Multiple PCI Bus Simulator & GUI Design and Implementation

Project report

Supervisor: Mike Macgregor

Author: Yile Sun

Email: yile@ualberta.ca

MSC in internetworking

University of Alberta

September 2007

Abstract

In modern computer system, the CPU and memory communicate by fast system bus; for example, the network cards (NICs) use DMA transfers over the PCI bus to place incoming data in buffers and in main memory or to transmit data from buffers in the main memory.

The Peripheral Component Interconnect, or PCI Standard (in practice almost always shortened to PCI) is a high performance bus for interconnecting chips, expansion boards, and processor/memory subsystems.

In this project, we implement a multiple PCI Buses (up to 6 buses) simulator in the pure software environment with a user-friendly GUI; this simulator includes two parts: the simulation function and the GUI.

The simulation function provides some basic PCI bus simulations and generates output data & files. During the design of this PCI Multi-bus simulation system, it is rather important to check that PCI bus and PCI-to-PCI Bridge (P2PB) are able to transfer data to and from the attached devices at the rate required for correct operation of the device.

The GUI provides widgets for both input and output. In input widgets, we can specify bus devices independent parameters, simulation information, and input & out files management information and bus devices parameters. In the output widget, we provide simulator control functions and interfaces to exhibit both input and output information. The output widget also provides a connector to generate graphs based on the simulation results by GNU PLOT. We select Qt to write our codes of the GUI; Qt is a cross-platform application development framework, widely used for the development of GUI programs. GUI codes are written within Qt designer 3 in red hat 9 Linux systems.

This report includes two sections: first section provides some background acknowledge; second section describes the principle about the simulator and the implementation about the GUI.

Key words

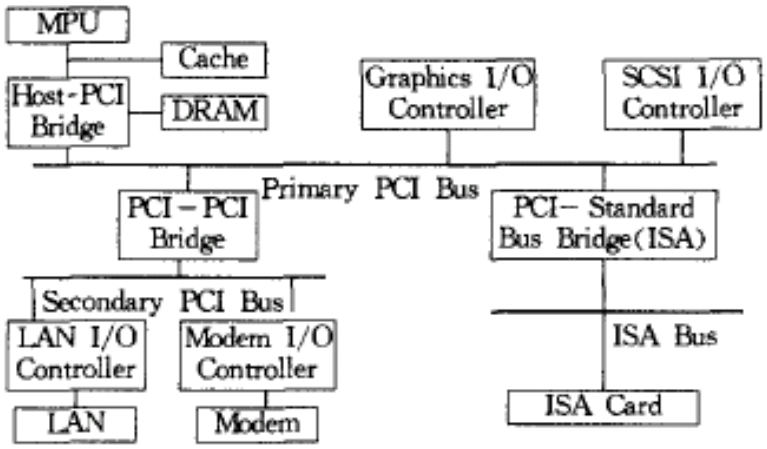
Computer Bus, PCI, PCI Bridge (P2PB), PCI Simulator, Arbitrator, Priority Algorithm, GUI, Qt, Qt Designer, GNU PLOT

Background

1. PCI

The Peripheral Component Interconnect, or PCI Standard (in practice almost always shortened to PCI), specifies a computer bus for attaching peripheral devices to a computer motherboard. It originated at Intel in the early 1990s as a standard method of interconnecting chips on a board. It was later adopted as an industry standard administered by the PCI Special Interest Group or "PCI SIG". Under the PCI SIG the definition of PCI was extended to define a standard expansion bus interface connector for add-in boards.

PCI was first adopted for use in personal computers in about 1994 with Intel's introduction of the "Saturn" chipset and "Alfredo" motherboard for the 486 processor. With the introduction of chipsets and motherboards for the Intel Pentium processor, PCI largely replaced earlier bus architectures such as EISA, VL, and Micro Channel. The ISA bus has initially continued to co-exist with PCI for support of "legacy" add-in boards that don't require the high performance of the PCI bus. But as legacy boards are redesigned, PCI is expected to completely replace ISA as well.



Architecture of multiple PCI buses

PCI is a synchronous bus architecture with all data transfers being performed relative to a system clock (CLK). The initial PCI specification permitted a maximum clock rate of 33 MHz allowing one bus transfer to be performed every 30 nanoseconds. Later, Revision 2.1 of the PCI specification extended the bus definition to support operation at 66 MHz, but the vast majority of today's personal computers continue to implement a PCI bus that runs at a maximum speed of 33 MHz.

PCI implements a 32-bit multiplexed Address and Data bus (AD[31:0]). It architects a means of supporting a 64-bit data bus through a longer connector slot. At 33 MHz, a 32-bit slot supports a maximum data transfer rate of 132 Mbytes/sec, and a 64-bit slot supports 264 Mbytes/sec.

The multiplexed Address and Data bus allows a reduced pin count on the PCI connector that enables lower cost and smaller package size for PCI components. Typical 32-bit PCI add-in boards use only about 50 signals pins on the PCI connector of which 32 are the multiplexed Address and Data bus. PCI bus cycles are initiated by driving an address onto the AD [31:0] signals during the first clock edge called the *address phase*. The address phase is signaled by the activation of the FRAME# signal. The next clock edge begins the first of one or more *data phases* in which data is transferred over the AD [31:0] signals.

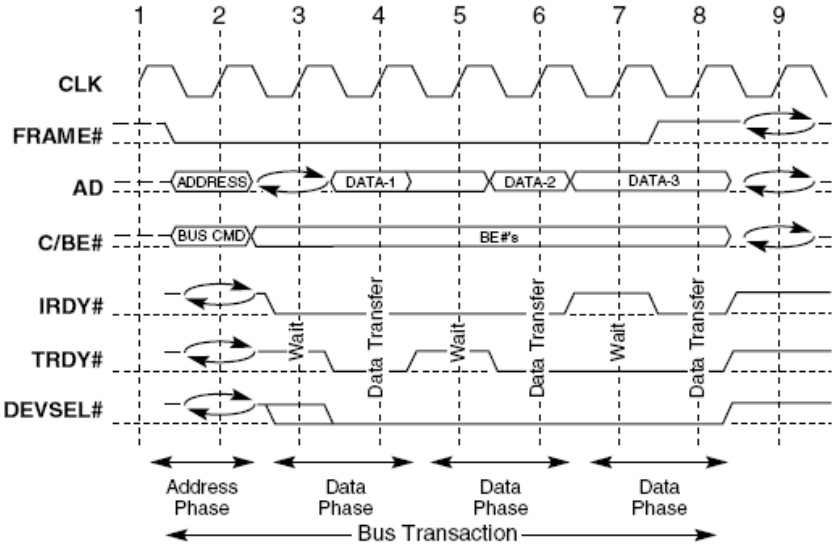
In PCI terminology, data is transferred between an initiator that is the bus master, and a target that is the bus slave. The initiator drives the C/BE [3:0]# signals during the address phase to signal the type of transfer (memory read, memory write, I/O read, I/O write, etc.). During data phases the C/BE [3:0]# signals serve as byte enable to indicate which data bytes are valid. Both the initiator and target may insert wait states into the data transfer by deasserting the IRDY# and TRDY# signals. Valid data transfers occur on each clock edge in which both IRDY# and TRDY# are asserted.

A PCI bus transfer consists of one address phase and any number of data phases. I/O operations that access registers within PCI targets typically have only a single data phase. Memory transfers that move blocks of data consist of multiple data phases that read or write multiple consecutive memory locations. Both the initiator and target may terminate a bus transfer sequence at any time. The initiator signals completion of the bus transfer by deasserting the FRAME# signal during the last data phase. A target may terminate a bus transfer by asserting the STOP# signal. When the initiator detects an active STOP# signal, it must terminate the current bus transfer and re-arbitrate for the bus before continuing. If STOP# is asserted without any data phases completing, the target has issued a *retry*. If STOP# is asserted after one or more data phases have successfully completed, the target has issued a disconnect.

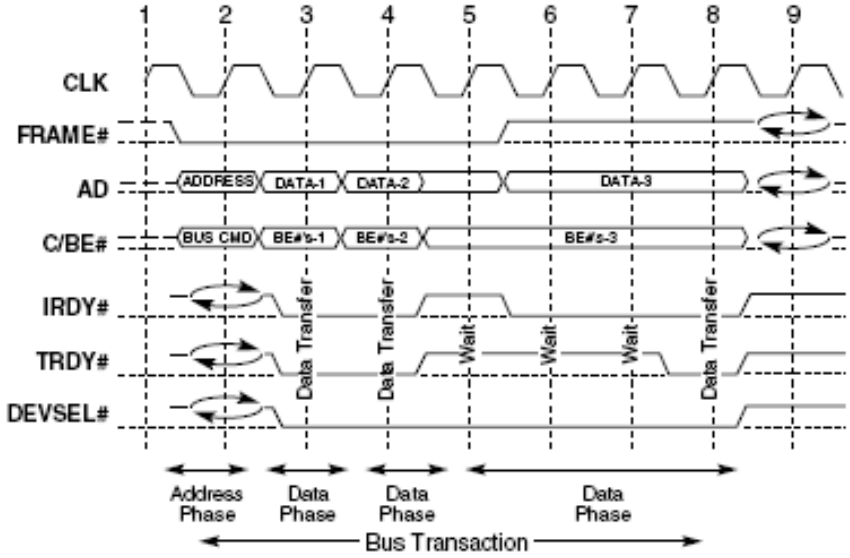
Initiators arbitrate for ownership of the bus by asserting a REQ# signal to a central arbiter. The arbiter grants ownership of the bus by asserting the GNT# signal. REQ# and GNT# are unique on a per slot basis allowing the arbiter to implement a bus fairness algorithm. Arbitration in PCI is "hidden" in the sense that it does not consume clock cycles. The current initiator's bus transfers are overlapped with the arbitration process that determines the next owner of the bus.

Generally speaking, a transaction on the PCI bus involves an address phase, in which an address is transferred, followed by a sequence of one or more data phases, in each of which a data item is transferred. Each transaction is initiated by a bus device acting as master and involves another participant known as the target. Data transfers may be oriented towards the master or towards the target; with the corresponding transactions respectively being denoted read and write transactions. The address specified at the start of the transaction is the address within the target of the first data item transferred.

There are two figures about basic PCI read and write function.



PCI bus (32bit) Basic Read Operation



PCI bus (32bit) Basic Write Operation

PCI supports a rigorous auto configuration mechanism. Each PCI device includes a set of configuration registers that allow identification of the type of device (SCSI, video, Ethernet, etc.) and the company that produced it. Other registers allow configuration of the device's I/O addresses, memory addresses, interrupt levels, etc.
Although it is not widely implemented, PCI supports 64-bit addressing. Unlike the 64-bit data bus option, which requires a longer connector with an additional 32-bits of data signals, 64-bit addressing can be supported through the base 32-bit connector. *Dual Address Cycles* are issued in which the low order 32-bits of the address are driven onto the AD[31:0] signals during the first address phase, and the high order 32-bits of the address (if non-zero) are driven onto the AD[31:0] signals during a

second address phase. The remainder of the transfer continues like a normal bus transfer.

PCI defines support for both 5 Volt and 3.3 Volt signaling levels. The PCI connector defines pin locations for both the 5 Volt and 3.3 Volt levels. However, most early PCI systems were 5 Volt only, and did not provide active power on the 3.3 Volt connector pins. Over time more use of the 3.3 Volt interface is expected, but add-in boards which must work in older legacy systems are restricted to using only the 5 Volt supply. A "keying" scheme is implemented in the PCI connectors to prevent inserting an add-in board into a system with incompatible supply voltage.

Although used most extensively in PC compatible systems, the PCI bus architecture is processor independent. PCI signal definitions are generic allowing the bus to be used in systems based on other processor families.

PCI includes strict specifications to ensure the signal quality required for operation at 33 and 66 MHz. Components and add-in boards must include unique bus drivers that are specifically designed for use in a PCI bus environment. Typical TTL devices used in previous bus implementations such as ISA and EISA are not compliant with the requirements of PCI. This restriction along with the high bus speed dictates that most PCI devices are implemented as custom ASICs.

The higher speed of PCI limits the number of expansion slots on a single bus to no more than 3 or 4, as compared to 6 or 7 for earlier bus architectures. To permit expansion buses with more than 3 or 4 slots, the PCI SIG has defined a *PCI-to-PCI Bridge* mechanism. PCI-to-PCI Bridges are ASICs that electrically isolate two PCI buses while allowing bus transfers to be forwarded from one bus to another. Each bridge device has a "primary" PCI bus and a "secondary" PCI bus. Multiple bridge devices may be cascaded to create a system with many PCI buses

2. GUI

A GUI is a graphical (rather than purely textual) user interface to a computer. The term came into existence because the first interactive user interfaces to computers were not graphical; they were text-and-keyboard oriented and usually consisted of commands you had to remember and computer responses that were infamously brief. The command interface of the DOS operating system (which you can still get to from your Windows operating system) is an example of the typical user-computer interface before GUIs arrived. An intermediate step in user interfaces between the command line interface and the GUI was the non-graphical menu-based interface, which let you interact by using a mouse rather than by having to type in keyboard commands.

Today's major operating systems provide a graphical user interface. Applications typically use the elements of the GUI that come with the operating system and add

their own graphical user interface elements and ideas. A GUI sometimes uses one or more metaphors for objects familiar in real life, such as the desktop, the view through a window, or the physical layout in a building. Elements of a GUI include such things as: windows, pull-down menus, buttons, scroll bars, iconic images, wizards, the mouse, and no doubt many things that haven't been invented yet. With the increasing use of multimedia as part of the GUI, sound, voice, motion video, and virtual reality interfaces seem likely to become part of the GUI for many applications. A system's graphical user interface along with its input devices is sometimes referred to as its "look-and-feel."

The GUI familiar to most of us today in either the Mac or the Windows operating systems and their applications originated at the Xerox Palo Alto Research Laboratory in the late 1970s. Apple used it in their first Macintosh computers. Later, Microsoft used many of the same ideas in their first version of the Windows operating system for IBM-compatible PCs.

When creating an application, many object-oriented tools exist that facilitate writing a graphical user interface. Each GUI element is defined as a class widget from which you can create object instances for your application. You can code or modify prepackaged methods that an object will use to respond to user activity. Qt is an excellent tool to design GUI.

3. Qt

In this project, we select Qt 3.3 to implement the GUI; Qt is a cross-platform application development framework, widely used for the development of GUI programs. Qt uses non-standard C++, extended by an additional pre-processor that generates the standard C++ code, which is necessary to implement Qt's extensions. Qt has defined a large number of classes for our GUI programming, all those classes can be found in http://doc.trolltech.com/3.3/classes.html, and all functions can be found in http://doc.trolltech.com/3.3/functions.html. Qt class hierarchy class table can be found in http://doc.trolltech.com/3.3/classchart.html.

Qt introduces a unique inter-object communication mechanism called “signals and slots”. Qt has excellent cross-platform support for 2D and 3D graphics, internationalization, SQL, XML (ui documents are written in XML), unit testing. Qt applications can be built visually using Qt designer, a flexible user interface builder with support for IDE integrations, Qt designer will be discussed in the coming segment.

Qt is supported on the following platforms:
(1) MS/Windows -- 95, 98, NT 4.0, ME, 2000, and XP
(2) Unix/X11 -- Linux, Sun Solaris, HP-UX, Compaq Tru64 UNIX, IBM AIX, SGI IRIX and a wide range of others

(3) Macintosh -- Mac OS X
(4) Embedded -- Linux platforms with frame buffer support.

Here are also some well-known applications about Qt: such as KDE--Popular desktop environment for Linux, Google Earth--Virtual Globe Program from Google.

4. Qt designer

Qt Designer is a tool for designing and implementing user interfaces built with the Qt multi-platform GUI toolkit. Qt designer enables rapid development of high-performance user interfaces with native look and feel across all supported platforms. Stand-alone, or integrated with IDEs like Microsoft Visual Studio. Qt Designer makes it easy to experiment with user interface design. At any time you can generate the code required to reproduce the user interface from the files Qt Designer produces, changing your design as often as you like.

Qt Designer helps you build user interfaces with layout tools that move and scale your widgets (controls in Windows terminology) automatically at runtime. The resulting interfaces are both functional and attractive, comfortably suiting your users' operating environments and preferences. Qt Designer supports Qt's signals and slots mechanism for type-safe communication between widgets. Qt Designer includes a code editor, which you can use to embed your own custom slots inside the generated code (ui documents). Those who prefer to separate generated code from hand crafted code can continue to use the sub classing approach pioneered in the first version of Qt Designer.

In Qt Designer, we need a .pro file to generate a project for our applications, the list of forms, the image collection and information about available databases and how to access them are included in this project together with extra information that Qt Designer needs to manage the project.

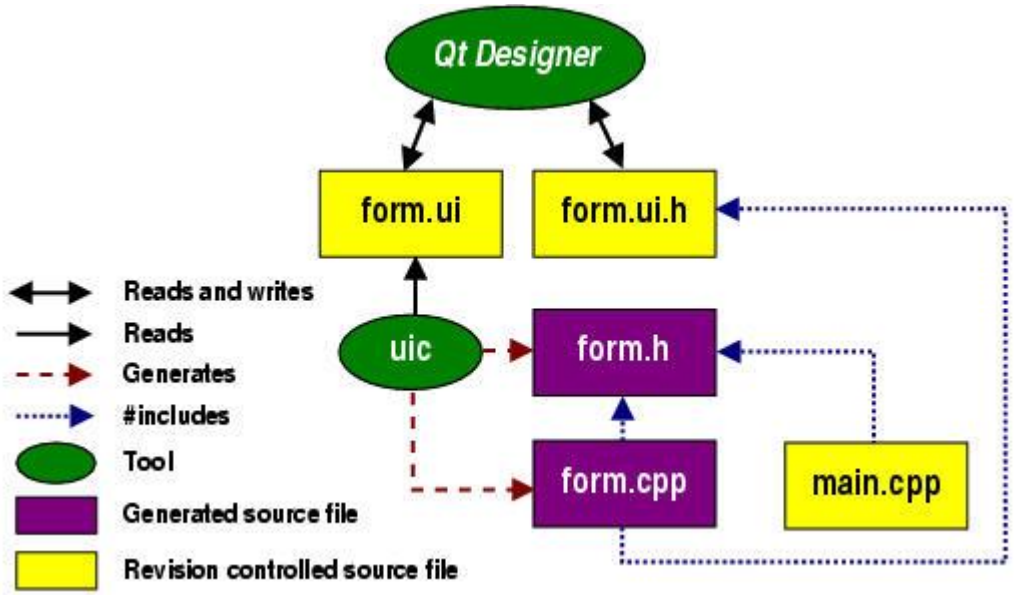
Qt Designer reads and writes .ui files. The user interface compiler, uic, creates both a header file, and an implementation file, from the .ui file. Typically main.cpp is used to instantiate the QApplication object and start off the event loop.

Qt Designer has two approaches to implement custom slots: ui.h and sub classing.
ui.h approach: we can write codes about custom slots directly inside the ui.h documents. The ui.h file has a special position among all other files. It is a shared source file that gets written and read by both the user and Qt Designer. As such it is an ordinary revision controlled source file and not generated by uic. Qt Designer's responsibility is to keep the file in sync with the custom slot definitions of the associated form.

Whenever the user adds a new slot to the form, *Qt Designer* adds a stub to the .ui.h

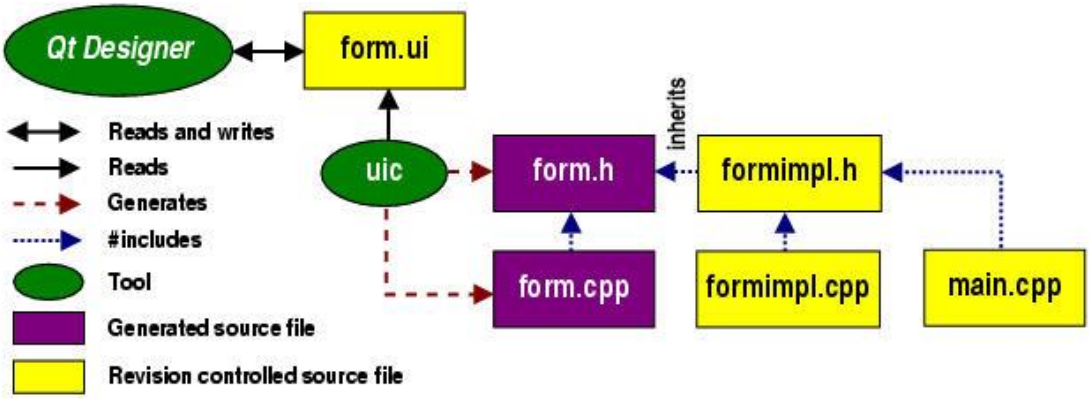
file. Whenever the user changes a custom slot's signature, Qt Designer updates the corresponding implementation. Whenever the user removes a custom slot, Qt Designer removes it from the ui.h file.

The ui.h approach is easy to apply and it makes codes looks clearly. However, there exist some problems with it. This approach just fits the situation that is not that complex; if we need coordinate with other codes, it becomes more difficult to combine them; I even found that the GUI became unstable when I put too many codes in a ui.h file. And there is another disadvantage: the ui.h and the ui file can be used to generate C++ .h file and .cpp file; if the ui.h file and the ui file change, we need to recompile them to generate the .h file and the .cpp file. The ui file is generated and managed by the Qt designer, we do not need care about it when we are coding, but the ui.h file can be modified inside the Qt designer; In conclusion, my suggestion is just define simple and fixed slots in the ui.h file.



ui approach

Sub classing approach: subclass is a more “C++” mode: leave the virtual function definition in the parent class, then use the heritage class to give the function implementation code; here is an example about how it works: we wrote an additional class FormImpl, which is split into the header file formimpl.h and the implementation file formimpl.cpp. The header file includes the uic generated form.h and re-implements all the custom slots. This is possible because uic generated custom slots are virtual. In addition to implementing custom slots, this approach gives the user a way to do extra initialization work in the constructor of the subclass, and extra cleanups in the destructor. In this project, I implemented both of these two approaches in my coding.



Sub classing approach

5. GNU PLOT

Gnu plot is a portable command-line driven interactive data and function plotting utility for UNIX, IBM OS/2, MS Windows, DOS, Macintosh, VMS, Atari and many other platforms. It was originally intended to allow scientists and students to visualize mathematical functions and data. It does this job pretty well, but it has grown to support many non-interactive uses, including web scripting and integration as a plotting engine for third-party applications like Octave. Gnu plot has been supported and under development since 1986.

Gnu plot supports many types of plots in either 2D or 3D. It can draw using lines, points, boxes, contours, vector fields, surfaces, and various associated text. It also supports various specialized plot types.

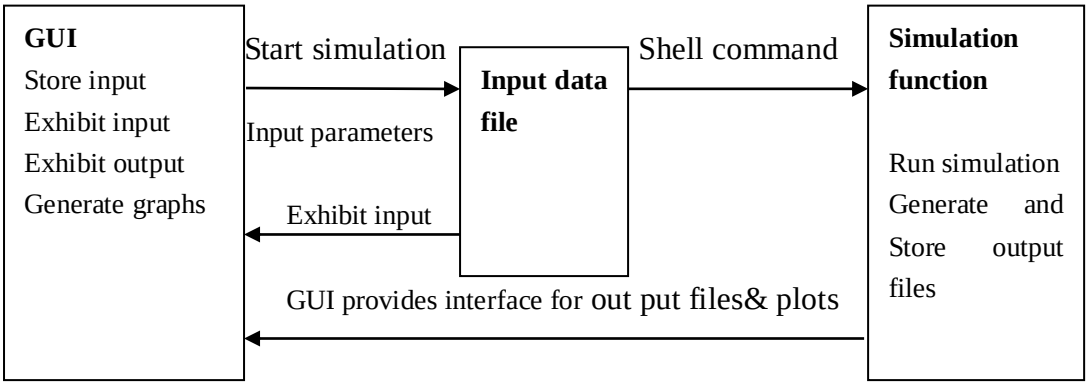
Gnu plot supports many different types of output: interactive screen terminals (with mouse and hotkey functionality), direct output to pen plotters or modern printers (including postscript and many color devices), and output to many types of file (eps, fig, jpeg, LaTeX, metafont, pbm, pdf, png, postscript, svg...) Gnu plot is easily extensible to include new devices. Recent additions include interactive terminals based on aquaterm (OSX) and wxWidgets (multiple platforms).

Included in our multiple PCI Buses simulation results are four text files, three of them can be used to generate graphs, and all generated graphs are saved as .png files by Gnu plot.

Multiple PCI Buses Simulator Implementation

The simulator can be separate into two parts: the simulation function and the GUI; the simulation function provides how the simulator works, it also generates and stores some wanted output files; the GUI provides a friendly visible user interface to specify inputs, all inputs will be written into a file with a specific formation, then a shell command will load this input file as the input of the simulation function; finally, the GUI also provides functions to store and generate output graphs.

There is a figure showing the relationship between each part of the simulator. The GUI manages the input file, and provides interface for output results; the simulation function completes the simulation process; this figure just shows the basic profile of the simulator; detailed structures and connection functions will be discussed soon. We will introduce the simulation function first, and then the GUI.

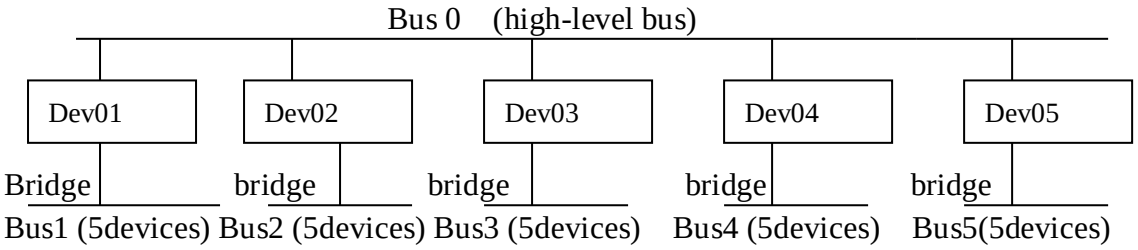


Basic relationship between the simulation function and the GUI

1. PCI simulation function

The simulator is based on the PCI Local Bus Specification and P2PB Architecture Specification. The simulator implemented in the pure software environment, which means it is completely hardware-independent; the simulator deals with regular data transfer transactions on the single PCI bus and multiple PCI buses by P2PB. It can simulate the operation of both the 32bit/64bit versions of PCI bus at synchronous clock frequencies: 33MHz, 66MHz, or 133MHz.

In the simulator, we implement a multiple PCI buses system with six buses, one high-level bus: bus0 and five secondary buses: bus1~bus5. There are five devices on each bus; all devices are supposed to be the same devices, while bus0 devices also support the bridge function; secondary bus can be activated only when related bus0 device is enabled and its bridge function is enabled.



Multiple PCI Buses Topology in the Simulator

The simulator makes the simplifying assumption that the first (N-1) data phases of an N data phase transaction involve transfer of data with the maximum width (4 bytes or 8 bytes) for the bus version concerned and that only the final transfer may involve fewer bytes (the final transfer contains the remainder of the ((transmitted data)/ (4 or 8)) bytes), to allow for data transfers whose length is not an integral multiple of 4 or 8 bytes respectively.

The PCI bus specification allows the target to delay the transfer of a data item by inserting wait states, each lasts one clock cycle, in any data phase. In more detail, the first data phase of a transaction can be extended to at most 16 clock cycles and each subsequent data phase to at most 8 clock cycles. The simulator allows the user to specify a maximum number of wait states for each device. Note that these wait states are specified for the device as master, although strictly speaking they describe the behavior of the device's target in the system under consideration. The user can also specify whether the actual number of wait states used in each data phase is selected deterministically or stochastically. With deterministic selection, exactly wait states are inserted in every data phase. With stochastic selection, the simulator will for each data phase select a random whole number as the number of wait states to be inserted. These features make it possible to simulate the behavior of devices with slow response and of targets such as main memory, which may exhibit access contention.

Our simulator will terminate the data transfer transaction when another master is attempting to gain control of the bus; and then bus arbitrator grants this new master to get control of the bus. The simulator implements the rules for handling the master's latency timer whose initial value is specified by the user. A single idle state lasting one clock cycle is always interpolated on the bus between the current transaction and its successor. The simulator does not support fast back-to-back access without interpolated idle states.

We try to use fix priority algorithm, rotating priority algorithm, and time quantum algorithm for the arbitrator; here is the simple introduction to these three algorithms.

Fix priority algorithm: it is simple and just arbitrates devices according to their fixed priority. The device with high priority (smaller number) always gets the bus control right for data transfer operation.

Rotating priority algorithm: the arbitration scheme will not consider every device's priority and just according to a sequence to operate. For example, when device 1 finish its data transmission, device 2 request the data transfer, when arbitrator finds its buffer is not empty, the arbitrator will give device 2 the bus control right. Or else, device 2 will have no right to get bus control and arbitrator will check device 3 for data transmission operation.

Time quantum algorithm: the arbitrator will give each device the same time quantum to transfer their data. The time quantum is a constant that equals 16 clock cycles (defined inside the simulation function). If a device's operation time is smaller than the time slice (or time quantum), it will finish data transaction smoothly. Or else, the arbitration scheme will stop it, push its related parameters into a stack for protection when its operation does not finish in a fixed time slice and select next device to use bus for data transfer operation according to corresponding algorithm.

There are seven input widgets (in the GUI) collect parameters for the simulation process: BUS_SETTING_FORM; BUS0 (HIGH-LEVEL)_DEVICES_SETTING_FORM; BUS1_DEVICES_SETTING_FORM; BUS2_DEVICES_SETTING_FORM; BUS3_DEVICES_SETTING_FORM; BUS4_DEVICES_SETTING_FORM; BUS5_DEVICES_SETTING_FORM. Then these parameters will be send to the input file and then send to simulation function.

After getting relative system parameters, the simulation function executes a big loop that implements the whole data transfer transaction. This big loop consists of four parts:

Part 1: Initialization before starting data transfer transaction.

Part 2: If a new buffer full of data is due for a device, checking that current buffer is empty, insetting data, and deciding instant when next buffer is due.

Part 3: If there are one or more devices with data in their buffers, then select the device with highest priority to start data transaction operation according to the arbitration algorithm

Part 4: When the device with highest priority start to transfer data, we use a function defined inside the simulation function called switch() to implement the whole data transfer process; switch() includes IDLE operation to do overhead operation before transferring data, ADDRESS operation to transfer address, WAIT operation to execute wait state action during data transfer, and DATA operation to transfer data stream with general method and burst method

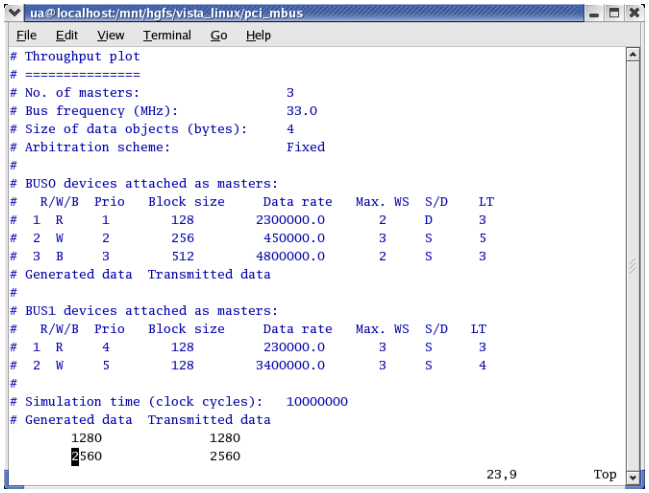
The PCI simulation function will also provide files about throughput, data rate, histogram and detailed information when required. Detailed information file includes:

(1) The time instants at which new buffers full of data are generated for the individual devices.

(2) The time instants at which a new master is selected on the PCI bus.

(3) The time instants at which data overruns occur. If more than 10 data overruns take place in a given simulation experiments, further messages about data overruns during the given experiment are suppressed, however, at the end of the experiment a message "*** *Data overrun messages suppressed*" is produced.

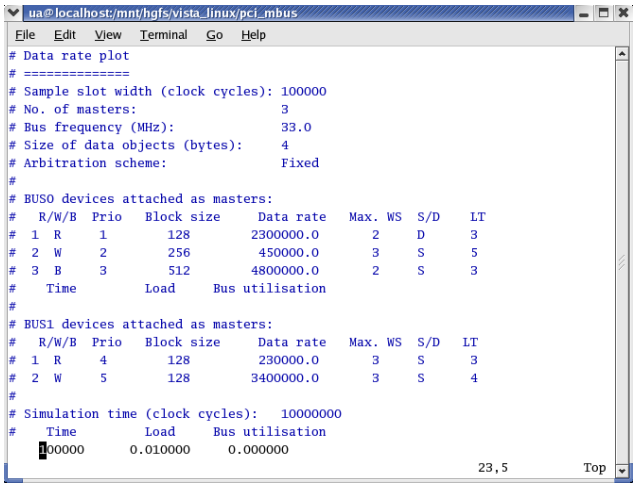
Throughput file includes: (Generated data, Transmitted data) pairs, suitable for plotting by the "gnuplot plot" command. Here is a figure about throughput file:



Throughput file

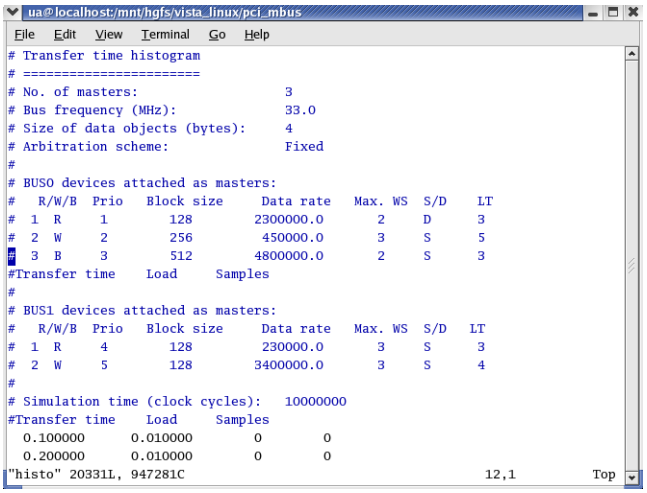
Data rate file includes: (Time, Load, Bus utilization) triplets, (t, l, u), suitable for plotting a 3D plot by the "gnuplot splot" command; Here is a figure about data rate

file:



Data rate file

Histogram file includes: (Transfer time, Load, Samples) triplets, (t,l,n), suitable for plotting a 3D plot by the "gnuplot splot" command; Here is a figure about histogram file:



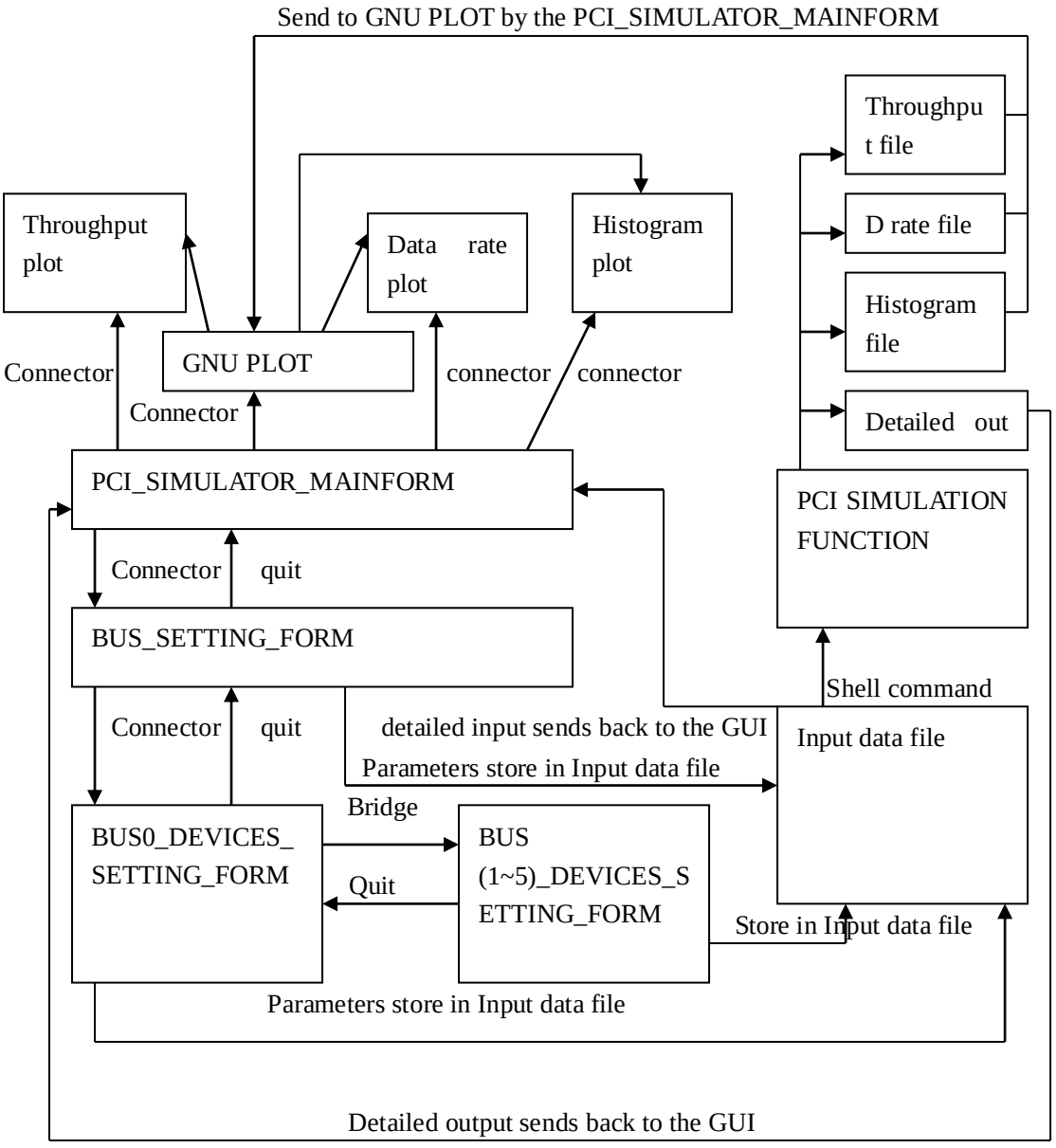
Histogram file

Throughput, data rate and histogram files will send to GNU PLOT to generate graphs when the user presses RUNPLOT Push Button in PCI_SIMULATOR_MAINFORM (in the GUI).

2. PCI Simulator GUI

This GUI is written in the environment of Qt designer 3 in red hat Linux 9 and it is a mixture of both C++ and Qt codes; it can be separated into three sections: input widgets; the output widget and an input file. Input widgets are the places to specify input parameters; the output widget controls the simulator and shows defined input

and output information; the input file generates the input data stream for the simulation function. Input and output section will be discussed separately, the description of the input file will be discussed inside the out put section; because the data formation of input file is defined in a slot function called "SIM" in the output widget. Here is a figure about the GUI inter-relationship map; the description of the GUI is based on this map.



GUI Connectivity Map

The GUI includes lots of .h and .cpp files and a folder for input & out files management; In this GUI, we provide eight visible widgets, each visible widget includes two classes; one provides the profile of the visible widget and some basic slot, and a heritage class to define specific functions and slots, the class name is also

the name of the visible widget. Detailed descriptions of these widgets are in the "input" section and the "output" section below; out put graphs are generated by GNU Plot and can be loaded by the GUI. From the view of the GUI, in fact, we do not care about how the simulation function works; we just care about the correct method to generate input files and the correct method to load output files and graphs. Here is a list and basic descriptions of files in the GUI.

PCI_SIMULATOR_MAINFORM (output)

(1) guimainform.ui.h: defined a class by the same name as the file name "guimainform" for the main form; the ui.h file generated automatically by the Qt designer and can be used to define custom slots;

(2) guimainform.ui: ui file for widget "guimainform"; the ui.file generated automatically by the Qt designer; ui file includes the profile and tool information about the widget; when needed ,it can be used to related generate .h and .cpp files.

(3) subguimain.h: defined the heritage class of "guimainform" called "subguimain" to implement custom slots and functions.

(4) subguimain.cpp: codes about functions in "subguimain".

BUS_SETTING_FORM (input)

(5) bus_settingform.ui.h: defined a class called "bus_settingform" for widget "bus_setting form".

(6) bus_settingform.ui: ui file for widget "bus_setting form".

(7) subbus_settingform.h: defined the heritage class of "bus_settingform" called "subbus_settingform" to implement custom slots and functions.

(8) subbus_settingform.cpp: codes about functions in "subbus_settingform".

BUS0 (HIGH-LEVEL)_DEVICES_SETTING_FORM (input)

(9) bus0_dev_settingform.ui.h: defined a class called "bus0_dev_settingform" for widget " bus0_dev_settingform".

(10) bus0_dev_settingform.ui.: ui file for widget "bus0_dev_settingform".

(11) subbus0_dev_settingform.h: defined the heritage class of "bus0_dev_settingform" called "subbus0_dev_settingform" to implement custom slots and functions.

(12) subbus0_dev_settingform.cpp: codes about functions in

"subbus0_dev_settingform".

BUS1_DEVICES_SETTING_FORM (input)

(13) bus1_dev_settingform.ui.h: defined a class called "bus1_dev_settingform" for widget "bus1_dev_settingform".

(14) bus1_dev_settingform.ui.: ui file for widget "bus1_dev_settingform".

(15) subbus1_dev_settingform.h: defined the heritage class of "bus1_dev_settingform" called "subbus1_dev_settingform" to implement custom slots and functions.

(16) subbus1_dev_settingform.cpp: codes about functions in "subbus1_dev_settingform".

BUS2_DEVICES_SETTING_FORM (input)

(17) bus2_dev_settingform.ui.h: defined a class called "bus2_dev_settingform" for widget " bus2_dev_settingform".

(18) bus2_dev_settingform.ui.: ui file for widget "bus2_dev_settingform".

(19) subbus2_dev_settingform.h: defined the heritage class of "bus2_dev_settingform" called "subbus2_dev_settingform" to implement custom slots and functions.

(20) subbus2_dev_settingform.cpp: codes about functions in "subbus2_dev_settingform".

BUS3_DEVICES_SETTING_FORM (input)

(21) bus3_dev_settingform.ui.h: defined a class "bus3_dev_settingform" for widget " bus3_dev_settingform".

(22) bus3_dev_settingform.ui.: ui file for widget "bus3_dev_settingform".

(23) subbus3_dev_settingform.h: defined the heritage class of "bus3_dev_settingform" called "subbus3_dev_settingform" to implement custom slots and functions.

(24) subbus3_dev_settingform.cpp: codes about functions in "subbus3_dev_settingform".

BUS4_DEVICES_SETTING_FORM (input)

(25) bus4_dev_settingform.ui.h: defined a class called "bus4_dev_settingform" for widget " bus4_dev_settingform".

(26) bus4_dev_settingform.ui.: ui file for widget “bus4_dev_settingform”.

(27) subbus4_dev_settingform.h: defined the heritage class of “bus4_dev_settingform” called “subbus4_dev_settingform” to implement custom slots and functions.

(28) subbus4_dev_settingform.cpp: codes about functions in “subbus4_dev_settingform”.

BUS5_DEVICES_SETTING_FORM (input)

(29) bus5_dev_settingform.ui.h: defined a class called “bus5_dev_settingform” for widget “ bus5_dev_settingform”.

(30) bus5_dev_settingform.ui.: ui file for widget “bus5_dev_settingform”.

(31) subbus5_dev_settingform.h: defined the heritage class of “bus5_dev_settingform” called “subbus5_dev_settingform” to implement custom slots and functions.

(32) subbus5_dev_settingform.cpp: codes about functions in “subbus5_dev_settingform”.

Input parameters definition and initialization

(33) init.h: defined all input parameters; input parameters were defined as “extern” variables to make them more flexible to be transferred among different widgets and classes.

(34) init.cpp: initialize all “extern” input parameters during the initialization process of the GUI.

Throughput plot interface (output)

(35) throughputview.h: defined a class called “throughputview” to load the throughput plot shows in a widget from a push button.

(36) throughputview.cpp: codes about functions in class “throughputview”.

Data rate plot interface (output)

(37) datarateview.h: defined a class called “datarateview” to load the data rate plot shows in a widget from a push button.

(38) datarateview.cpp: codes about functions in class “datarateview”.

Histogram plot interface (output)

(39) histview.h: defined a class called “histview” to load the histogram plot shows in a

widget from a push button.

(40) histview.cpp: codes about functions in class “histview”.

Other files

(41) pcisim.pro: project file, generated automatically by Qt designer;

(42) main.cpp: main process of the GUI, it also determined which widget is the main user form.

(43) plotmanagement folder: this folder should include 8 files; two .sh files(eg1.sh & eg.sh) to load the simulation function and the GNU Plot; three files(drate.gnp, tplot.gnp, histo.gnp) are GNU Plot codes for generating graphs; three files(fordata, forthrough, forhist) are managed by the GUI and response to user’s action to set graphs loading and saving information

INPUT

All input information will be collected by input widgets and sent to a specific input file with a pre-defined formation, and then sent to the simulation function. INPUT section includes seven widgets: BUS_SETTING_FORM; BUS0 (HIGH-LEVEL)_DEVICES_SETTING_FORM; BUS1_DEVICES_SETTING_FORM; BUS2_DEVICES_SETTING_FORM; BUS3_DEVICES_SETTING_FORM; BUS4_DEVICES_SETTING_FORM; BUS5_DEVICES_SETTING_FORM. These seven widgets define bus & files parameters and devices parameters on six PCI buses.

All input parameters are defined as “extern” variables in a .h file called “init.h”. It is not the best way to arrange these parameters, but in the GUI, it is enough for the data transmission between different widgets.

Related code (init.h):

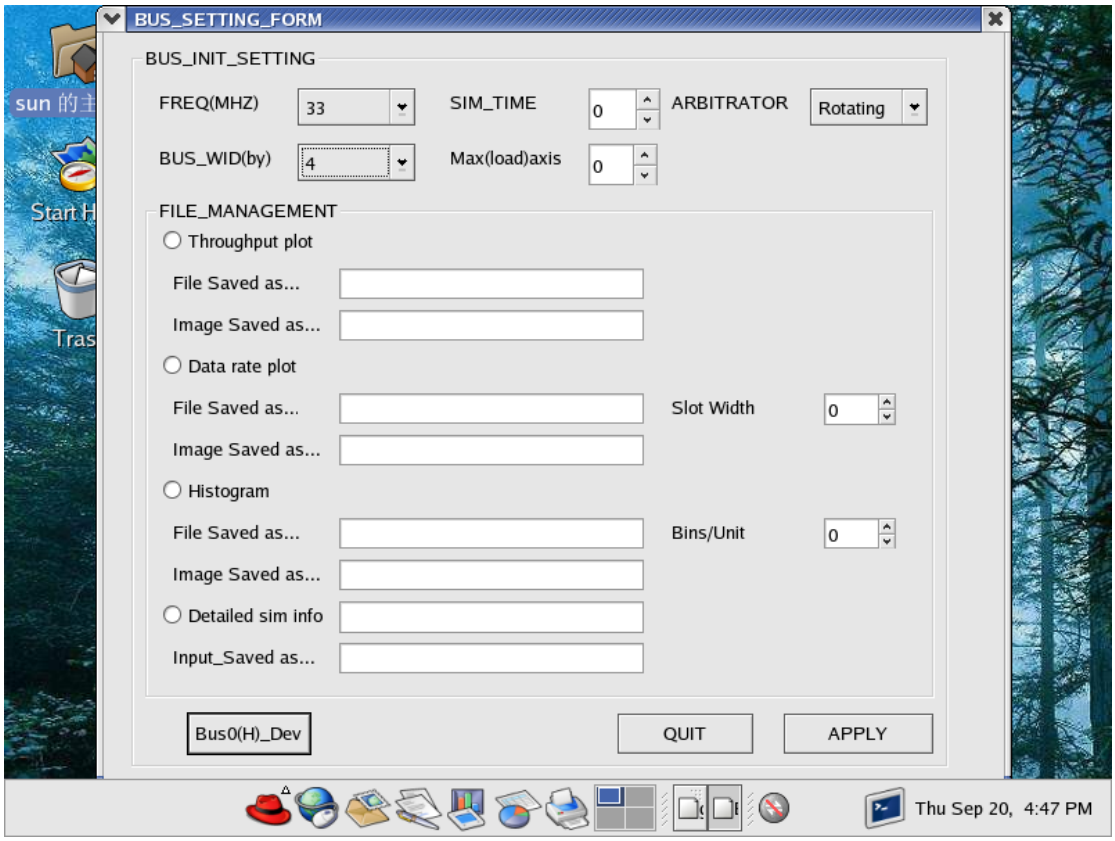
```
#include <qstring.h>
//bus0 devices attributes
extern int pr[6];//priority pr[1] means bus0dev1 pr[2] means bus0 dev2……
extern double dr[6];// data rate
extern int bs[6]; //buffer size
extern int mws[6]; //wait state
extern int LT[6]; //latency
extern QString rw[6]; //function :write, read or both
extern QString sd[6]; //wait state scheme
extern int PBAbus1[6]; //identifier about inter-connectivity
extern int nm;    //bus0 enabled devices numbers
extern int con[6];// which bus the specified device connected to
//bus1~5 parameters ignored.
//bus attributes
```

```
extern double f;// operation frequency
extern int buswidth;
extern long int simtot;// simulation time
extern QString arb;// arbitrator scheme
extern int ndr; //    No. of points on data rate axis
extern QString tqvalue;// for time qutaum; it is useless, just reserved for further
                                    //extension
//plot and file management
extern QString tnam; //throughput file name
extern QString rnam; //data rate file name
extern QString hnam; //historgam file name
extern QString detailinfo; //input file name
extern QString detailinfo2; //detailed output information
extern QString imagethrouplot; //throughput plot name
extern QString imagedrateplot; /data rate plot name
extern QString imagehistoplot; //histogram plot name
extern long int slotw;// Width of time slot for activity graph
extern int nh; // Bin index for normalized time 1.0
//bridge selection
extern int PB[6]; //bridge identifier for bus0 devices, if PB[i]!=0, the device is a
                        //bridge ( only available for bus0 devices)
extern int busnum; // Number of enabled buses
```

1. BUS_SETTING_FORM

This widget is implemented to initialize all bus (devices independent) parameters; simulation parameters, input file and output files management information about current simulation. All parameters specified will be sent to the input file. We have defined two classes for this widget: *bus_settingform* and its heritage class *subbus_settingform*; Class *bus_settingform* provides the profile of the visible widget and a basic slot to close that widget, while class *subbus_settingform* defined all functions and manages parameters that would be sent to the simulation function during the simulation.

Here is a figure about the profile of BUS_SETTING_FORM



BUS_SETTING_FORM

BUS_SETTING_FORM includes three segments:

1. Bus & current simulation parameters:

In this segment, bus (devices independent) and simulation parameters will be specified. These parameters will be sent to "extern" variables defined in "itih.h". Later, when the simulation starts, these parameters will also be written to the input file; Bus parameters include:

(1) BUS Frequency (MHz): Bus clock frequency (33, 64, or 133 MHz)

(2) BUS Width (Bytes): Size of data objects (4 or 8 bytes)

(3) BUS Arbitrator scheme: arbitrator selection (Rotating, Fixed or Time quantum); If Fixed, bus arbitration works according to fixed priority; If Rotating, bus arbitration works with a rotating (round robin) arbitration scheme; If Time quantum, bus arbitration works with time quantum or time slot scheme

Simulation parameters include:

(1) Total Simulation Time: number of slots for current simulation

(2) Number of points on current simulation result graph data rate axis: A positive integer giving the number, n_r, of simulation experiments at increasing applied loads which will be performed. In experiment, each device will generate data at a

fraction i/n_r of its maximum data rate

2. Input & Out put files parameters:

In this segment, input & out put files and graphs management are implemented. Users can choose whether they need output files and images and detailed information about the simulation or not, paths of related files and graphs have to be typed into the specific line editors when needed; and parameters for generating graphs are also available. Input file must be valid because it is the path where the simulation function will load the input file to start simulation; the input file is also useful for the analyzing after the simulation; these parameters will be sent to "extern" variables defined in "itih.h". Later, when the simulation starts, these parameters will also be written to the input file; Input & output files parameters include:

(1) Throughput file saving path and name (ignored when the throughput plot radio button is not clicked)

(2) Throughput plot saving path and name (ignored when the throughput plot radio button is not clicked)

(3) Data rate file saving path and name (ignored when the Data rate plot radio button is not clicked)

(4) Data rate plot saving path and name (ignored when the Data rate plot radio button is not clicked)

(5) Width of time slot for activity graph (ignored when the Data rate plot radio button is not clicked)

(6) Histogram file saving path and name (ignored when the Histogram plot radio button is not clicked)

(7) Histogram plot saving path and name (ignored when the Histogram plot radio button is not clicked)

(8) Bin index for normalized time 1.0 (ignored when the Histogram plot radio button is not clicked)

(9) Detailed output information file saving path and name (ignored when the detailed information radio button is not clicked)

(10) Input file saving path and name (a valid path must be typed in during each simulation!)

3. Slot buttons:

In this segment, we provide: links to both bus0 devices setting form; slot to save current bus (devices independent) parameters and slot to close the bus setting widget.

(1) **BUS0 (H)_DEV push button** connects to BUS0 (HIGH-LEVEL)_DEVICES_ SETTING FORM.
Related code:

```
void subbus_settingform::bus0devset()
{
//connect to dev0-setting forms
subbus0_dev_settingform *V=new subbus0_dev_settingform();
 V->show();}
```

(2) **QUIT push button** connects to the close () function provided by Qt. When pressed, BUS_SETTING_FORM will be closed without saving

(3) **APPLY push button** connects to the slot to save all bus (devices independent) parameters and files management information; before inserting these parameters, all existing values will be erased first.
Relevant code:

Clear all file setting that have done in the previous simulation

```
int subbus_settingform::initvalue()
{
 tnam="";
 rnam="";
 hnam="";
 detailinfo="";
 detailinfo2="";
 imagethrouplot="";
 imagedrateplot="";
 imagehistoplot="";
 slotw=0;
 nh=0;
 tqvalue="";
  }
```

Parameters setting

```
int subbus_settingform::businit()
{
      initvalue();
      //bus attributes
      if (feq_comboBox->currentText()=="33"){f=33;}
      if (feq_comboBox->currentText()=="66"){f=66;}
      if (feq_comboBox->currentText()=="133"){f=133;}
      if(buswidth_comboBox->currentText()=="4"){ buswidth=4;}
      else { buswidth=8;}
```

```
simtot=sim_time_spinBox->value();
ndr=max_axis_spinBox->value();
tqvalue=arbitrate_comboBox->currentText();
if( arbitrate_comboBox->currentText()=="Fixed"){arb="f";}
if( arbitrate_comboBox->currentText()=="Rotating"){arb="r";}
if( arbitrate_comboBox->currentText()=="Time Quatum"){arb="t";}
//plot and file management
if(Throughput_plot_radioButton->isOn()){
tnam=throughput_file_path_lineEdit->displayText();
imagethrouplot=throughput_image_path_lineEdit->displayText();
}
if(Data_rate_plot_radioButton->isOn()){
rnam=datarate_file_path_lineEdit->displayText();
imagedrateplot=datarate_image_path_lineEdit->displayText();
        slotw=drate_slotwidth_spinBox->value();
}
if(Histogram_plot_radioButton->isOn()){
hnam=histogram_file_path_lineEdit->displayText();
imagehistoplot=histogram_image_path_lineEdit->displayText();
        nh=hist_bin_peru_spinBox->value();
  }
```

Input file:

```
detailinfo=detail_info_lineEdit->displayText();
```

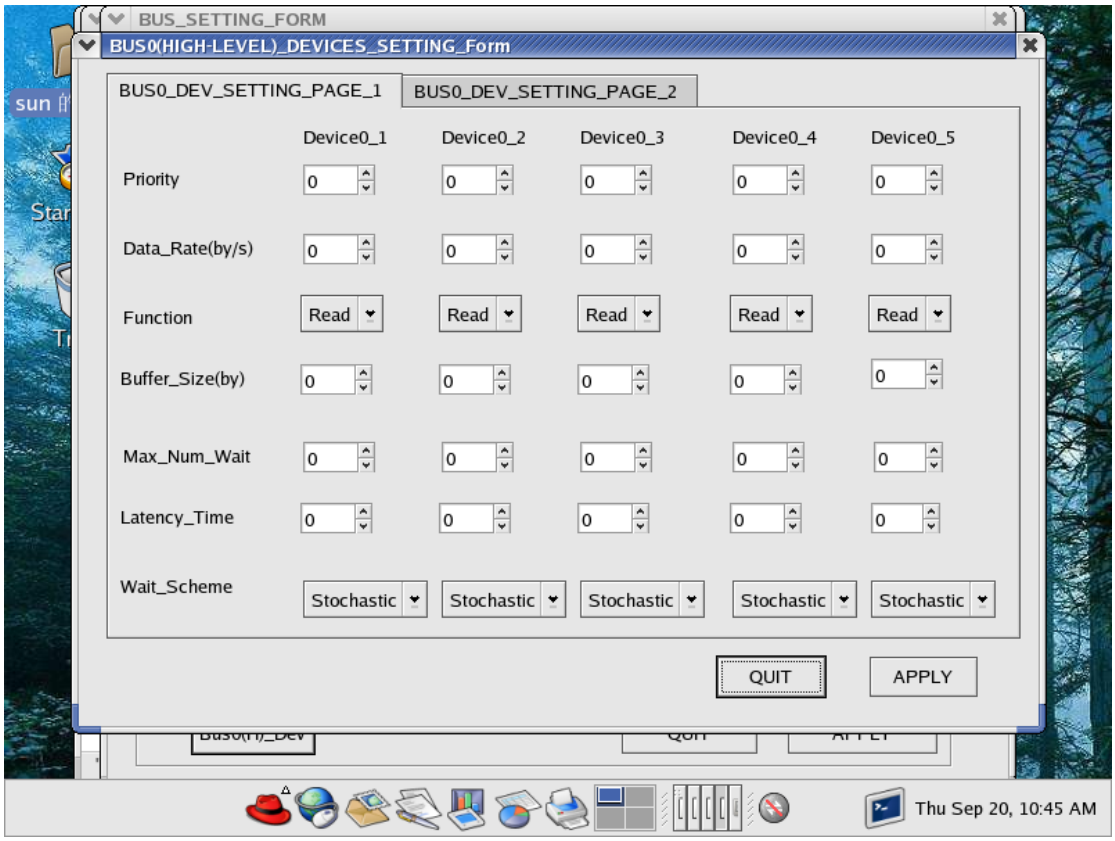
Detailed output file, if available

```
if(Detailed_sim_info_radioButton->isOn()){

detailinfo2=siminfo_lineEdit->displayText();
}
}
```

2. BUS0 (HIGH-LEVEL)_DEVICES_SETTING_FORM

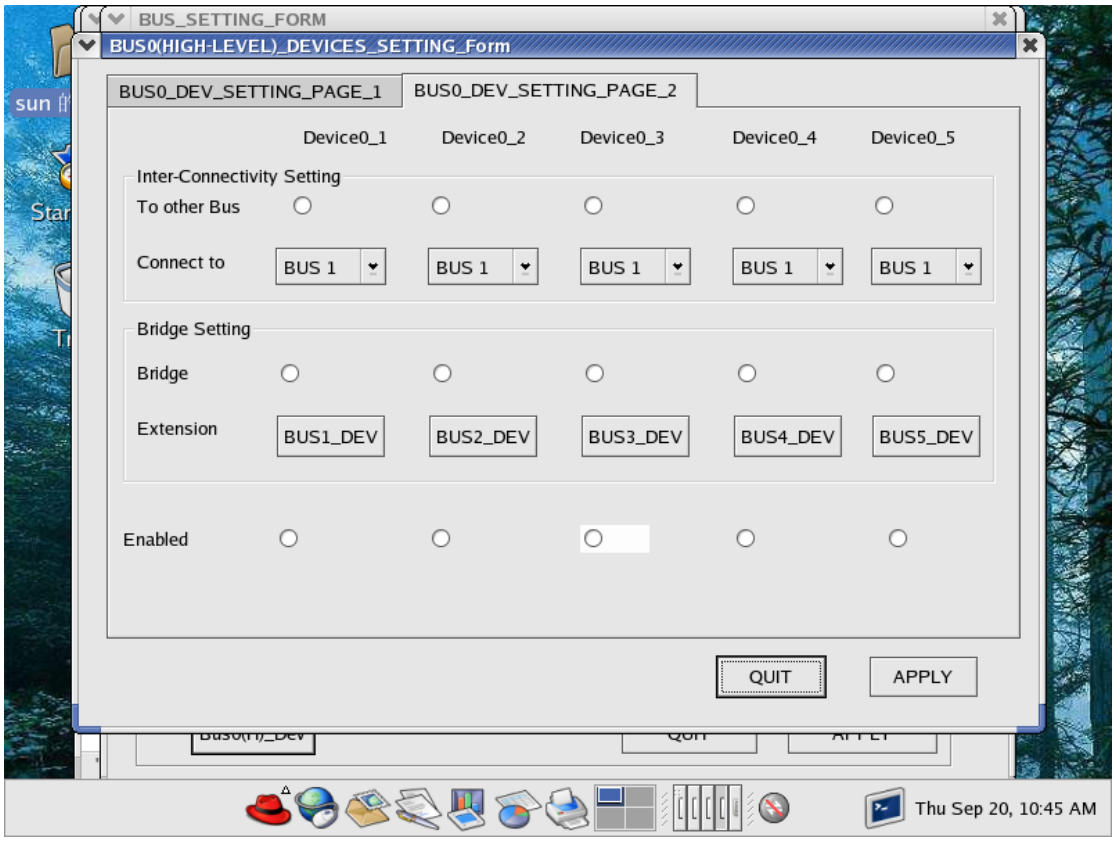
This widget is implemented to initialize all bus0 devices parameters and send these parameters to the simulation function. There are five available devices on BUS0. This widget includes two pages; We have defined two classes for this widget: *bus0_dev_settingform* and its heritage class *subbus0_dev_settingform*; Class *bus0_dev_settingform* provides the profile of the visible widget and a basic slot to close that widget, while class *subbus0_dev_settingform* defines all functions and manages bus0 devices parameters that would be sent to the simulation function during the simulation.

Here are two figures about the profile of BUS0 _DEVICES_SETTING_FORM



BUS0 _DEVICES_SETTING_FORM (1)

This widget includes lots of parameters to be specified; one page is not enough, I implemented a “tabwidget” tool in Qt designer to arrange these parameters into two pages in one widget; the same approach is also implemented in other bus devices setting forms.



BUS0 _DEVICES_SETTING_FORM (2)

BUS0 (HIGH-LEVEL)_DEVICES_SETTING_FORM includes two segments:

1. **Bus0 devices parameters:**

In this segment, bus0 parameters will be specified. We select bus0 as the high-level PCI bus; bridge management is also included in this widget. Bus0 devices parameters in this simulator include:

(1) Priority: 0-5; set priorities for devices on bus0, bus0 device priorities value will be ignored when choosing rotating and time quantum priority arbitration (ignored when related bus0 device enabled radio button is not clicked)

(2) Transaction type: Read, Write and R/W; Read for a bus0 device which reads from target, write for a device which writes data to target and R/W for device that supports both read and write function (ignored when related bus0 device enabled radio button is not clicked)

(3) Buffer size: 0-1024bytes; set the size in bytes of the buffer, which the bus0 device as master will attempt to fill or empty during each transaction. If there is no competing devices are active during the transaction, the amount of data will be transferred in a single burst. If pre-emption occurs before the end of the transaction, the rest of the buffer will be transferred in on or more following secondary transactions (ignored when related bus0 device enabled radio button is not clicked)

(4) Maximum data rate: 0-65536bytes/s; define maximum data rate for each bus0 device (ignored when related bus0 device enabled radio button is not clicked)

(5) Maximum wait states: 0-8 time slot; the maximum number of wait states, which can potentially be inserted in each data phase by the device's target in the system under consideration (ignored when related bus0 device enabled radio button is not clicked)

(6) Wait state generator process: Stochastic, Deterministic; if Stochastic, bus0 device wait state value is selected randomly from 0 to the maximum wait state for each data phase; if Deterministic, bus0 device wait state value is the maximum wait state for each data phase (ignored when related bus0 device enabled radio button is not clicked)

(7) Latency timer: 0-20 time slots; for the latency timer at the start of each new transaction involving the bus0 device (ignored when related bus0 device enabled radio button is not clicked)

Inter-connectivity setting:

(8) To other Bus: defined as a radio button; when clicked, related bus0 device will have the ability to connect to another bus (ignored when related bus0 device enabled radio button is not clicked)

(9) Connect to: it determines the bus number the selected device will be connected to (ignored when related bus0 device enabled radio button is not clicked, ignored when related bus0 device "to other bus" radio button is not clicked)

Bridge setting:

(10) PCI Bridge: defined as a radio button; when clicked, related bus0 device will be the P2PB. (Ignored when related bus0 device enabled radio button is not clicked)

(11) Extension: defined as a push button; it provides the connector to a low-level bus; in our simulator, the bridge setting follows such rules: bus0_dev1 connect to bus1; bus0_dev2 connect to bus2; bus0_dev3 connect to bus3; bus0_dev4 connect to bus4; bus0_dev5 connect to bus5, which have been mentioned in the Multiple PCI Buses Topology (ignored when related bus0 device enabled radio button is not clicked, ignored when related bus0 device "PCI Bridge" radio button is not clicked)

Related code (to bus1):

```
void subbus0_dev_settingform::bus1devset()
{  if((bus0_dev1_enabled_radioButton->isOn())&(bus0_dev1_bridge_radio
     Button->isOn())){
 //connect to bus1-setting forms
 subbus1_dev_settingform *V=new subbus1_dev_settingform();
```

```
V->show();}
 }
 //bus2~5 ignored…
```

(12) A variable to count number of devices available on Bus0 (nm); (number of clicked bus0 device enabled buttons)

2. Slot buttons:

In this segment, we provide slots to save current bus0 devices parameters and to close the bus0 devices setting widget.

(1) **QUIT push button** connects to the close () function provided by Qt. When pressed, BUS0 (HIGH-LEVEL)_DEVICES_SETTING_FORM will be closed without saving

(2) **APPLY push button** connects to the slot to save all bus0 devices parameters; before inserting these parameter, all existing values will be erased first.

Related code:

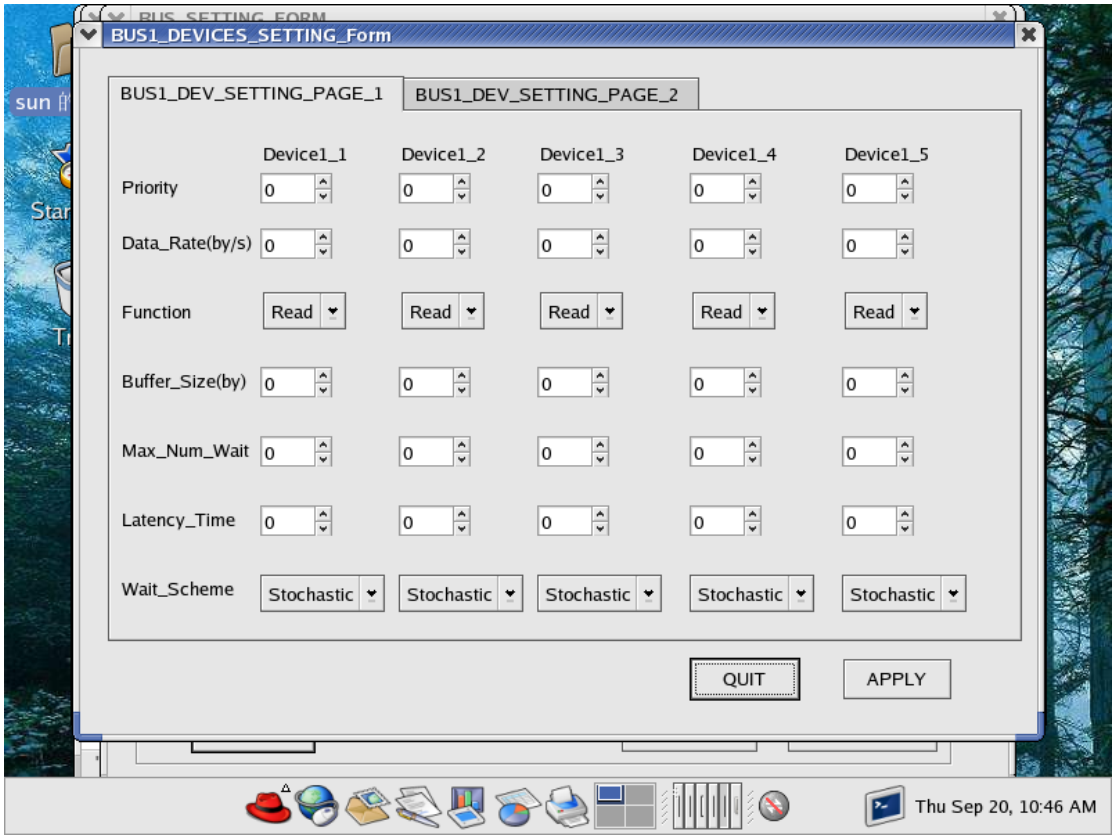
```
int subbus0_dev_settingform::initvaluebus0()
  {
    for (int t=1;t<6;t++)
 {
 pr[t]=0;
 dr[t]=0;
 bs[t]=0;
 mws[t]=0;
 LT[t]=0;
 rw[t]="";
 sd[t]="";
 PBAbus1[t]=0;
 PB[t]=0;
 con[t]=-1;
}
 nm=0;
 busnum=1;
}
int subbus0_dev_settingform::bus0devinit()
{
    int i=1;
     initvaluebus0();
     //bus0 devices 1
  if(bus0_dev1_enabled_radioButton->isOn()){
     nm++;
     pr[i]=bus0_dev1_prio_spinBox->value();
```

```
        dr[i]=bus0_dev1_datarate_spinBox->value();
        bs[i]=bus0_dev1_buffsize_spinBox->value();
        mws[i]=bus0_dev1_maxwait_spinBox->value();
        LT[i]=bus0_dev1_latencytime_spinBox->value();
        if (bus0_dev1_func_comboBox->currentText()=="Read"){rw[i]="r";}
        if (bus0_dev1_func_comboBox->currentText()=="Write"){rw[i]="w";}
        if (bus0_dev1_func_comboBox->currentText()=="R/W"){rw[i]="b";}
        if
(bus0_dev1_waitscheme_comboBox->currentText()=="Stochastic"){sd[i]="s";}
        if
(bus0_dev1_waitscheme_comboBox->currentText()=="Deterministic"){sd[i]="d";}
        if(bus0_dev1_tobus1_radioButton->isOn() )
        {PBAbus1[i]=i;
       if(bus0_dev1_connect_comboBox->currentText()=="BUS 1"){con[i]=1;}
                    if(bus0_dev1_connect_comboBox->currentText()=="BUS
2"){con[i]=2;}
                    if(bus0_dev1_connect_comboBox->currentText()=="BUS
3"){con[i]=3;}
                    if(bus0_dev1_connect_comboBox->currentText()=="BUS
4"){con[i]=4;}
       if(bus0_dev1_connect_comboBox->currentText()=="BUS 5"){con[i]=5;}
          }

if(bus0_dev1_bridge_radioButton->isOn()){PB[i]=1;PBAbus1[i]=0;con[i]=-1;busnu
m++;}

     i++;
}
//bus0_dev2~bus0_dev5 ignored…
}
```

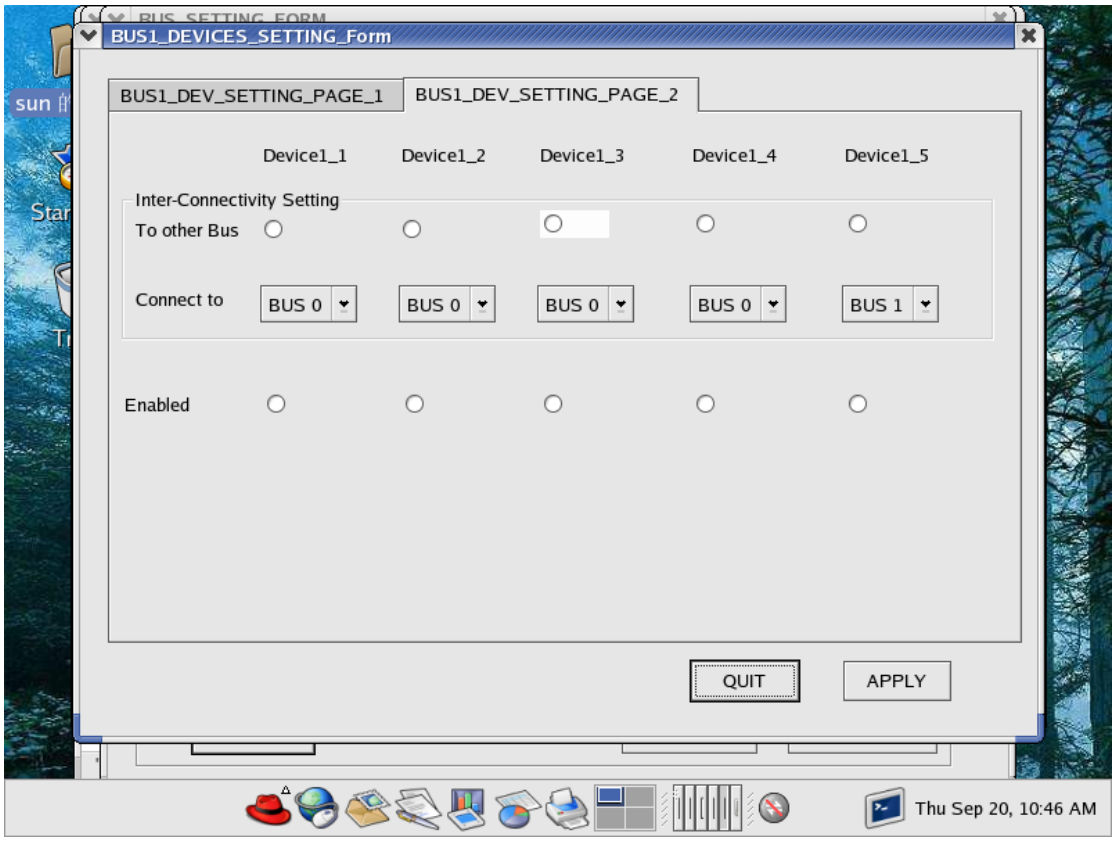
3. BUS1_DEVICES_SETTING_FORM

This widget is implemented to initialize all bus1 devices parameters and send them to the simulator function. There are five available devices on BUS1. This widget includes two pages; We have defined two classes for this widget: *bus1_dev_settingform* and its heritage class *subbus1_dev_settingform*; Class *bu1_dev_settingform* provides the profile of the visible interface and a basic slot to close that widget, while class *subbus1_dev_settingform* defines all functions and manages bus1 devices parameters that would be sent to the simulation function in the simulation.

Here are two figures about the profile of BUS1_DEVICES_SETTING_FORM



BUS1_DEVICES_SETTING_FORM (1)



BUS1_DEVICES_SETTING_FORM (2)

BUS1_DEVICES_SETTING_FORM includes two segments:

1. **Bus1 devices parameters**:

In this segment, bus1 parameters will be specified. These parameters share the same format that was implemented in bus0. Compared to bus0, bus1 does not need to provide bridge function; when we simulate a single PCI bus, this widget is useless. Bus1 devices parameters in this simulator include:

(1) Priority:
(2) Transaction type
(3) Buffer size
(4) Maximum data rate
(5) Maximum wait states
(6) Wait state generator process
(7) Inter-connectivity setting:
(8) To other Bus
(9) Connect to
(10) A variable to count number of devices available on Bus1 (nm1); (number of clicked bus1 device enabled buttons)

2. Slot buttons:

In this segment, we provide slots to save current bus1 devices parameters and to close the bus1 devices setting widget.

(1) **QUIT push button** connects to the close () function provided by Qt. When pressed, BUS1_DEVICES_SETTING_FORM will be closed without saving

(2) **APPLY push button** connects to the slot to save all bus1 devices parameters; before inserting these parameters, all existed values will be erased first.

Related code:

```
int subbus1_dev_settingform::initvaluebus1()
{
  for (int t=1;t<6;t++)
 {
 pr1[t]=0;
 dr1[t]=0;
 bs1[t]=0;
 ws1[t]=0;
 LT1[t]=0;
 rw1[t]="";
 sd1[t]="";
 PBAbus0[t]=0;
 con1[t]=-1;
 }
 nm1=0;
```

```
}
int subbus1_dev_settingform::bus1devinit()
{
     int i=1;
       initvaluebus1();
       //bus1 devices 1
   if(bus1_dev1_enabled_radioButton->isOn()){
       nm1++;
       pr1[i]=bus1_dev1_prio_spinBox->value();
       dr1[i]=bus1_dev1_datarate_spinBox->value();
       bs1[i]=bus1_dev1_buffsize_spinBox->value();
       ws1[i]=bus1_dev1_maxwait_spinBox->value();
       LT1[i]=bus1_dev1_latencytime_spinBox->value();
       if (bus1_dev1_func_comboBox->currentText()=="Read"){rw1[i]="r";}
       if (bus1_dev1_func_comboBox->currentText()=="Write"){rw1[i]="w";}
       if (bus1_dev1_func_comboBox->currentText()=="R/W"){rw1[i]="b";}
       if
(bus1_dev1_waitscheme_comboBox->currentText()=="Stochastic"){sd1[i]="s";}
       if
(bus1_dev1_waitscheme_comboBox->currentText()=="Deterministic"){sd1[i]="d";}
       if(bus1_dev1_tobus0_radioButton->isOn() ){PBAbus0[i]=i;
                    if(bus1_dev1_connect_comboBox->currentText()=="BUS
0"){con1[i]=0;}
                    if(bus1_dev1_connect_comboBox->currentText()=="BUS
2"){con1[i]=2;}
                    if(bus1_dev1_connect_comboBox->currentText()=="BUS
3"){con1[i]=3;}
                    if(bus1_dev1_connect_comboBox->currentText()=="BUS
4"){con1[i]=4;}
      if(bus1_dev1_connect_comboBox->currentText()=="BUS 5"){con1[i]=5;}       }
       i++;
}
//bus1_dev2~bus1_dev5 ignored…
}
```

4. BUS (2~5)_DEVICES_SETTING_FORM

In our consideration, bus1~bus5 share the same format; there are five devices for each bus, and devices in these five buses have same parameters; widgets created for these five buses share almost the same code. After discussing the “BUS 1_DEVICES_SETTING_FORM”, we just leave detailed code in my appendix for any one who is interested in these four buses.

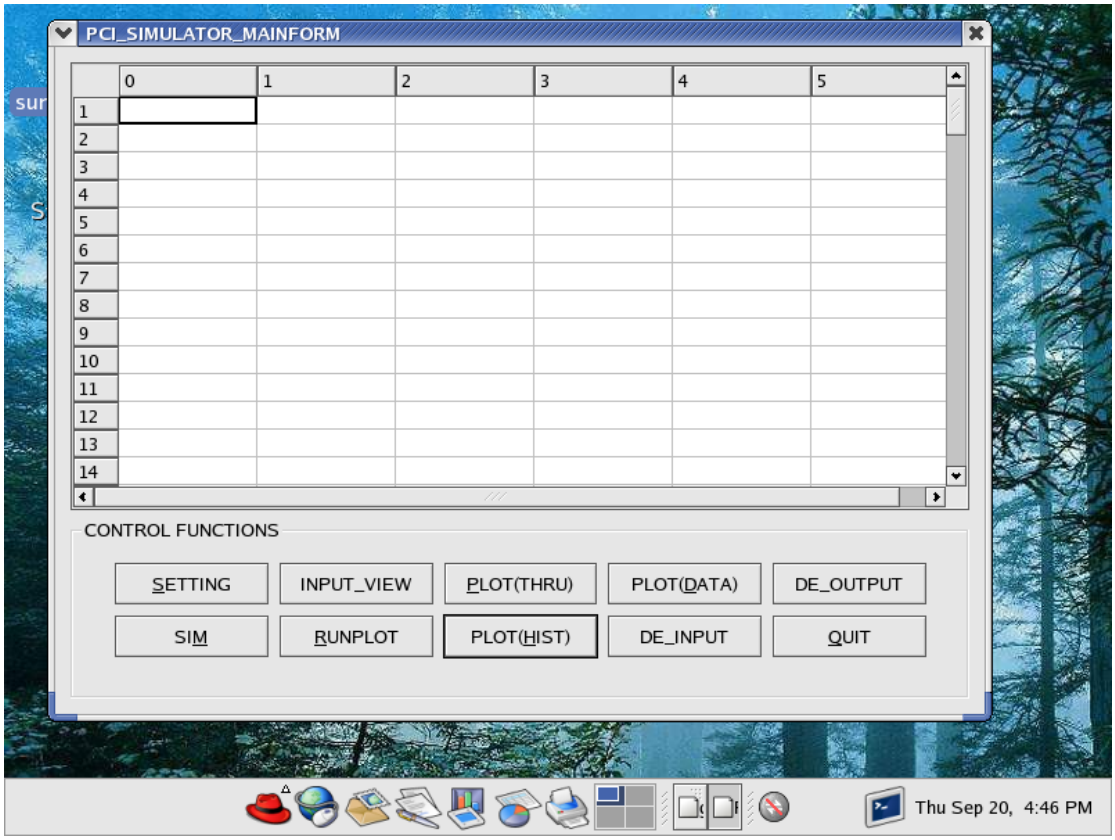
OUTPUT

In this section, we provide the interface to load the simulation function and output management; Output section includes three parts: PCI_SIMULATOR_MAINFORM widget, three classes (throughputview, class datarateview and class histview) to load graphs and some GNU PLOT management files to draw graphs.

1. PCI_SIMULATOR_MAINFORM

This is the main widget of the PCI simulator. We have defined two classes for this widget: *guimainform* and its heritage class *subguimain*; Class *guimainform* provides the profile of the visible widget and also defines some connection slots, while class *subguimain* defines slots to exhibit basic input & output and slots to write input file and to implement simulation function.

Here is a figure about the profile of PCI_SIMULATOR_MAINFORM.



PCI_SIMULATOR_MAINFORM

PCI_SIMULATOR_MAINFORM includes two segments:

1. Key PCI buses and devices parameters：

In this segment, we would like to exhibit the basic input of current simulation in a form. Then the user can have a clear sense of current simulation setting; the first column in the form contains names of the parameters; the form also provides line

number and column number to make it flexible to locate any parameter; these parameters has been discussed in the input section, Exhibited information includes:
(1) Bus frequency;
(2) Bus width;
(3) Simulation time (length of current simulation);
(4) Arbitrator scheme;
(5) Devices setting (bus0~bus5 if enabled) which includes:
    (1) Buffer size
    (2) Data rate
    (3) Priority
    (4) Function
    (5) Maximum wait slot
    (6) Wait scheme
    (7) Latency
    (8) Bridge enabled (bus0 only)
    (9) Connectivity to other buses
    (10) Number of devices enabled on each bus
(6) Total number of enabled buses

Code about this segment is defined inside a slot function called "inputview()", detailed code will be introduced in the coming segment.

2. Simulator control slot buttons：

In this segment, we provide connectors that connect to relevant widgets or functions; detail input & out information about current simulation and generated graphs can be loaded from corresponding buttons, also we provide a interface that connects to bus setting form to initiate the simulation parameters setting. There are 10 slot buttons in all:

(1) **SETTING Push Button** connects to BUS_SETTING_FORM.
Related code:

```
void guimainform::viewbusetting()
  {
 subbus_settingform *V=new subbus_settingform();
 V->show();
  }
```

(2) **SIM Push Button** connects to the sim () slot that is used to implement the simulation. sim () slot can be separated into three steps: first, send all input parameters to the input file with the specific formation; then edit a .sh file to let the simulation function load this input file; finally, a process to execute this .sh file. When pressed, the simulation starts with the specified parameters and an xterm widget will show up to exhibit the detailed information about current simulation, when the simulation completes, the xterm will be closed

automatically. Each time when the simulator operates, the input file will be saved for further analyzing and data comparison.

Related code:

```
   void subguimain::sim()
{
```

Send all input parameters to the input file with the specific formation

```
int i;int j; int k; int l; int m; int n;
QFile file(detailinfo);
        if (file.open(IO_WriteOnly))
{
            QTextStream t(&file);
//sim parameters
            t<<QString::number(nm)<<"\n";
t<<QString::number(simtot)<<"\n";
t<<QString::number(ndr)<<"\n";
//throughput
if (tnam!="")
{   t<<"y"<<"\n";
    t<<tnam<<"\n";}
else {t<<"n"<<"\n";}
//data rate
if (rnam!="")
{   t<<"y"<<"\n";
    t<<rnam<<"\n";
    t<<QString::number(slotw)<<"\n";
}
else {t<<"n"<<"\n";}
//detailed info in shell
if (detailinfo2!="")
{   t<<"y"<<"\n";}
else {t<<"n"<<"\n";}
//histogram
if (hnam!="")
{   t<<"y"<<"\n";
    t<<hnam<<"\n";
    t<<QString::number(nh)<<"\n";
}
else{t<<"n"<<"\n";}
//sim parameters
t<<QString::number(f)<<"\n";
t<<QString::number(buswidth)<<"\n";
t<<arb<<"\n";
            //bus0 dev parameters
if (nm!=0){
```

```
    for( i=1;i<=nm;i++){
            t<<rw[i]<<"\n";
            t<<QString::number(pr[i])<<"\n";
            t<<QString::number(bs[i])<<"\n";
            t<<QString::number(dr[i])<<"\n";
            t<<QString::number(mws[i])<<"\n";
            t<<sd[i]<<"\n";
            t<<QString::number(LT[i])<<"\n";
            if(PB[i]!=0)
            { t<<"y"<<"\n";}
            else{ t<<"n"<<"\n";}
            if(PBAbus1[i]!=0)
            { t<<"y"<<"\n";
           t<<QString::number(con[i])<<"\n";}
            else{ t<<"n"<<"\n";}}}
    //bus1 dev parameters
  if(nm1!=0){
    t<<QString::number(nm1)<<"\n";
    for(j=1;j<=nm1;j++){
            t<<rw1[j]<<"\n";
            t<<QString::number(pr1[j])<<"\n";
            t<<QString::number(bs1[j])<<"\n";
            t<<QString::number(dr1[j])<<"\n";
            t<<QString::number(ws1[j])<<"\n";
            t<<sd1[j]<<"\n";
            t<<QString::number(LT1[j])<<"\n";
            if(PBAbus0[j]!=0)
            { t<<"y"<<"\n";
           t<<QString::number(con1[j])<<"\n";}
            else{t<<"n"<<"\n";}}}
  //bus2-5 dev parameters ignored…
}
      file.close();
```

The .sh file is implemented to let the simulation function load the input file and exhibit detailed information about current simulation in a xterm; here, “pcibridge is the name of the executable simulation function.

```
    QFile file1("/home/sun/project/plotmanagement/eg1.sh");
    if (file1.open(IO_WriteOnly))
{           Qstring s2="\""
            QTextStream t(&file1);
  t<<"xterm -e "<<s2<<"/home/sun/ pcibridge"<<" <"<<detailinfo<<s2<<"\n";
  }
     file1.close();
```

A process to load the .sh file

```
QProcess *proc;
proc =new QProcess(this);
proc->addArgument("/home/sun/project/plotmanagement/eg1.sh");
proc->start();
}
```

(3) **RUNPLOT Push Button** connects to GNU PLOT to generate graphs with relevant data files. This slot includes two steps: first, write loading and saving information into specific files, then a process to execute GUN PLOT. This function needs to be re-compiled before operating the simulator on other computers, because files loading paths would be different from the original edition.

Related code:

```
void guimainform::generateplots()
{
QProcess *proc;
QFile file("/home/sun/project/plotmanagement/fordrate");
QFile file1("/home/sun/project/plotmanagement/forhist");
QFile file2("/home/sun/project/plotmanagement/forthrough");
```

Put “splot ‘data rate file saving path’.” and “set output ‘data rate plot saving path’” commands in the file “fordrate” for the generation of data rate graph;

```
if (file.open(IO_WriteOnly))
{   QString s1="splot ";
    QString s2="\"";
 //file saved address
    QString s3=rnam;
    QString s4="set output ";
 //plot image saved address
    QString s5=imagedrateplot;
    QTextStream t(&file);
    t<<s4<<s2<<s5<<s2<<"\n"<<s1<<s2<<s3<<s2<<"\n";}
    file.close();
```

Put “splot ‘hist file saving path’.” and “set output ‘hist plot saving path’” commands in the file “forhist” for the generation of histogram graph;

```
if (file1.open(IO_WriteOnly))
{   QString s1="splot ";
    QString s2="\"";
 //file saved address
QString s3=hnam;
QString s4="set output ";
//plot image saved address
QString s5=imagehistoplot;
QTextStream t(&file1);
t<<s4<<s2<<s5<<s2<<"\n"<<s1<<s2<<s3<<s2<<"\n";}
```

```
file1.close();
```

Put “plot ‘Throughput file saving path’.” and “set output ‘Throughput plot saving path’” commands in the file “forthrough” for the generation of throughput graph;

```
if (file2.open(IO_WriteOnly))
{   QString s1="plot ";
    QString s2="\"";
    //file saved address
    QString s3=tnam;
    QString s4="set output ";
    //plot image saved address
    QString s5=imagethrouplot;
    QTextStream t(&file2);
    t<<s4<<s2<<s5<<s2<<"\n"<<s1<<s2<<s3<<s2<<"\n";}
    file2.close();
```

Here is the process to load the .sh file to run GNU PLOT

```
    proc =new QProcess(this);
    proc->addArgument("./project/plotmanagement/eg.sh");
    proc->start();
}
```

Codes inside the "./project/plotmanagement/eg.sh" are:

```
gnuplot "/home/sun/project/plotmanagement/drate.gnp"
gnuplot "/home/sun/project/plotmanagement/histo.gnp"
gnuplot "/home/sun/project/plotmanagement/tplot.gnp"
```

These three .gnp files are GNU Plot codes; .gnp files will load related “forthrough”, “fordrate” or “forhist”; then, GNU Plot will load these three .gnp files to generate graphs.

(4) **PLOT (THRU) Push Button** connects to the class used for displaying throughput (2D) graph in a widget.

Related code:

```
void guimainform::viewthroughputplot()
{
 throughputview *V=new throughputview();
 V->show();
  }
```

(5) **PLOT (DATA) Push Button** connects to the class used for displaying data rate (3D) graph in a widget.

Related code:

```
void guimainform::viewdatarateplot()
{
  datarateview *V=new datarateview();
  V->show();
}
```

(6) **PLOT (HIST) Push Button** connects to the class used for displaying histogram (3D) graph in a widget.
Related code:

```
void guimainform::viewhistplot()
  {
    histview *V=new histview();
    V->show();
    }
```

(7) **DE_OUTPUT Push Button** would open the detailed output information about current simulation in a "gedit" window. The detailed output information is generated during the simulation process.
Related code:

```
void guimainform::viewdetailinfoout()
{    QProcess *proc;
     proc =new QProcess(this);
     proc->addArgument("gedit");
     proc->addArgument(detailinfo2);
     proc->start();}
```

(8) **DE_INPUT Push Button** would open the input file about current simulation in a "gedit" window. From my suggestion, the input file shows the structure of the simulation inputs, if you can really understand it, it is possible to set parameters without the GUI, just type in them in a text editor. Then type the command in a terminal like "simulator path < input file path".
Related code:

```
void guimainform::viewdetailinfo()
{
  QProcess *proc;
  proc =new QProcess(this);
  proc->addArgument("gedit");
  proc->addArgument(detailinfo);
  proc->start();}
```

(9) **INPUT_VIEW Push Button** would send key input parameters to the form in this widget. When the simulator works, we can have a better understanding of current simulation by checking this form; actually, the input file includes all information contained in this form, but input file is much more difficult to read, the form is much clearer;
Related code:

```
void subguimain::inputview()
{ for (int i=0;i<6;i++)
      {              //bus parameters
     bus_event_table->setText(0,0,"BUS_FEQ");
```

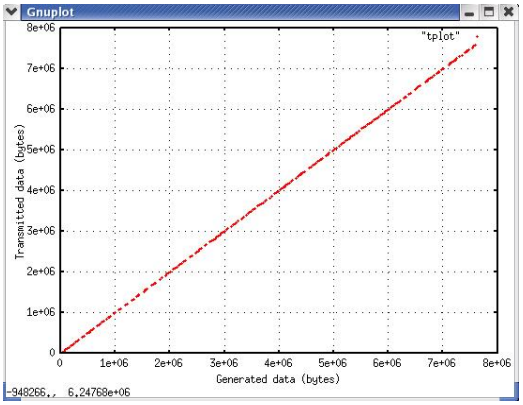
```
bus_event_table->setText(0,1,QString::number(f));
bus_event_table->setText(1,0,"BUS_WID");
bus_event_table->setText(1,1,QString::number(buswidth));
bus_event_table->setText(2,0,"SIM_TIM");
bus_event_table->setText(2,1,QString::number(simtot));
bus_event_table->setText(3,0,"ARBITRATOR");
bus_event_table->setText(3,1,arb);
//bus0 dev parameters
bus_event_table->setText(4,0,"BUS0_DEV");
bus_event_table->setText(4,1,"DEV01");
bus_event_table->setText(4,2,"DEV02");
bus_event_table->setText(4,3,"DEV03");
bus_event_table->setText(4,4,"DEV04");
bus_event_table->setText(4,5,"DEV05");
bus_event_table->setText(5,0,"BUF_SIZE");
bus_event_table->setText(5,i+1,QString::number(bs[i+1]));
bus_event_table->setText(6,0,"DATA_RATE");
bus_event_table->setText(6,i+1,QString::number(dr[i+1]));
bus_event_table->setText(7,0,"PRIORITY");
bus_event_table->setText(7,i+1,QString::number(pr[i+1]));
bus_event_table->setText(8,0,"FUNCTION");
bus_event_table->setText(8,i+1,rw[i+1]);
bus_event_table->setText(9,0,"MAX_WAIT");
bus_event_table->setText(9,i+1,QString::number(mws[i+1]));
bus_event_table->setText(10,0,"WAIT_SCHEME");
bus_event_table->setText(10,i+1,sd[i+1]);
bus_event_table->setText(11,0,"LATENCY");
bus_event_table->setText(11,i+1,QString::number(LT[i+1]));
bus_event_table->setText(12,0,"BRIDGE");
bus_event_table->setText(12,i+1,QString::number(PB[i+1]));
bus_event_table->setText(13,0,"CONNECT..TO");
bus_event_table->setText(13,i+1,QString::number(con[i+1]));
bus_event_table->setText(14,0,"BUS0_DEV_No");
bus_event_table->setText(14,1,QString::number(nm));
…….//bus1~bus5 parameters
//enabled bus number
bus_event_table->setText(71,1,QString::number(busnum));
 }}
```

(10) **QUIT Push Button** connects to the close () function provided by Qt. When pressed, the simulator will be closed.

2. OUTPUT Graphs and GNU PLOT Files

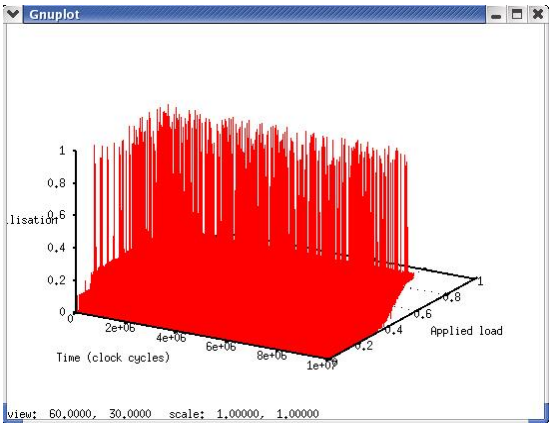
The simulator includes three output graphs: throughput, data tare and histogram; those graphs show an intuitionistic profile about the capacity of our simulated system; information to generate these graphs is written into related files during the simulation process. Those related files are generated by the simulation function as we have mentioned before. Then GNU PLOT can use these files to generate Graphs. To implement GNU PLOT, we need to write codes (.gnp files) for the generation of each graph; GUN PLOT codes are written outside from the simulator; detailed formation of those codes can be find in my appendix; our output includes both 2D (1) and 3D (2) graphs.

Throughput (2D) Plot: generated data will be plotted along the x-axis while transmitted data will be plotted along the y-axis. This graph shows the relationship between generated data and transmitted data. In other words, it shows the capacity of transmission efficiency. Here is a figure of Throughput (2D) Plot:



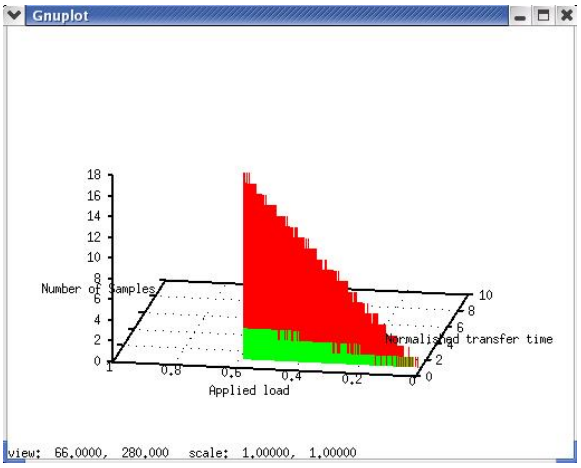
Throughput (2D) Plot

Data tare (3D) Plot: Time will be plotted along the *x*-axis, load along the *y*-axis and bus utilization along the *z*-axis. Time is given in bus cycles, Load is the fraction of the maximum data rate generated by each device, and Bus utilization is the fraction of the bus cycles during which data are actually transferred. If none of the devices insert wait states, the maximum possible value for Bus utilization is 1. If wait states are inserted, fewer bus cycles are available for actual data transfer, and the maximum achievable value for Bus utilization is less than 1. Here is a figure of Data tare (3D) Plot:



Data tare (3D) Plot

Histogram (3D) Plot: The transfer time, t, will be plotted along the x-axis, load l along the y-axis and the number of samples n along the z-axis. Transfer time is given as a normalized transfer time, Load is the fraction of the maximum data rate generated by each device, and Samples is the number of samples observed for this transfer time and load. Here is a figure of Histogram graph:



Histogram (3D) Plot

3. Class throughputview, class datarateview and class histview

All these three classes are used for loading relevant graphs generated by GNU PLOT, their function is just like an image viewer with zoom in and zoom out functions; these classes do not have any templates, just pure code; these three classes share almost the same codes, here, we just give class throughputview as a demonstration.

```
class throughputview:public QWidget
    {
    Q_OBJECT
    public:
```

```
    throughputview(QWidget *parent=0, const char *name=0);
   ~throughputview();
    QLabel *imageLabel1;
    void sizearrange();
    void zoomin();
    void zoomout();
    void load();
    void keyPressEvent(QKeyEvent *e);
    double a,b;
      };
```

All functions and slots in this class are defined as:

```
 throughputview::throughputview(QWidget *parent, const char *name)
   :QWidget( parent, name)
   {
  imageLabel1= new QLabel(this);
  imageLabel1->setScaledContents(true);
  setCaption("ThroughPut_Plot_View");
  a=1.0,b=1.0;
  load();
}
throughputview::~throughputview(){}
void throughputview::sizearrange()
{
      Q_ASSERT(imageLabel1->pixmap());
      imageLabel1->setAutoResize(true);
}
```

Load a graph in a widget based on the file name

```
void throughputview::load(){
    QPixmap pix;
    QWMatrix m;
    QImage image(imagethrouplot);
    if(image.isNull())
    {setCaption("no such files");}// if loading failed….message
    pix.convertFromImage(image);
    m.scale(a,b);
    pix=pix.xForm(m);
    imageLabel1->setPixmap(pix);
    sizearrange();
}
void throughputview::zoomin()
{
    a=a-0.1;
    b=b-0.1;
    load();
```

```
}
void throughputview::zoomout()
{
      a=a+0.1;
      b=b+0.1;
      load();
}
```

Keyboard response definition

```
void throughputview::keyPressEvent(QKeyEvent *e) {
      switch(e->key())
      { case Qt::Key_F1:
          zoomin();
      update();
      break;
       case Qt::Key_F2:
      zoomout();
      update();
      break;
       case Qt::Key_Escape:
      close();}
```

Conclusion

In this project, we implement a multiple buses PCI simulator to support up to six buses; our simulator can simulate basic/ burst read and write behaviors of the PCI system; and this system is extensible for further use. The GUI of the simulator provides a clear and friendly interface to the user; we provide basic input information and out put files management function in the GUI; the GUI is completely compatible with the simulation function, and the GUI is completely separated from the simulation function; the reason why we separate this GUI from the simulation function is that it can eliminate lots of debug problems in the software design and make further extension easier.

References

http://doc.trolltech.com/3.3/aboutqt.html
http://doc.trolltech.com/3.3/designer-manual-1.html
http://doc.trolltech.com/3.3/classchart.html
http://doc.trolltech.com/3.3/functions.html

http://doc.trolltech.com/3.3/classes.html
http://en.wikipedia.org/wiki/Qt_(toolkit)
http://en.wikipedia.org/wiki/Graphical_user_interface
http://www.qiliang.net/qt/designer-manual-4.html
http://searchvb.techtarget.com/sDefinition/0,290660,sid8_gci213989,00.html
http://www.techfest.com/hardware/bus/pci.htm#1.0
http://www.gnuplot.info/docs/gnuplot.html
http://www.quatech.com/support/comm-over-pci.php
http://www.tech-pro.net/intro_pci.html
http://www.udif.com/pcisim/pcisim-frame.html
<Qt 4.2 white paper>
<Qt 3.3 white paper>
<PCI-to-PCI Bridge Architecture Specification> revision 1.1, December 18, 1998
<PCI Hardware and Software --Architecture and Design> *Edward Solari, George Willse*
<PCI Local Bus Specification revision 2.3>
<C++ GUI Programming with Qt 3> *Jasmin Blanchette, Mark Summerfield*