SystemC Modeling of a Parallel Processor Broadcast Interconnection System

Joe Booth; Phase IV Systems Inc.; Huntsville, AL
Jeffrey Kulick; University of Alabama in Huntsville; Huntsville, AL

Keywords: Simulation, Hardware description languages, Parallel processing architectures

ABSTRACT

Modeling of complex hardware/software systems is becoming more difficult due to the complexity of interactions that occur between hardware and software and the need to model each component at multiple levels of detail. System modeling languages such as SystemC are assisting in this area by allowing real application level software to be interfaced with hardware models that maintain great fidelity to the actual hardware realization.

This paper describes a project to develop a model of a large complex hardware/software system that is the heart of a parallel processor interconnection architecture being developed at The University of Alabama in Huntsville. The model developed allows the investigators to vary the parameters of system workload, policy for message passing protocols, and hardware features such as size of elasticity buffers and DMA controller burst size in a single homogeneous model. Initial results are encouraging and the hope is that as SystemC synthesis tools become available, the hardware components of the model can be translated automatically into hardware designs for FPGA and other rapid prototyping platforms without redesign or coding.

1. INTRODUCTION

The operation and performance of complex multi-processor systems is currently being investigated at the University of Alabama in Huntsville. The development of new message passing protocols is of particular interest. SystemC has been chosen as the preferred simulation tool due to its versatility, high modeling fidelity, and ability to express system functionality at various levels of abstraction. This paper presents key aspects of the modeling environment, a general overview of the multi-processor system currently being investigated, and brief review of the simulation's user interface. Source listings for a simple FIFO implementation have been included in the narrative to illustrate the use of SystemC.

2. THE SYSTEMC MODELING ENVIRONMENT

SystemC provides a set of C++ class libraries and a simulation kernel to extend the capabilities of C/C++ as a modeling tool. It supports hardware abstraction at the system, behavioral and register transfer levels and provides a unified modeling environment for systems containing both hardware and software components.

Structural designs are implemented in SystemC using modules, ports, processes and signals. A variety of data types are supported to include single bits, bit vectors and fixed-point integers. Four value logic signals are also supported to allow the implementation of tri-state buses.

Modules lie at the heart of SystemC. They act as containers for collections of ports, signals, internal data structures and processes that together form a system component. The use of modules allows a complex system to be divided into a number of simpler elements early in the design process. Modules also allow the verification of individual system components that greatly simplifies the task of model validation. A module's interface to other modules is public while its internal structure is hidden from the remainder of the system. Modules may easily be substituted with other modules as long as their interfaces and basic functionality are consistent. As a result, modules may be revisited and revised as necessary during the course of a design's evolution. Modules are declared using the SystemC keyword SC_MODULE.

Ports define how a module communicates. They provide the data path between a module's environment and its internal processes. SystemC supports three types of ports: in, out and inout. Input ports are declared using the SystemC port mode SC_IN; output ports are declared with SC_OUT; while inout ports are declared with SC_INOUT. The data type of each port can be defined as a standard C++ data type, a SystemC type or a user-defined type. This rich set of data types allows an engineer to concentrate on basic system functionality in the early stages of a design. Later, once the system's operational aspects have been verified, port data types can be revised to improve hardware fidelity. An example of this is the conversion of a memory address port from an unsigned integer representation to a bit vector to model a finite bus width.

SystemC signals are closely related to ports. They provide the data paths necessary to connect devices. Signals join modules at a system's top level. In hierarchical designs, signals are used inside a module to interconnect lower level modules. Unlike ports, signals don't have a mode attribute since the ports being connected determine the direction of data flow. Signals are declared using the SystemC keyword SC_SIGNAL.

Processes are internal functions that establish a module's behavior and functionality. They respond to changes in module inputs by updating the state of internal data structures, by altering values present at the module's output ports, or both. SystemC provides three types of processes: methods

declared with the SystemC keyword SC_METHOD; threads declared with the keyword SC_THREAD; and clocked threads declared with SC_CTHREAD.

Methods are the simplest form of SystemC process. Their execution is triggered by signal events associated with a method sensitivity list. When a signal associated with a method's sensitivity list changes state, the method is invoked. Once triggered, a method will run until a return is encountered or the end of the method routine is reached. Control is then returned to the simulation kernel.

Thread processes are similar to methods in that they are triggered by events associated with a process sensitivity list. However, their execution may be suspended through the use of wait() functions. Once a thread process is suspended it will wait until an event it is sensitive to occurs. At that time, execution of the thread resumes where it left off and it continues to run until the next wait() is encountered or the module's end is reached. Thread processes are useful in modeling controllers where device state information must be maintained between events.

Clocked thread processes are a specialization of the thread process. Their sensitivity list is limited to one edge of a single clock input that matches the way a clocked hardware device typically functions. This sensitivity restriction provides a more realistic device model that is easier to synthesize. Execution of a clocked thread may be suspended in a manner analogous to the thread process using wait() and wait_until() functions.

The parallel processor system being studied in this paper employs FIFOs as elasticity buffers in its communications channels. A SystemC module declaration for standard FIFO is shown below:

```
// FIFO.h
#include <stdlib.h>
#include <malloc.h>
#include "systemc.h"
#include "parameters.h"

struct fifo : sc_module {

// I/O Interface Definitions
public:
  sc_in<sc_bit>                      clear;
  sc_in<sc_uint<DATA_SIZE> >         data_in;
  sc_in<sc_bit>                      data_write;
  sc_out<sc_uint<DATA_SIZE> >        data_out;
  sc_in<sc_bit>                      data_read;
  sc_out<sc_bit>                     fifo_empty;
  sc_out<sc_bit>                     fifo_half_full;
  sc_out<sc_bit>                     fifo_full;

// Local Data
private:
  bool                               empty_fifo;
  bool                               half_full_fifo;
  bool                               full_fifo;
  int                                read_index;
  int                                write_index;
  sc_uint<DATA_SIZE>                 *fifo_data;

public:
  void fifo_clear();
  void fifo_write();
  void fifo_read();

// Module Constructor
  SC_CTOR( fifo ) {
       SC_METHOD( fifo_clear );
       sensitive_pos << clear;
       SC_METHOD( fifo_write );
       sensitive_pos << data_write;
       SC_METHOD( fifo_read );
       sensitive_pos << data_read;

       empty_fifo = true;
       half_full_fifo = false;
       full_fifo = false;
       read_index = 0;
       write_index = 0;

       fifo_data = (sc_uint<DATA_SIZE> *)
         malloc( FIFO_DEPTH * sizeof( sc_uint<DATA_SIZE> ));
  }

// Module Destructor
  ~fifo() {
       free(fifo_data);
  }
};
```

The include files provide access to standard C++ functions, SystemC classes, and static simulation parameters. The SC_MODULE declaration is followed by a list of module inputs and outputs. Ports data_in and data_out have fixed precision unsigned integer types whose widths are set by the simulation parameter DATA_SIZE. Local private storage is defined for the module to hold FIFO state information and data pointers. The module constructor, declared with SystemC keyword SC_CTOR, allocates FIFO memory and establishes the module's processes, sensitivity lists, and initial FIFO state. Finally, a module destructor is provided to free allocated memory when the object associated with a module instantiation is destroyed.

The FIFO module has now been declared, but to be useful, an instance it must be created (instantiated). Modules like this one may be instantiated exclusively from SystemC's top-level routine SC_MAIN to realize a flat design. Hierarchical designs result when modules are instantiated from within other modules as lower level components. Below is a partial listing of a receiver declaration that creates an instance of the FIFO module. Please note how signals are bound to the FIFO following its instantiation.

```
// Internal Signal Definitions
  sc_signal<sc_uint<DATA_SIZE> >     shift_data;
  sc_signal<sc_bit>                  shift_data_ready;
  sc_signal<sc_uint<DATA_SIZE> >     fifo_data;
  sc_signal<sc_bit>                  fifo_empty;
  sc_signal<sc_bit>                  fifo_half_full;
  sc_signal<sc_bit>                  fifo_full;
  sc_signal<sc_bit>                  fifo_read;

// Other signal declarations omitted
// Component Objects
  rcv_shift                          *shift;
  fifo                               *rcv_fifo;
  rcv_overrun                        *over_cnt;
  rcv_cntrl                          *cntrl;
```

```
  rcv_buf                    *buffer;
  rcv_dma                    *dma;

public:
  unsigned long read_overruns();
  unsigned long read_errors();

// Module Constructor
  SC_CTOR( rcv_channel );

// Custom Constructor Used to Accept Channel and Processor ID's
  rcv_channel( sc_module_name, int channel_number, int proc_number )
  {

// Instantiate component objects.
    shift = new rcv_shift ("shift");
    rcv_fifo = new fifo ("fifo");
    over_cnt = new rcv_overrun ("over_cnt");
    cntrl = new rcv_cntrl ("cntrl", channel_number, proc_number);
    buffer = new rcv_buf ("buffer");
    dma = new rcv_dma ("dma", channel_number);

// Bind FIFO object to shift object and RCV controller object.
    rcv_fifo->clear(reset);
    rcv_fifo->data_in(shift_data);
    rcv_fifo->data_write(shift_data_ready);
    rcv_fifo->fifo_empty(fifo_empty);
    rcv_fifo->fifo_half_full(fifo_half_full);
    rcv_fifo->fifo_full(fifo_full);
    rcv_fifo->data_read(fifo_read);
    rcv_fifo->data_out(fifo_data);

// Rest of module not shown
```

SystemC is an efficient development environment that is maturing quickly. The intent of this brief review was present the language's key constructs and provide some indication of its power. Additional information and software may be obtained at the SystemC web site: http://www.systemc.org.

3. MODELING A PARALLEL PROCESSOR SYSTEM WITH BROADCAST INTERCONNECTIONS

As part of an ongoing research project at University of Alabama in Huntsville, SystemC has been used to simulate the operation of a complex multi-processor device. The device being studied consists of an array of processing nodes interconnected by a fiber-optic network. A brief description of the system and each component model will now be presented.

3.1. System Architecture

The multi-processor system consists of several processing nodes, and the nodes are interconnected through a series of fiber-optic links that pass serial data. Figure 1 shows a functional diagram of a typical node. Each node is composed of a processing element, a dual-port global memory module, a global bus arbiter, one serial transmit module and N serial receive modules (where N represents the number of processors in the system). For the purpose of simulation, inter-processor data is transmitted in simple packets whose format is given by Table 1. The output of each transmitter fans out to all network nodes, and the packet's target address is used to direct packets to their intended destination.

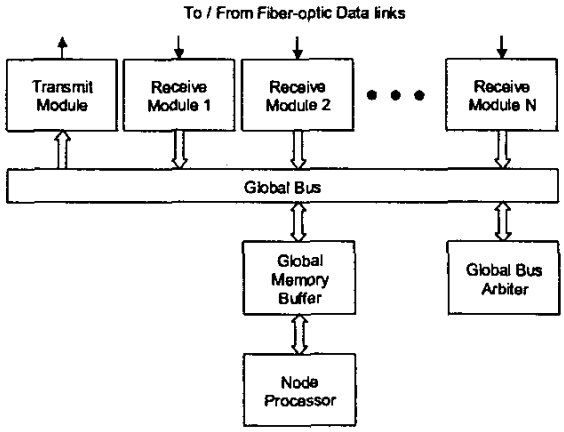

Figure 1. System Node Module.

Three packet types are modeled in this simulation. The first type is a point-to-point data packet directed from a single source node to a single target node. This will be referred to as a single-cast packet transmission. Loop-back transmissions are permitted (source and target addresses are identical). The second packet type is transmitted from a source node to all network nodes concurrently. This is referred to as a multi-cast packet transmission. The last packet is an acknowledge packet sent in response to a received packet.

Table 1. Simulated Packet Definition.

Packet Location	Packet Entry
1	Start of Packet Code
2	Packet ID
3	Packet Type
4	Source Address
5	Target Address
6	Data Section Word Count
7	First Packet Data Word
8	.....
9	End of Packet Code

3.2. Network Data Links

To model communications delays between system processors, each network data link is modeled as a digital delay element with a variable number of delay stages. The data links are synchronized to the master simulation clock; therefore, the modeled delays are multiples of the simulation clock period. The number of delay stages present in each link is defined in the simulation parameters file.

3.3. Network Nodes

As stated earlier, each system node consists of a processing element, a dual-port global memory module, a global bus arbitration unit, one serial transmitter and N serial receiver modules (where N represents the number of processors present in the system). A description of each node hardware element will now be presented.

3.4. Node Processor

Node processors are the source and destination of all data transfers, and they initiate all simulation activity. Three clocked threads are used to implement the processor: a packet receive process; a process to generate new outgoing packets and an outgoing packet transmission process.

The processor is modeled as a free running message generator that produces a new transmission packet at a rate determined by the simulation parameters file. When a new outgoing packet is created, a random number between 0 and 1 is generated, and thresholds are applied to determine if the transmitted packet is a single-cast or multi-cast packet. If it is to be a single-cast packet, a second random number is generated to select the packet's target address.

3.5. Global Memory Buffer

The global memory buffer serves as a holding tank for incoming and outgoing communications packets. It is implemented as a dual-port random access memory with a processor bus interface and the global bus interface. All aspects of the buffer (size, data width, etc.) are controlled through the simulation parameters file. The buffer's global bus interface employs tri-state signals for its data path since several devices share the bus. The tri-state signal type is not used for the processor data bus since it isn't necessary and requires more simulation time to model.

3.6. Global Bus Arbiter

Each communications channel uses the global bus to exchange information with the global memory buffer. The global bus arbiter employs hardware handshakes to coordinate activity on the global bus. Each potential bus master must request the bus and wait for a bus grant before it may access the bus. The arbiter polls bus requests in a round robin fashion starting with the transmit channel. When a bus request is found, the arbiter issues a bus grant to the associated channel and then waits. When the channel has finished, it drops the bus request, and the arbiter resumes its polling operation with the next channel in sequence. The node processor uses a burst count control word to limit how long a channel may be the global bus master. This helps to prevent communications bottlenecks due to bus hogging.

3.7. Transmit Channel

The node transmit channel consists of a transmit shift register, FIFO and DMA unit. Figure 2 shows a functional diagram of the transmit channel assembly.

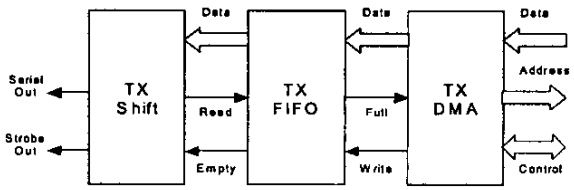

Figure 2. Transmit Module.

Transmit Shift Register. The transmit shift register converts parallel data held in the transmit FIFO to a serial data stream transmitted across a network data link. Data is transmitted most significant bit first. The shift register is implemented as a SystemC clocked thread process, which monitors the state of the FIFO empty flag and reads FIFO data as it becomes available.

Transmit FIFO. During a packet transmission, the transmit DMA unit must request the global bus repeatedly if the packet is too large to access in a single burst. When there is a lot of contention for the global bus, bus grants may be delayed; resulting in the potential for stalled transmissions. In an attempt to keep the transmit shift register busy and increase throughput efficiency, a FIFO has been incorporated into each transmit channel to act as an elastic buffer.

The FIFO is implemented using three SystemC methods. The first method provides for asynchronous clears of the FIFO device, and it is sensitive to the positive-going edge of the clear input. The second and third methods handle FIFO read and write requests.

Transmit DMA Unit. The transmit DMA unit automates the transfer of outgoing packet data from the global memory buffer to the transmit FIFO. After reset, the transmit DMA unit begins polling its start input. When a new transmission packet is ready, the node processor activates the start DMA bit to initiate a transfer. Once the transfer is underway, the DMA unit will access the global bus in short bursts until all packet words have been written to the FIFO. The node processor sets the size of DMA burst transfers.

3.8. Receive Channels

Node receive channels consist of a receive shift register, FIFO, receive controller, local buffer and DMA unit. Figure 3 shows a functional diagram of a receive channel assembly.

Receive Shift Register. The receive shift register accepts serial data obtained from one of the network fiber-optic data links, converts it to a parallel data word, and writes it to the receive FIFO. It accepts serial data most significant bit first and continues to collect data until a full word has been accumulated.

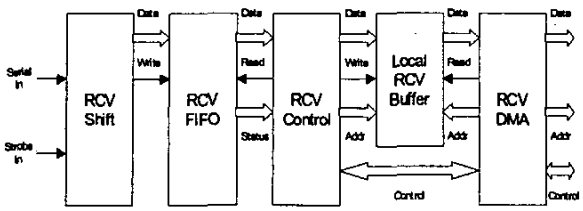

Figure 3. The Node Receiver.

Unlike the transmit shift register, which is implemented as a clocked thread process, the receive shift register is implemented as a SystemC method. The method is sensitive to the positive-going edge of receive data strobe

Receive FIFO. The receive FIFO performs a function very similar to that of the transmit FIFO. The amount of time required to transfer incoming packet data to global memory is directly related to the amount of global bus activity. The receive FIFO provides an elastic buffer necessary to prevent receiver overruns when the node's global bus is congested. Implementation of the receive FIFO is identical to that of the transmit FIFO.

Receive Controller. The receive controller module acts as an intelligent mediator for receive channel operations. Currently two receiver modes are supported: cut through and store and forward.

In cut through mode, the receive controller talks to the DMA controller directly and the local buffer is bypassed. The receive controller starts a DMA transfer to global memory as soon as a valid packet header has been received. The receiver's packet ready flag will automatically be set at the completion of the DMA transfer. The packet error flag will also be set at this time if any packet errors have been detected during the transfer.

In store and forward mode, the receive controller writes packet data to local buffer memory. The controller examines the incoming packet as it is received. Erroneous packets are discarded, and the packet error flag is set. When a valid packet is received, a DMA transfer from local buffer memory to global memory is initiated. The receiver's packet ready flag is then set automatically at the completion of the DMA transfer.

Local Receiver Buffer. The local receive buffer is a modified dual-port memory device. The buffer has a mode input that selects the current operating state. In store and forward mode, the receive buffer operates as a standard memory element and stores packet data. When cut thru mode is selected, the local receive buffer operates transparently, and the receive controller is connected directly to the receive DMA unit.

Receive DMA Unit. The receive DMA unit automates the transfer of incoming packet data to global memory. After reset, the receive DMA unit begins polling its start input. Unlike the transmit DMA unit, the receive DMA's start input is manipulated by the receive controller module, not the processor. When the receive controller is ready to pass incoming packet data to global memory, it activates the start DMA bit to initiate a transfer. The DMA unit will always get data via the local receive buffer; however, full rate transfers will occur only when the receiver is in store and forward mode. Once the transfer is underway, the DMA unit will access the global bus in short bursts until all packet words have been written to global memory. The node processor sets the size of DMA burst transfers.

4. OPERATIONAL ASPECTS OF THE SIMULATION

4.1. Entering Simulation Parameters

The simulation accepts compile-time and run-time parameters as a means of constraining the system being simulated. Compile-time parameters are hard coded into the simulation and may be accessed via the simulation parameters file. Altering definitions in the parameters file requires that the simulation executable be rebuilt, so frequent changes of this sort are not desirable. The following compile-time parameters are found in parameters file: transmit clock period, global bus clock period, processor clock period, processor data bus width, receive local buffer size, global double buffer size, DMA burst counter widths, Fiber-optic data link delay, number of system processors, and the inter-processor packet definitions.

Run-time parameters are entered via a simulation initialization file. These variables change too often during a parametric study for simulation rebuilds to be practical. For a large number of simulation runs, a batch process may be defined which generates a series of initialization files automatically and calls the simulation executable in a repetitive fashion. The following run-time parameters are found in the initialization file: inter-processor packet size, size of receive and transmit FIFO's, DMA burst transfer size, single-Cast packet transmission probability, multi-Cast packet transmission probability, VCD trace file creation flags, and the processor and communications channel IDs associated with the trace file.

4.2. Simulation Outputs

The simulation generates a log file automatically at the completion of a test run. The log file includes network performance statistics which include: the number of system processors, FIFO depth, size of data packets, DMA burst size, packet transmission probabilities, and the resulting number of packet overruns. Additionally, the simulation outputs a VCD trace file as shown in Figure 4.

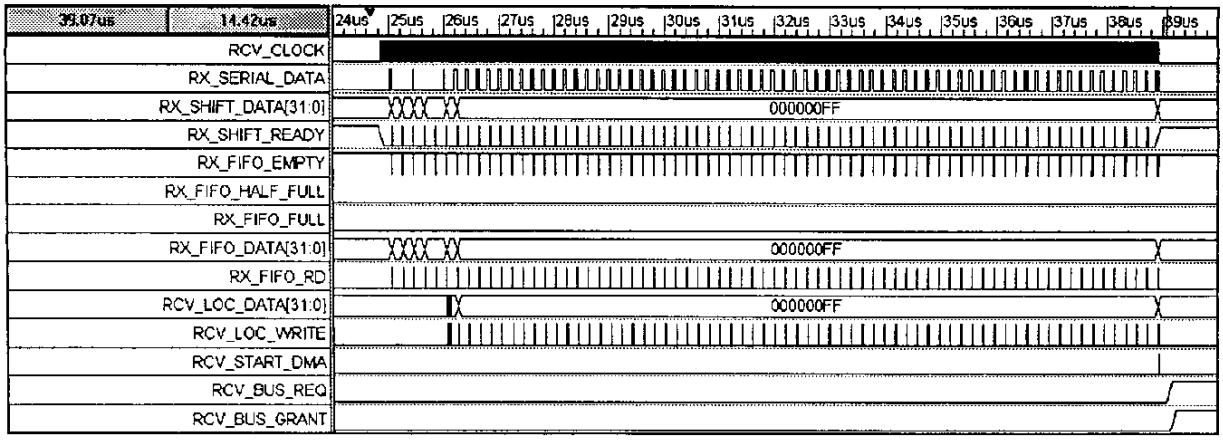

Figure 4. VCD Trace File Showing Receive Channel Activity.

5. SYSTEMC APPLICATIONS DEVELOPMENT

This section briefly presents areas where the SystemC design environment needs improvement. This is by no means a condemnation of SystemC. Rather it is list of areas where the open source distribution needs a little refinement. Several of these improvements already exist in the form of proprietary software builds.

Currently, the C++ development environment to which SystemC is attached restricts the designer. Most provide decent class browsers, but few support the type of hierarchical navigator that hardware designers have come to expect from languages like VHDL and Verilog. A text based navigator that shows the interrelationships of SystemC modules would be quite helpful. Even more helpful would be a graphical navigator that shows the interconnections between modules and allows the designer to graphically "dive" into lower level modules to show their internal structures.

Trace files are particularly useful in simulation debugging and documentation. SystemC currently does not include waveform viewers. However, there are several excellent shareware and freeware viewers available over the Internet. A windowed interactive debugger that presents graphical representations of the system (possibly in schematic form), text based code modules and selected device waveforms would be a welcome addition to SystemC.

There are currently proprietary translators that produce VHDL modules from SystemC code. The VHDL in turn may be used to program gate arrays or produce custom ASICs. Eventually a native mode hardware compiler and device fitters should be incorporated into SystemC.

CONCLUSIONS

The SystemC open source initiative was founded in 1999. In that time, the language has evolved a great deal. It has the support of industry and the engineering community at large; it provides a unified modeling approach for system software and hardware; and it allows the designer to represent system architectures at various levels of abstraction and detail. With minor revisions, SystemC will become a desirable alternative to VHDL and Verilog.

REFERENCES

1. SystemC User's Guide Version 2.0, May 2001, http://www.systemc.org
2. S. Swan, "An Introduction to System Level Modeling in SystemC 2.0", 2001, Cadence Design Systems Inc.
3. G, Economakos, P. Oikonomakos, et al, "Behavioral Synthesis with SystemC", 2000, Europen SystemC Users Group, http://www-ti.informatik.uni-tuebingen.de/~systemc/
4. K. Kranen, A.Ghosh, P. Hardee, SystemC User's Forum Presentation, June 2000, DAC
5. K. Bartleson, "A New Standard for System-Level Design", 1999, Synopsis

BIOGRAPHIES

Joe Booth
Phase IV Systems Inc.
3405 Triana Blvd
Huntsville, AL 35805

e-mail: jbooth@phaseiv.com

Joe Booth is a senior development engineer at Phase IV Systems Incorporated in Huntsville, AL. Before joining Phase IV he worked for the U.S. Army Missile Command as a research and development engineer at the Weapons Sciences Directorate. He holds a BS in Electronics Engineering from the University of Missouri in Columbia and a MS in Electrical and Computer Engineering from the University of Alabama in Huntsville.

Jeffrey H. Kulick
Department of Electrical and Computer Engineering
The University of Alabama in Huntsville
Huntsville, Al. 35899

e-mail: kulick@ece.uah.edu

Jeffrey Kulick is a Professor in the Department of Electrical and Computer Engineering at The University of Alabama in Huntsville where he teaches and does research in the area of computer systems architecture and operating systems. Before joining UAH he previously held positions at Queen's University in Kingston Ontario, Canada and visiting positions at the MIT Media Laboratory where he worked in holographic displays. He has a B.Sc. in Engineering Physics from New York University's College of Engineering and a M.Sc and Ph.D from the Moore School of Electrical Engineering at the University of Pennsylvania.