage is an abstract class that serves as an interface to any number of different data storage schemes to be implemented in a subclass. These schemes could involve a storing the data in another JVM process, on the machine's local disk or across a network. Although the Runtime Data Storage mechanism has a number of potential implementations, the current implementation (SimpleRuntimeDataStorage) stores the data in the Java Virtual Machine heap.

Chapter 4:

Visualization Implementation

This chapter covers the implementation of the mechanisms that are responsible for creating visual representations of the simulation data in the VISTA Environment. Once the simulation data has been imported into the VISTA Environment, these visualization mechanisms can access and present the data to the user. Users can configure a set of standard visualizations or of develop their own.

These components are developed using the Swing Graphics User Interface package from the Java Foundation Classes (JFC). Swing includes a number of methods and features that help users create powerful Graphical User Interfaces and users familiar with Swing will find the task of developing new visualizations straightforward.

The **View Object** is the basic building block of visualizations in the VISTA Environment**. View Objects** are implemented as Swing components and are used to visualize a single signal's simulation data.

View Layouts are containers for **View Objects**. A **View Layout** manages a set of **View Objects**, controls their boundaries, and responds to any events they might fire. **View Layouts**, like **View Objects,** are Swing components but do not represent any signal data or metadata.

The **View Object** and **View Layout** model is a flexible, reusable and modular model for developing visualizations in the VISTA Environment. However, not all visualizations fit this model and the **Signal Explorer** demonstrates how the VISTA Environment can be made to support alternative design models.

Finally, this chapter discusses the **Global Time Listener Interface** that visualizations must implement in order to be notified of current time changes.

4.1 View Objects

View Objects are used to visualize the simulation data imported into the VISTA Environment. View Objects are subclasses of JPanel (javax.swing.JPanel), a standard Java Swing component, and make use of Swing's painting, layout and event-handling methods. Users familiar with Swing are encouraged to write new View Objects and will find the task straightforward.

View Objects have access to the simulation data stored in the VISTA Environment. A View Object has a key—a unique case-sensitive string identifier that corresponds to the signal—and can use the key to retrieve signal data (see 3.1.2) or signal metadata (see 3.1.1) from the VISTA Environment data storage (see 3.4) for any simulation time. The VISTA Environment has a concept of a current simulation time (see 3.1.4) and calls a View Object's `UpdatePresentation` method whenever this current simulation time changes.

View Objects make use of the JPanel class from Java's Swing package. Although a detailed explanation of Swing and JPanels are beyond the scope of this paper, this implementation allows users familiar with Swing to create new View Objects quickly and import visualizations written for other applications without much effort. It also gives users the ability to include other Swing components, layouts, painting routines and event-handling methods in a View Object.

In this release of VISTA, there are two View Objects. The Single Value View Object displays a text representation of the signal's data and metadata at a single simulation time. The Plot View Object graphs the signal's data over a range of simulation times.

Although these are quite simple View Objects, VISTA users who want to write new View Objects should examine the code behind these View Objects to learn how View Objects interact with the VISTA Environment. Swing users will find the code straightforward and users new to Swing will find that the View Objects provide a good introduction to Swing's capabilities and functions.

4.1.1 Single Value View Object

The Single Value View Object (see Figure 4-1) is a simple View Object that displays a text representation of the signal's data and metadata at a single simulation time. The Single Value View Object provides a good example of how use other Swing components in a View Object and how to interact with the VISTA Environment.

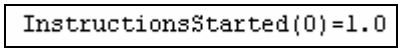


Figure 4-1: Example of a Single Value View Object

The Single Value View Object is a simple View Object that displays a text representation of the signal's data and metadata at a single simulation time. This Single Value View Object is displaying that the Signal named "InstructionsStarted" at simulation time "0" is equal to "1.0".

A Single Value View uses a JLabel (another swing component) to display the signal's data and metadata at the current simulation time in a text format. A JLabel can display plain text, images and interpret HTML encoded text. The Single Value View Object delegates the painting and layout responsibility to the JLabel.

The VISTA Environment notifies the Single Value View Object of a change in the current simulation time by calling the Single Value View Object's `UpdatePresention` method. In this method, the Single Value View Object retrieves from the VISTA Environment the current simulation time, the signal's data, and the signal's metadata. The Single Value View Object makes this data into a String, and passes the String onto the JLabel to present it.

4.1.2 Plot View Object

The Plot View Object (see Figure 4-2) is slightly more complicated than the Single Value View Object. The Plot View Object presents a signal's data over a range of simulation time, implements its own Swing painting method, and it fires a Java Property Change Event whenever a user modifies the Plot View Object's range of simulation time.

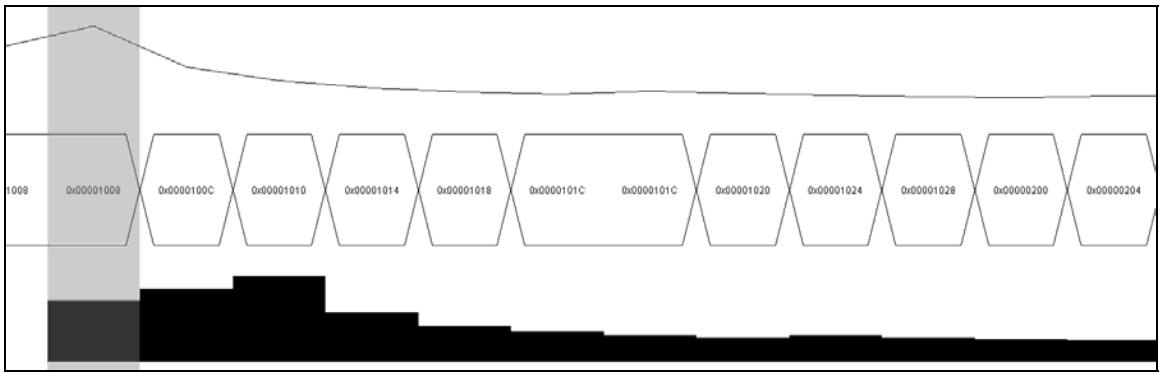


Figure 4-2: Examples of a Plot View Object: Line Graph, Waveform Graph, and a Bar Graph

The Plot View Object presents a signal's data over a range of simulation time, implements its own Swing painting method, and it fires a Property Change Event whenever a user modifies the Plot View Object's range of simulation time. In this example, there are three Plot View Objects that are plotting simulation data over a range of 13 simulation time intervals. The vertical bar represents the current simulation time in the VISTA Environment.

A Plot View Object represents the signal's data over a range of simulation time. Although the VISTA Environment has a concept of current simulation time, it also provides methods to retrieve signal data at any arbitrary simulation time. While the Plot View Object could obtain the signal's data within the entire range of the simulation time each time the Plot View Object needed to repaint, this would cause an undue burden on the Runtime Data Manager in the VISTA Environment. Instead, the Plot View object will obtain all of the data within the range of simulation time when it is created and store it internally.

The Plot View Object implements its own painting method to plot a waveform, a bar graph or a line graph over a range of simulation time. Also, the Plot View Object paints a marker on the graph to represent the current simulation time within the VISTA Environment and repaints this marker on notification from the VISTA Environment of a change in current simulation time.

Implementing a custom painting routine is more difficult than delegating painting responsibilities to another Swing component, but Swing provides simple graphics tools that allow for sophisticated control over geometry, color, and text presentation. The Plot View Object's `updatePresentation`, `paintComponent`, `drawBarGraph`, `drawLineGraph`, `drawWaveformGraph`, and `drawCurrentBar` are good examples of the simplicity and power of the Swing graphic tools.

Finally, the Plot View Object uses Swing's model of Event Handling in order to inform other Swing components that its range of simulation time have changed. If a user modifies the range of simulation time of a Plot View Object at runtime, the Plot View Object will fire a Property Change Event (java.beans.PropertyChangeEvent). These events can be heard by other Swing components and the other Swing components can act accordingly. For instance, if a Plot View Layout (see 4.2.3) hears this event from one of its Plot View Objects, the Plot View Layout will change the simulation time ranges of its other Plot View Objects.

4.2 View Layouts

View Layouts are containers for View Objects. A View Layout manages a set of View Objects, controls their boundaries, and responds to any Property Change Events they might fire. View Layouts, like View Objects, are subclasses of JPanel and are notified when the current time changes in the VISTA Environment. However, View Layouts require a Swing Layout Manager—either a new implementation or a standard Swing Layout Manager—and do not represent any signal data or metadata.

In this release of VISTA, there are three View Layouts. Flow View Layout demonstrates the simplicity of the View Layout system by employing Flow Layout, a standard Swing Layout Manager. Grid View Layout implements its own layout manager and can load an image as its background. Finally, Plot View Layout responds to events generated by Plot View Objects and delegates its layout duties to a JScrollBar, a Swing Component, coupled with a Box Layout, a standard Swing Layout Manager.

4.2.1 Flow View Layout

The Flow View Layout (see Figure 4-3) is the simplest of any View Layout. The Flow View Layout does not do anything in response to changes in the current simulation time of the VISTA Environment nor does it listen for Property Changed Events that might be fired by its set of View Objects.

The Flow View Layout employs a Flow Layout Manager (java.awt.FlowLayout), a standard Java Layout Manager, to control the boundaries of its set of View Object. The

Flow Layout Manger arranges the set of View Objects in a left-to-right flow, much like lines of text in a paragraph.

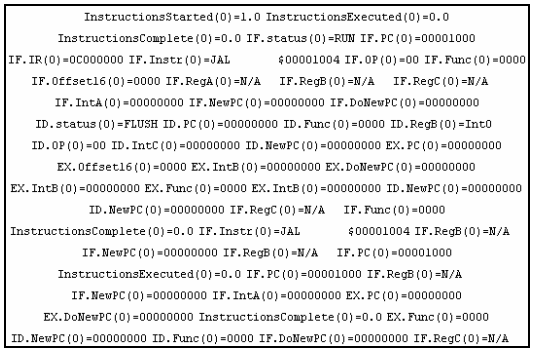


Figure 4-3: Example of a Flow View Layout with a Set of Single Value Views

> Flow View Layout is the simplest of any View Layout. The Flow Layout Manger arranges the set of View Objects in a left-to-right flow, much like lines of text in a paragraph.

4.2.2 Grid View Layout

The Grid View Layout (see Figure 4-4 and Figure 4-5) implements its own Layout Manager in order to "snap" View Objects to a grid and can load an image as its background. The Grid View Layout does not do anything in response to changes in the current simulation time of the VISTA Environment nor does it listen for Property Changed Events that might be fired by any of its View Objects.

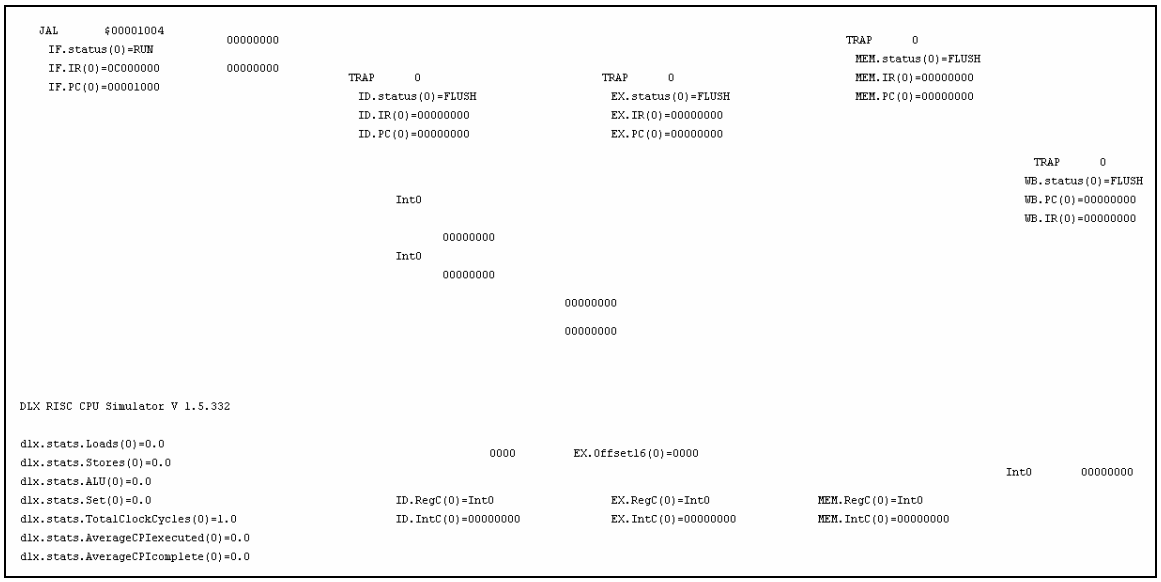


Figure 4-4: Example of a Grid View Layout with a Set of Single Value Views

The Grid View Layout implements its own Layout Manager to "snap" View Objects to a grid. The Grid View Layout does not do anything in response to changes in the current simulation time of the VISTA Environment nor does it listen for Property Changed Events that might be fired by any of its View Objects.

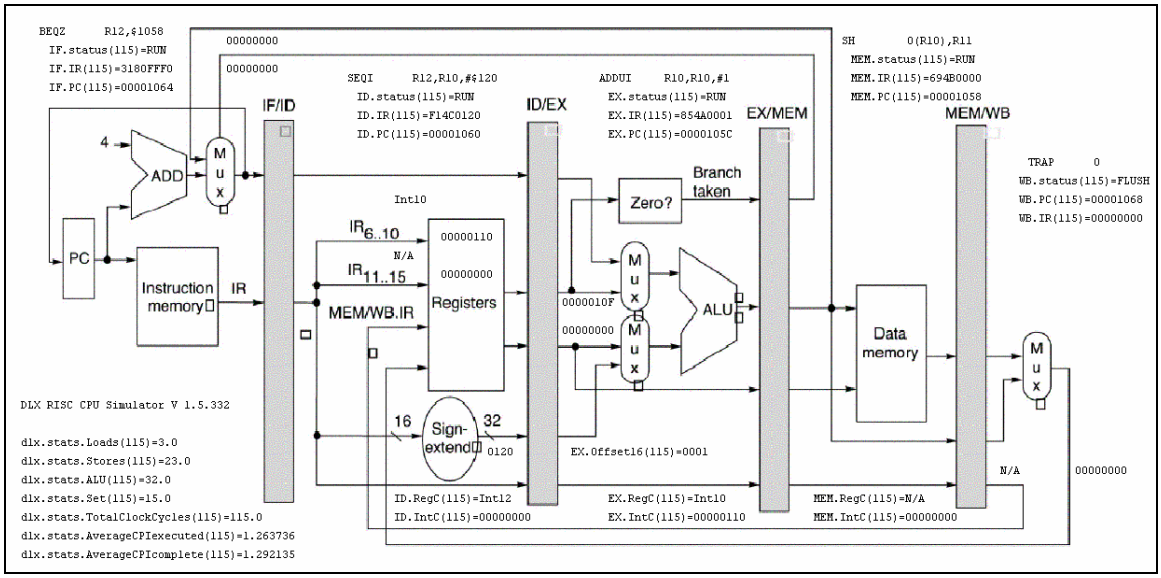


Figure 4-5: Example of a Grid View Layout with Single Value Views and Background Image

The Grid View Layout implements a custom painting method in order to control presentation. For example, this Grid View Layout has a set of Single Value Views of signal data overlaying an image of the corresponding Five Stage Pipelined processor.

Java comes standard with eight implemented Layout Managers and the ability to use an "Absolute Positioning" layout scheme (by setting the Layout Manager to null).

Although these Layout Mangers are designed to handle almost any layout task, there is not a Layout Manger designed to “snap” its components to a grid.

The Grid View Layout Manager was designed to perform this task by “snapping” the upper left corner of a View Object to an intersection on the grid. The size of the grid can be change by a user at runtime.

The Grid View Layout, like the Plot View Object (see 4.1.2), also implements a custom painting method in order to control its presentation. The Grid View Layout can paint its background as a grid, an image from a file, or both. This enables users to overlay View Objects on top of images that complement the View Objects (see Figure 4-5).

4.2.3 Plot View Layout

The Plot View Layout (see Figure 4-6), unlike any of the other View Layouts, delegates its layout duties another Swing component, responds to events generated by its set of View Objects and responds to changes in the current simulation time maintained by the VISTA Environment.

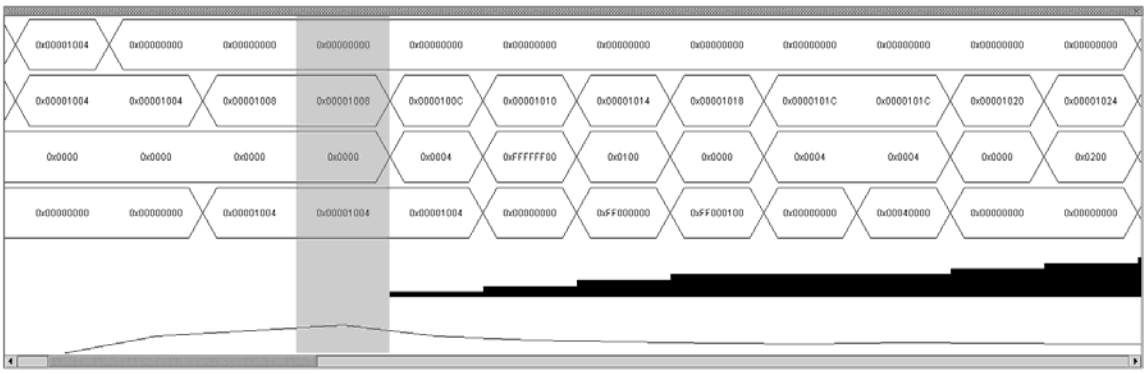


Figure 4-6: Example of a Plot View Layout with a set of Plot View Objects

The Plot View Layout, unlike any of the other View Layouts, delegates its layout duties another Swing component, responds to events generated by its set of View Objects and responds to changes in the current simulation time maintained by the VISTA Environment.

The Plot View Layout uses a JScrollBar in order to view a portion of the View Object at once. This is useful for View Objects, like the Plot View Objects (see 4.1.2), that represent a signal over a range of simulation times. The boundaries of the set of View Objects are controlled by a Grid Layout Manager (java.awt.GridLayout) in order to

maintain the View Object in one column with unlimited number of equally distributed rows. In order to achieve the effect of zooming in and out of the View Object, the width of the underlying panel is adjusted.

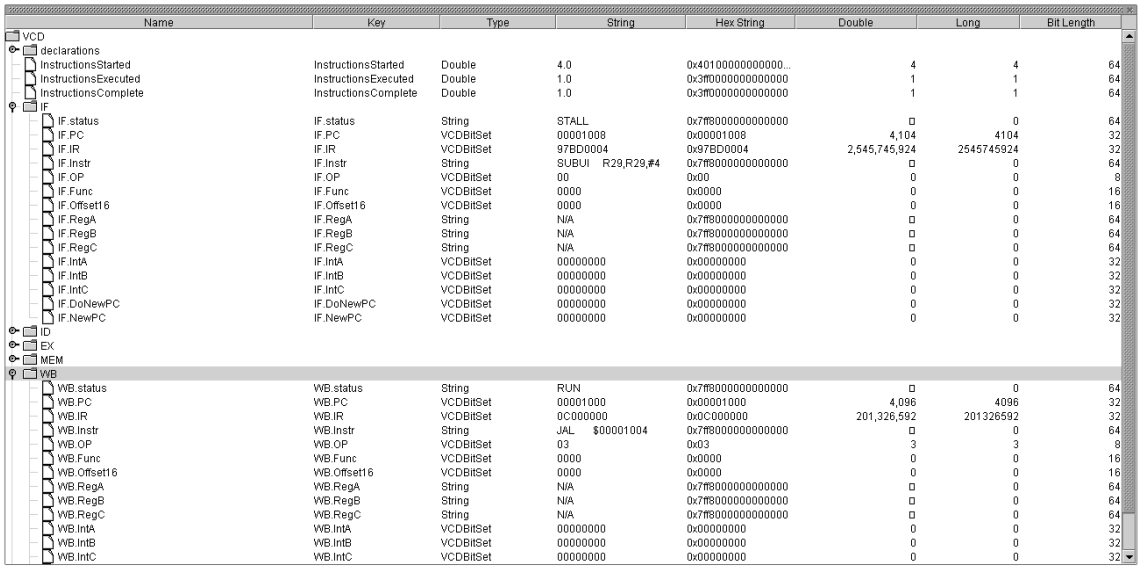
Some View Objects can fire Property Change Events and the Plot View Layout listens for events that signify a change in the range of data that is being represented by one of its View Objects. If a Plot View Layouts hears one of its View Objects fire one of these Property Change Events, the Plot View Layout propagates this change to its other View Objects.

Finally, the Plot View Layout responds to changes in the current simulation time in the VISTA Environment. Once notified of the change in current simulation time, a Plot View Layout will check to see if this simulation time data is visible. If the simulation time data is not visible, the Plot View Layout will move the JScrollBar to a position such that the simulation time data is visible.

4.3 Signal Explorer

The View Object and View Layout model provides users with flexible, reusable and modular model to develop visualizations in the VISTA Environment. Occasionally, however, a user might run into a situation where this model does not fit their visualization design. The VISTA Environment can be made to support alternative design models, but it is not encouraged and requires more work on the designer's part.

The Signal Explorer (see Figure 4-7) is an example when the View Object and View Layout model failed. In this instance, there existed an excellent Swing component named JTreeTable [15] and the Signal Explorer needed to display data obtained from the entire Structure (see 3.1.1).



Name	Key	Type	String	Hex String	Double	Long	Bit Length
VCD							
declarations							
InstructionsStarted	InstructionsStarted	Double	4.0	0x401000000000...	4	4	64
InstructionsExecuted	InstructionsExecuted	Double	1.0	0x3ff0000000000000	1	1	64
InstructionsComplete	InstructionsComplete	Double	1.0	0x3ff0000000000000	1	1	64
IF							
IF.status	IF.status	String	STALL	0x7ff8000000000000	□	0	64
IF.PC	IF.PC	VCDBitSet	00001008	0x00001008	4,104	4104	32
IF.IR	IF.IR	VCDBitSet	97BD0004	0x97BD0004	2,545,745,924	2545745924	32
IF.Instr	IF.Instr	String	SUBUI R29,R29,#4	0x7ff8000000000000	□	0	64
IF.OP	IF.OP	VCDBitSet	00	0x00	0	0	8
IF.Func	IF.Func	VCDBitSet	0000	0x0000	0	0	16
IF.Offset16	IF.Offset16	VCDBitSet	0000	0x0000	0	0	16
IF.RegA	IF.RegA	String	N/A	0x7ff8000000000000	□	0	64
IF.RegB	IF.RegB	String	N/A	0x7ff8000000000000	□	0	64
IF.RegC	IF.RegC	String	N/A	0x7ff8000000000000	□	0	64
IF.IntA	IF.IntA	VCDBitSet	00000000	0x00000000	0	0	32
IF.IntB	IF.IntB	VCDBitSet	00000000	0x00000000	0	0	32
IF.IntC	IF.IntC	VCDBitSet	00000000	0x00000000	0	0	32
IF.DoNewPC	IF.DoNewPC	VCDBitSet	00000000	0x00000000	0	0	32
IF.NewPC	IF.NewPC	VCDBitSet	00000000	0x00000000	0	0	32
ID							
EX							
MEM							
WB							
WB.status	WB.status	String	RUN	0x7ff8000000000000	□	0	64
WB.PC	WB.PC	VCDBitSet	00001000	0x00001000	4,096	4096	32
WB.IR	WB.IR	VCDBitSet	0C000000	0x0C000000	201,326,592	201326592	32
WB.Instr	WB.Instr	String	JAL $00001004	0x7ff8000000000000	□	0	64
WB.OP	WB.OP	VCDBitSet	03	0x03	3	3	8
WB.Func	WB.Func	VCDBitSet	0000	0x0000	0	0	16
WB.Offset16	WB.Offset16	VCDBitSet	0000	0x0000	0	0	16
WB.RegA	WB.RegA	String	N/A	0x7ff8000000000000	□	0	64
WB.RegB	WB.RegB	String	N/A	0x7ff8000000000000	□	0	64
WB.RegC	WB.RegC	String	N/A	0x7ff8000000000000	□	0	64
WB.IntA	WB.IntA	VCDBitSet	00000000	0x00000000	0	0	32
WB.IntB	WB.IntB	VCDBitSet	00000000	0x00000000	0	0	32
WB.IntC	WB.IntC	VCDBitSet	00000000	0x00000000	0	0	32

Figure 4-7: Example of the Signal Explorer

The Signal Explorer is an example when the View Object and View Layout model failed. In this instance, there existed an excellent Swing component named JTreeTable and the Signal Explorer needed to display data obtained from the entire Structure.

The Signal Explorer was originally intended to be a View Layout; the Signal Explorer would place View Objects in their appropriate position in a layout that demonstrated the hierarchal relationship between the signals. However, the JTreeTable, a third-party Swing component, provided a simple and sophisticated visualization that could not be matched by standard Swing components. Unfortunately the JTreeTable couldn't be a View Layout because the entries could not accept JPanels, and it couldn't be a View Object because it presented the entire Structure, not only one signal.

This type of visualization can attach itself to the VISTA Environment to gain access to the simulation data and notifications of changes in the current simulation time. The developer can mimic the methods that the View Layouts and View Objects use to perform these tasks. However, many of the other features of the VISTA Environment, like the ability to transfer data through drag-and-drop, will depend on the implementation of the Graphical User Interface (see Chapter 5).

4.4 Global Time Listener Interface

In order to receive notification from the VISTA Environment of changes in the current simulation time, a component must implement the Global Time Listener Interface and register itself with the VISTA Environment. Fortunately for developers using the View Layout and View Object model, this is built into the model. However, alternative visualization design models must implement this interface if they want to receive notification of changes in the current simulation time.

A Global Time Listener is a subclass of the traditional Java event listener for the VISTA Environment. In order to implement this listener interface, the component must contain two public methods: `GlobalTimeAdded` and `GlobalTimeChanged`. The `GlobalTimeAdded` method will be called whenever the Runtime Data Manager detects that new simulation data has been imported by the Runtime Import Manager. The `GlobalTimeChanged` method will be called whenever the current simulation time changes in the VISTA Environment.

Objects that implement the Global Time Listener can add themselves to the Global Time Listener Event Queue by calling the `Vista.registerGlobalTimeListener` and remove themselves by calling `Vista.deregisterGlobalTimeListener`.

Chapter 5:

Graphical User Interface Implementation

This final chapter details the implementation of the VISTA Graphical User Interface (see Figure 5-1). The GUI is responsible for managing the presentation of the VISTA Environment to the user and for handling the user's interaction with the VISTA Environment. These components include top-level containers for View Layouts, menus, toolbars and other general graphical user interface components that serve as many of the functional aspects of the VISTA Environment.

Managing Visualizations Using a Multiple Document Interface discusses the Java implementation of the MDI and the benefits of using this interface in Java applications. Also, this section describes how the VISTA GUI handles user interaction to present and modify the visualizations of simulation data.

Finally, **Modifying and Examining the State of the VISTA Environment** describes how users can interact with the VISTA Environment through the VISTA GUI and how the VISTA GUI presents the state of the VISTA Environment to the user. This section details the "actions" that users can use to change the VISTA Environment, the user interface of these actions (e.g. menus and toolbars), and the presentation of the state of the VISTA Environment through a status bar.

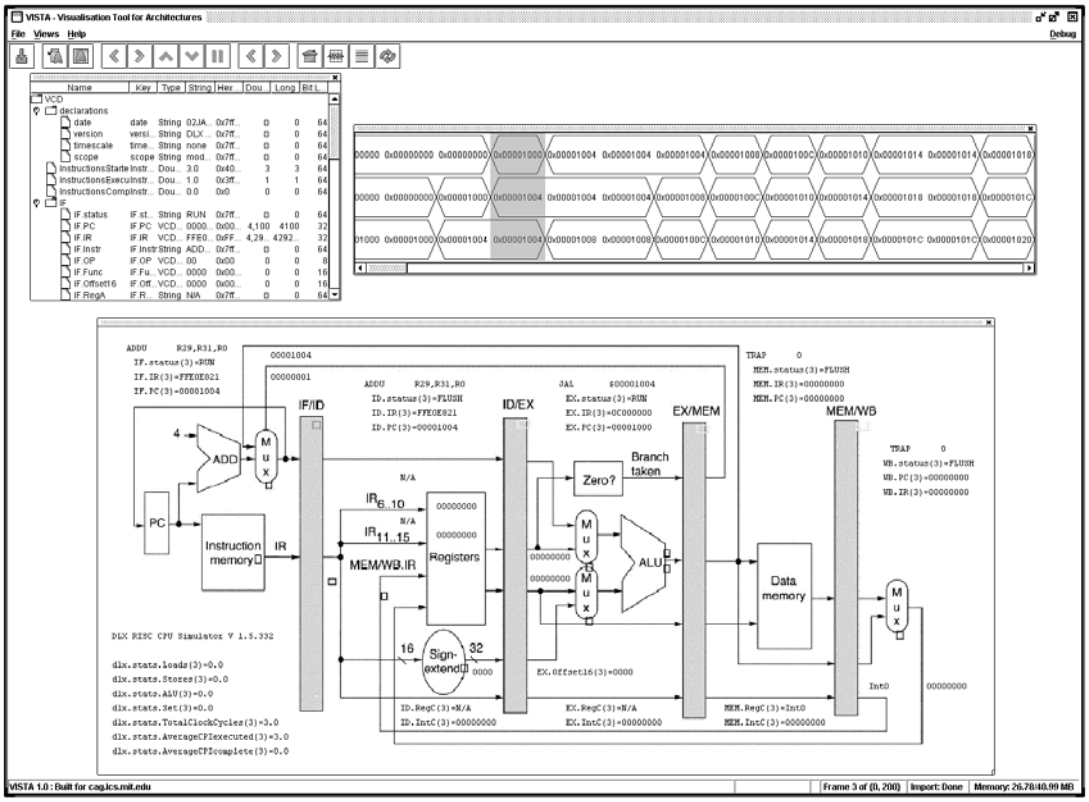


Figure 5-1: The Graphical User Interface of the VISTA Environment

The components that comprise the Graphical User Interface are responsible for managing the presentation of the VISTA Environment to the user and for handling the user's interaction with the VISTA Environment. These components include top-level containers for View Layouts, menus, toolbars and other general graphical user interface components that serve as many of the functional aspects of the VISTA Environment.

5.1 Managing Visualizations Using a Multiple Document Interface

The VISTA Graphical User Interface is presented using the Java implementation of the Multiple Document Interface. The VISTA GUI relies on the MDI to provide top-level containers to hold the visualizations, notify the visualization of user interaction and to transfer data between visualizations.

The MDI is one of several different objects that Java can use to present a Graphical User Interface. The two other objects are the Applet and the Frame. The Applet is useful for applications intended to be embedded in a web page and the Frame object becomes programmatically cumbersome in a cross-platform multi-frame

environment. The MDI, however, is platform independent, provides users with a single GUI to the VISTA Environment and allows the greater control over the presentation visualizations of simulation data.

5.1.1 View Layouts Containers

View Layouts (see 4.2) are implemented as Java panels and require a top-level container—a frame or main window—to control their boundaries. Since the VISTA GUI uses a Multiple Document Interface (MDI), the top-level container for a View Layout is an internal frame.

An added benefit of using the Java MDI and the internal frames comes from the MDI being implemented using platform-independent code. This implementation allows for features in the internal frames that normal Java frames cannot provide in a cross-platform environment. For various reasons, Java cannot implement certain Frame commands nor can it guarantee that certain Frame commands will behave the same across all platforms.

The VISTA Environment uses this increased programmatic control over the internal frame to maximize the amount of screen area that is used for the visualization of data. The VISTA Environment can enable the "palette" frame boarders on the internal frame (see Figure 5-2) in order to use screen area more efficiently than the standard frame.

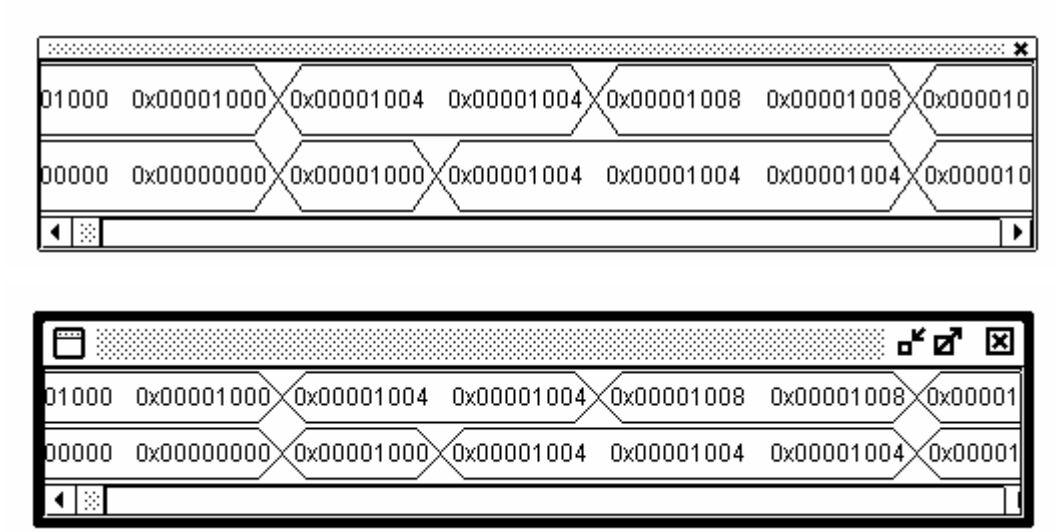


Figure 5-2: A Standard Frame and a Palette Frame

The VISTA Environment uses the Multiple Document Interface controls to enable the "palette" frame boarders that reduce the amount of screen area that the frame uses. The "palette" frame (top) uses screen area more efficiently than the standard frame (bottom).

5.1.2 Dispatching Mouse Events

The internal frame of the MDI, like any top-level container in Java, has multiple panes and stores the View Layout component in the Content Pane (see Figure 5-3). All top-level containers have a Glass Pane above the Content Pane that can catch events or paint over an area within the container. The VISTA GUI makes use of this Glass Pane to catch the user's mouse events and perform the desired operation on the corresponding View Object or View Layout.

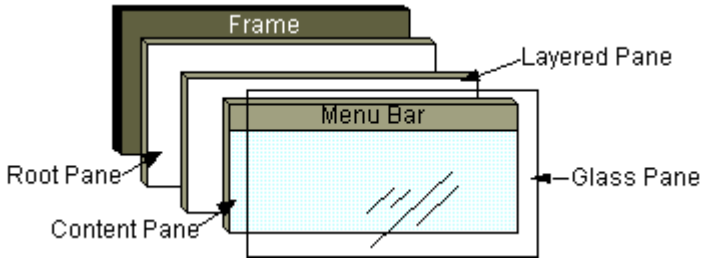


Figure 5-3: The Structure of a Top-Level Container

The internal frame of the MDI, like any top-level container in Java, has multiple panes and stores the View Layout component in the Content Pane. The Graphical User Interface makes use of the Glass Pane to catch the user's mouse events and perform the desired operation on the corresponding View Object or View Layout (Figure from Sun Microsystems, Inc [17]).

While this task is straightforward when using the Java Frame, the internal frame of the MDI presents some difficulties that have been documented in the Sun's Bug Database. For instance, once an internal frame is deactivated (either it loses focus or it is iconified), the modified Glass Pane is replaced with a fresh Glass Pane [18] . The VISTA GUI works around this problem by replacing the fresh Glass Pane with a modified Glass Pane once the internal frame is reactivated.

5.1.3 Drag and Drop Implementation

Drag-and-drop allows the user to transfer information between visualizations in the VISTA Environment. In the VISTA GUI, drag-and-drop is implemented using Swing's Transfer Handler interface and is used transfer a signal's key between visualization. This implementation of Drag and Drop enables users to create new View Objects in a View Layout and to attach signals to View Objects.

The Swing Transfer Handler interface (see Figure 5-4) is used to transfer data between Swing components. Swing components that contain objects that implement this interface can exchange data through the system clipboard or by drag-and-drop operations. Every View Object contains a transfer handler that can send and receive a key—a unique case-sensitive string identifier—that corresponds to the signal being visualized. Every View Layout also has a transfer handler that can receive a signal's key and create the appropriate View Object for that signal the View Layout.

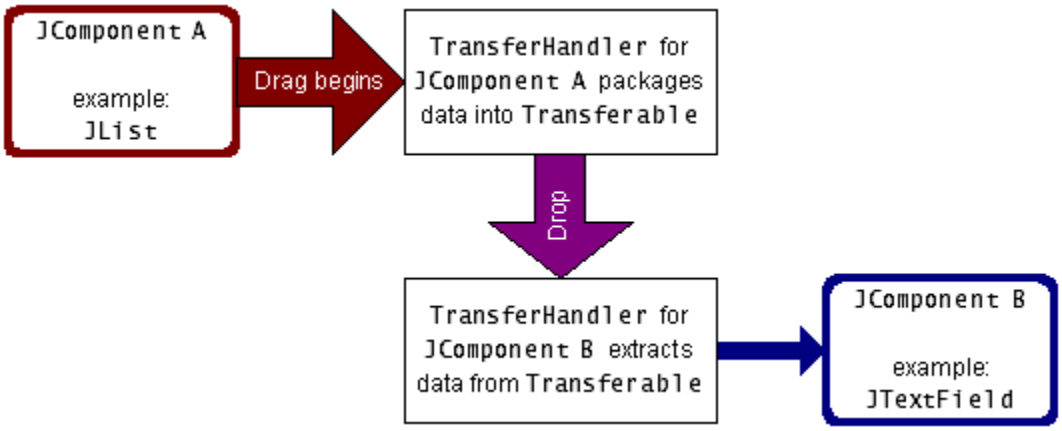


Figure 5-4: The Use of Swing Transfer Handlers in Drag and Drop

The Swing Transfer Handler interface is used to transfer data between Swing components. Swing components that contain objects that implement this interface can exchange data through the system clipboard or by drag-and-drop operations. (Figure from Sun Microsystems, Inc [16])

The Transfer Hander interface provides a simple way to exchange data between Swing components and native applications. However, the Transfer Handler does not provide a "drop" location reference when the data is transferred using the drag-and-drop operation. Since the "drop" location of a View Object into a View Layout is important, a View Layout also uses the traditional Java Abstract Window Toolkit (AWT) Drop Target Adapter to monitor the location of the "drop".

5.2 Modifying and Examining the State of the VISTA Environment

In the VISTA Environment, users will want to execute a set of commands and modify parameters in order to change the state of the VISTA Environment. Also, the VISTA Environment is responsible for providing users with information to that describes its state.

In order to execute commands, there are a set of user actions that contain almost all of commands a user could want to execute. These actions are packaged into the Toolbars and Menus in the VISTA GUI. The VISTA GUI provides to users the VISTA Environment state information through a status bar.

Finally, users can access and adjust parameters through a set of Property Boxes. These Property Boxes are hard-coded into the VISTA Environment and developers need

to create and modify Property Boxes to allow users to access parameters at run-time. However, these Property Boxes provide developers with numerous tools that allow developers to create them quickly.

5.2.1 User Actions in the VISTA Environment

While using the VISTA Environment, the user will need to execute commands to change the state of the VISTA Environment. In the VISTA GUI, these commands are encapsulated within User Action object. The set of User Action objects comprises of almost all of the commands a user could want to execute and more commands can be easily added.

This set of User Action objects include common commands such as importing simulation data, creating new visualizations, and changing the current simulation time. In order to create a new User Action (see Figure 5-5), a developer must write Java code that specifies how the action interacts with the VISTA Environment and update the property bundle file that contains information provided to the GUI about the User Action. This model separates the functionality of the action (which is independent of the VISTA GUI) and the display information (which is independent of the rest of the VISTA Environment).

```
public void actionPerformed(ActionEvent event) {
      frameNo = Vista.getGlobalTime();
      refresh();
      if (enabled) {
            Vista.setGlobalTime(Vista.getGlobalTime() + 1);
      }
}
```

```
### StepForwardAction ###

StepForwardAction.NAME=Step Forward
StepForwardAction.SMALL_ICON=Forward24.gif
StepForwardAction.MNEMONIC_KEY=F
StepForwardAction.ACCELERATOR_KEY=control RIGHT
StepForwardAction.LONG_DESCRIPTION=Step Forward
StepForwardAction.SHORT_DESCRIPTION=Step Forward
```

Figure 5-5: Example of a User Action's Java Code and Description Entry

In order to create a User Action, a developer must write Java code that specifies how the action interacts with the VISTA Environment (top) and update the property bundle file that contains information provided to the GUI (bottom).

Once the User Action has been created, it can be packaged in Menus or Toolbars (see Figure 5-6). The description entry in the property bundle file provides the GUI with information that determines the button icon for the toolbar, the text label for the menu and shortcut keystroke binding in the VISTA GUI. The Java code is performed when the user clicks on the button in the toolbar, chooses the item in a menu or presses the shortcut keystroke.

Figure 5-6: Menus and Toolbar from the VISTA Graphical User Interface

User Actions can be packaged in Menus or Toolbars. The description entry in the property bundle file provides the GUI with information that determines the button icon for the toolbar, the text label for the menu and shortcut keystroke binding in the VISTA GUI. The Java code is performed when the user clicks on the button in the toolbar, chooses the item in a menu or presses the shortcut keystroke.

5.2.2 Providing VISTA State Information

The VISTA Environment has state information that the user can change through the set of User Actions. In turn, the VISTA GUI is responsible for presenting changes in this state information back to the user. This is accomplished by use of a status bar.

Currently, the status bar provides users with state information pertaining the current simulation time, the range of simulation time imported, the status of the Runtime Import Manager, the amount of memory that the Java Virtual Machine is using and the amount of memory available to the JVM.

Figure 5-7: VISTA Graphical User Interface Status Bar

The status bar provides users with state information pertaining the current simulation time, the range of simulation time imported, the status of the Runtime Import Manager, the amount of memory that the Java Virtual Machine is using and the amount of memory available to the JVM.

5.2.3 Property Boxes

The VISTA GUI allows users to access and adjust parameters through a set of Property Boxes. These Property Boxes are hard-coded into the VISTA GUI and developers need to create and modify Property Boxes to allow users to access parameters at run-time. However, these Property Boxes provide developers with numerous tools that allow developers to create them quickly.

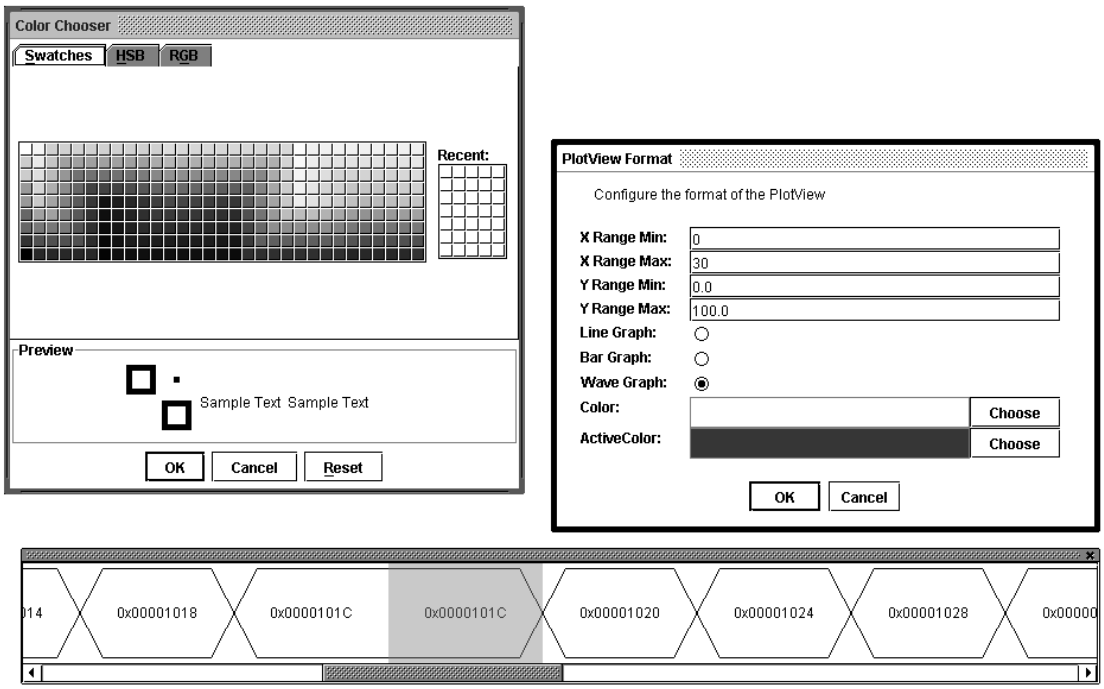


Figure 5-8: A Plot View Object with a Property Box and Color Chooser

The VISTA GUI allows users to access and adjust parameters through a set of Property Boxes. In this figure, a user is adjusting the format of a Plot View Object (bottom). The Property Box (top, right) allows users to set various parameters of the Plot View Object, and will launch other tools—like a color chooser (top, left)—when required.

A Property Box can be used examine and adjust almost any type of object in Java. The Property Box component includes methods to add text fields, check boxes, color choosers and file choosers. The design and code for the Property Boxes are based on components from the Ptolemy project at the University of California, Berkley [19].

Chapter 6:

Conclusion and Future Work

Since the creation of the VISTA Environment by Mathew Jack in 2002, the VISTA Environment has changed significantly. Although the initial goals of the VISTA Environment remain the same, the data models and the main components of the VISTA Environment have changed drastically. There are two immediate problems that need to be addressed involving JVM memory usage and the graphical user interface. Also there is future work with the VISTA Environment that would involve modifying the data models and adding other functionality to the main components in VISTA.

The Data Manager currently has a lofty storage overhead for the simulation data. When VISTA imports a Value Change Dump file, the Data Manager requires approximately ten times the amount of memory to store the simulation data in memory than was required to store it on disk.

The graphical user interface has artifact problems on the Macintosh platform, responds slower than expected and requires a large amount of memory. Since the Macintosh artifact problems haven't been experienced on Linux and Windows environments, this is probably just a bug in the Macintosh JVM. The responsiveness, however, warrants investigation into other Java window managers. Finally, the current configuration of the Plot View has an entire bit-mapped image of the plot stored in memory, and this bitmap becomes very large once users zoom in.

The current data types in the VISTA Environment allow users to import and visualize signal data that is represented as bit data, integers and real numbers, or strings of characters. While the VISTA Environment can visualize this data in multiple different

ways, it cannot manipulate this data. There are no functions to combine data from various sources and derive a new piece of data.

The data visualizations can display only one piece of signal data. The simplicity of having "one signal" correspond to "one visual representation" is a design decision that I struggled with numerous times. The argument is that users can build complex visualizations out of simple visualization objects. Future work should experiment with designing simple visualizations that represent multiple pieces of signal state.

The VISTA Environment has been used to visualize data obtained from two different Value Change Dump sources: the SyCHOSys simulation and a DLX simulation. Both of these simulations made use of the Value Change Dump simulation parser included with the VISTA Environment. While the VISTA Environment provides tools for users to develop new parsers in Java, potential users have been groaning about the prospect of writing a new parser. Instead, the VISTA Environment would benefit from a Graphical User Interface toolset that allowed users to create parsers

Chapter 7:

Bibliography

[1] Jack St. Clair Kilby. Texas Instruments. 25 Dec. 2003 <http://www.ti.com/corp/docs/kilbyctr/jackstclair.shtml>.

[2] Moore, Gordon E.. "Cramming More Components onto Integrated Circuits." Electronics 19 Apr. 1965. 25 Dec. 2003 <ftp://download.intel.com/research/silicon/moorespaper.pdf>.

[3] Intel Microprocessor Quick Reference Guide. Intel Corporation. 25 Dec. 2003 <http://www.intel.com/pressroom/kits/quickreffam.htm#i486>

[4] Howstuffworks.com. "How Microprocessors Work." 15 Jan 2004. <http://computer.howstuffworks.com/microprocessor1.htm>

[5] Weaver, Chris, et al. "Performance Analysis Using Pipeline Visualization." Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2001), Nov. 2001.

[6] Marwedel, Peter, and Birgit Sirocic. "Multimedia components for the visualization of dynamic behavior in computer architectures." Proceedings of the Workshop on Computer Architecture Education (WCAE'03), 8 June 2003.

[7] LS XII / Software. University of Dortmund – Department of Computer Science. 29 Dec. 2003 <http://ls12.cs.uni-dortmund.de/ravi/>

[8] Branovic, Irina, Roberto Giargi, and Antonio Prete. "Web-based training on computer architecture: The case of JCachesim." Proceedings of the Workshop on Computer Architecture Education (WCAE'02), 26 May 2002.

[9] Burger, Doug and Todd M. Austin. "The SimpleScalar Tool Set, Version 2.0." University of Wisconsin Computer Sciences Technical Report #1342, June 1997.

[10] Coe, P.S., et al. "A Hierarchical Computer Architecture Design and Simulation Environment," ACM TOMACS, Vol. 8, No. 4, 1998, pp. 431-446.

[11] DLX with Parallel Function Units (and labels). School of Informatics, University of Edinburgh. 29 Dec. 2003 <http://www.icsa.informatics.ed.ac.uk/cgi-bin/hase/ident.pl?Dlx_v2>

[12] Bosch, Robert P. Jr.. "Using Visualization to Understand The Behavior Of Computer Systems." Ph.D. Dissertation, Stanford University, August 2001.

[13] Visualizing Complex Systems: The Rivet Project. Stanford Computer Graphics Laboratory. 29 Dec. 2003 <http://graphics.stanford.edu/projects/rivet/>

[14] The Institute of Electrical and Electronics Engineers, Inc. "IEEE Standard Description Language Based on the VERILOG Hardware Description Language, 1364-1995." New York: IEEE Standard Office, 1996.

[15] Milne, Philip. "Creating TreeTables in Swing." 14 Jan 2004. <http://java.sun.com/products/jfc/tsc/articles/treetable1/>

[16] Sun Microsystems, Inc. "How to Use Drag and Drop and Data Transfer." 15 Jan 2004. <http://java.sun.com/docs/books/tutorial/uiswing/misc/dnd.html>

[17] Sun Microsystems, Inc. "How to Use Root Panes." 15 Jan 2004. <http://java.sun.com/docs/books/tutorial/uiswing/components/rootpane.html>

[18] Sun Microsystems, Inc. "bug id: 4222821 Iconifying a JInternalFrame causes its GlassPane to lose state." 15 Jan 2004. <http://developer.java.sun.com/developer/bugParade/bugs/4222821.html>

[19] Department of EECS, UC Berkeley. "Ptolemy." 15 Jan 2004. <http://ptolemy.eecs.berkeley.edu/>