Fall 2008

Application development process for GNAT, a SOC networked system

Christopher L. Plumlee
University of New Hampshire, Durham

Application Development Process for GNAT, A SOC Networked System

by

Christopher L. Plumlee

BSEE, University of New Hampshire, USA, 1998

THESIS

Submitted to the University of New Hampshire

in Partial Fulfillment of

the Requirements for the Degree of

Master of Science

in

Electrical and Computer Engineering

September, 2008

This thesis has been examined and approved.

Thesis Director, Dr. Andrzej Rucinski, Professor

Dr. John LaCourse, Professor and Department Chairman

Dr. W. Thomas Miller, Professor

Dr. Thaddeus P. Kochanski, Affiliate Professor

Dr. Pieter Mosterman, Senior Research Scientist, The MathWorks

Date

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

Len Long, CIDlab Waltham, Director.

Donald P. Bliss, NI2 Center for Infrastructure Expertise, Director.

Dale Ferriere, NI2 Center for Infrastructure Expertise, Deputy Director.

Frank Hludik, ECE Department, Faculty Member.

Xilinx Inc.

The Mathworks.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

AIN	-	Ambient Intelligent Network
ASIC	-	Application-Specific Integrated Circuits
CAD	-	Computer Aided Design
CPU	-	Central Processing Unit
DSP	-	Digital Signal Processing
DVT	-	Design Verification Test
EDK	-	Embedded Development Kit
FPGA	-	Field Programmable Gate Array
GNAT	-	Global Network Academic Test
GPL	-	General Public License
IC	-	Integrated Circuit
IPC	-	Information Processing Capacity
ISE	-	Integrated Software Environment
PDA	-	Personal Digital Assistants
SoC	-	System-on-a-Chip
HDL	-	Hardware Description Language

ABSTRACT

Application Development Process for GNAT, A SOC Networked System

by

Christopher L. Plumlee

University of New Hampshire, September, 2008

The market for smart devices was identified years ago, and yet commercial progress into this field has not made significant progress. The reason such devices are so painfully slow to market is that the gap between the technologically possible and the market capitalizable is too vast. In order for inventions to succeed commercially, they must bridge the gap to tomorrow's *technology* with *marketability* today. This thesis demonstrates a design methodology that enables such commercial success for one variety of smart device, the Ambient Intelligence Node (AIN). Commercial Off-The Shelf (COTS) design tools allowing a Model-Driven Architecture (MDA) approach are combined via custom middleware to form an end-to-end design flow for rapid prototyping and commercialization. A walkthrough of this design methodology demonstrates its effectiveness in the creation of Global Network Academic Test (GNAT), a sample AIN. It is shown how designers are given the flexibility to incorporate IP Blocks available in the Global Economy to reduce Time-To-Market and cost. Finally, new kinds of products and solutions built on the higher levels of design abstraction permitted by MDA design methods are explored.

STATEMENT OF WORK

This thesis is a partial product of a collaborative work. It is published simultaneously with a companion thesis, "Architecture for GNAT, a SOC Networked System" by Tomasz Jankowski, MSEE. The following table details how the research was divided between the two authors:

Tomasz Jankowski	Joint Effort	Chris Plumlee
	Statement of Work	
Introduction		Introduction
Ambient Intelligence Environment		
Ambient Environment Definition		
Ambient Technology Mapping	Node Design Requirements	
Ambient Intelligence Environment: Architecture		Model Driven Architecture
Ambient Intelligence Node Implementation	Implementation	Node Design
	And	DSP Block Design
	Integration	Node Software
Results		Development Roadmap
Development Roadmap		Results
		GNATDVD

Table 1: Statement of Work

The depth of the material in the two theses varies accordingly. Topics that are covered in-depth in Mr. Jankowski's thesis are only summarized here. For a thorough understanding of the subject matter, it is recommended that the reader review both theses.

CHAPTER 1

INTRODUCTION

1.1 Motivation

The traditional embedded system design process has become cost-prohibitive. Accelerating marketing cycles do not always allow the teams to compare requirements against common-off-the-shelf (COTS) chipsets and carefully pick the cheapest solution. It is not that the selection is a long process, but that the months following selection must be spent combining systems coherently on a board and in a software interface to yield the desired product. Hardware changes then result in mandated certification steps followed by reliability studies. There are a variety of markets that open and close far faster than these standard design steps can be completed, and there now is a valid design process that allows one to take advantage of such opportunities.

A design niche characterized by fast market-cycle opportunities was defined by Tomasz Jankowsi in his related thesis, titled "An Architecture and Technology for Ambient Intelligence Node"[1]. The economies of scale currently favor the higher levels of optimization that come along with the longer development cycles of traditional methods for both large products and small products. Traditional design methods are capable of churning out large-scale, high profit-margin products such as data-centers small scale, high quantity products such as optosensors. Complex medium-sized devices, such as the

Ambient Intelligence Node (AIN) defined by Tomas Jankowski, deliver very thin profit margins when designed using traditional methods. It is the AIN design niche that is focus of the design methodology described herein.

The motivation of this thesis is to produce and demonstrate an end-to-end development process for reliably producing viable products on such an accelerated timetable. The example product, GNAT (Global Network Academic Test) is a generic solution incorporating many key functions necessary for modern products. It is built on a Model Driven Architecture (MDA) so that features can be added, removed, or reconfigured easily. While this architecture allows for efficient design reuse, this advantage is eclipsed by the ease with which 3rd party Intellectual Property (IP) modules from the Global marketplace can be integrated into the system. Furthermore, much of the design is at the FPGA and CPU "software" levels, allowing designers to shorten or eliminated hardware build and certification cycles prior to new product introduction.

1.2 Design Requirements

GNAT is a sample Ambient Intelligence Node (AIN), developed to allow testing of a universal design template for devices in the AIN genre. Such devices are defined in Tomasz's work as having the following requirements:

- **Information Processing Capacity (IPC)** – The node does not merely collect and relay data. It must make intelligent decisions about all the data and noise presented to it, in order to avoid idle chatter within the overarching AIN system.

- **Ranged Communication** – in order to be an Ambient Node, each device must be capable of communications over at least one medium, preferably using Internet Protocol (TCP/IP).
- **Power requirements** - Capabilities do not come free; power goes in, data comes out. Nodes on this scale should be relatively small and unobtrusive. The ability to run temporarily on batteries or permanently from solar panels would further expand this scale of node's marketability.
- **Adaptability** – Needs change. In order for a Node to remain useful for its entire operating life, it will need to be re-purposed. If this can be done remotely, the node becomes exponentially more useful and linearly more profitable.
- **Autonomy** - The basic node design must capture all this and yet be self-contained. Any node must be able to carry out some function on its own in case it is not deployed in, or becomes cut off from an overarching system.
- **Short Design Cycle** – Devices in this scale are not deployed in great enough numbers nor high enough profit margin to justify a lot of engineering investment. Even devices that see great success are often replaced by follow-on devices with greater capabilities before their designed operational lifetimes have expired.

1.3 Architecture

A specialized set of tools can only be crafted if the nature of the job at hand is well-defined. This is why the AIN Architecture specified in Tomasz's work is an important part of this thesis. An Ethernet interface is specified for its adaptability; off-the-shelf products are available to transform Ethernet into just about any other interface desired. A CPU is specified to manage Ethernet communications, implement protocols, and provide the Intelligence (via software) demanded of AIN products. An FPGA is specified to provide DSP capabilities and interfaces to the ambient environment while meeting the low-power requirements imposed on such products. The Xilinx Virtex-II FPGA with integrated CPUs is the best market fit for these needs at the time of this document's publication[3]. Again, traditional design methods already allow combination of all of these components. A well-implemented Model-Driven Architecture tools set will allow such combinations to be designed much faster and cheaper.

CHAPTER 2

MODEL DRIVEN ARCHITECTURE

Model Driven Architecture (MDA) is a design methodology that comes from the application of classic software design methods to the hardware realm. As Object Oriented software has allowed for reuse of modular software "objects", MDA design permits reuse of hardware "function blocks". Furthermore, such blocks can be bought and sold as Intellectual Property (IP) Blocks, a trading practice that permits designer to focus more on integrating existing technologies and less on reinventing them. In the GNAT example project, off-the-shelf MDA tools and existing practices are combined into a universal system for realizing Ambient Intelligent Network designs.

2.1 Concepts and Challenges

One viewpoint on system design is top-down modeling, with a focus shifting from software to FPGA. The FPGA industry has benefited from the same technological advances that advance the entire semiconductor industry at a breakneck pace. The resulting increase of design space in each FPGA allows (and from a marketing perspective requires) the creation of higher-level functions and greater quantities of features in each product. As a result, said designs become too complicated for a single person or even a small team to design in a reasonable time period. One solution is the concept of reusable blocks reconnected and recycled through various projects. These modules with predefined functions and interfaces can be organized in a systematic way for quick and easy re-use. Theoretically.

While there has been a lot of progress over the last ten years in the area of design reuse, it remains an elusive genie that no Computer Aided Design (CAD) tools company has yet managed to bottle and sell. Aiming to fix this technological gap, some of the most advanced design tools on the market were obtained. Xilinx and The MathWorks provided UNH with use of their Embedded Development Kit (EDK) and Simulink® software. These companies also provided quite a bit of technical support in getting these tools to work together (See APPENDIX I: Installation Procedure for Commercial). In order to prove out the design environment, it was necessary to craft a sample AIN Node design using mainly off-the-shelf IP Cores. Any new tools or IP Cores necessary to complete the task were designed for flexibility and included with this thesis on the GNATDVD distribution as the missing links required for quick-and-easy AIN Node design.

As an arch has its keystone, EDK is the central and most important tool in the AIN Node design process. The IP Cores used to build the AIN system have hardware and software components, and both are managed in the relatively orderly EDK graphical environment. It is good to remember that each of these modules was created by a design team with reusability in mind. In this chapter there will be a brief introduction to the design methodologies used to develop these component parts.

The current state of FPGA design tools today is best understood from a historical perspective. The evolution of FPGA hardware has adhered to "Moore's Law", doubling the gate-count of the new chips every two years. FPGA design tools, of course, started out simplifying hundreds of gates – then had to adapt to handle hundreds of thousands of gates and other structures. Linear thinking has yielded design tools that can be classified into two categories:

- Low-Level tools for designing basic elements.
- High-Level tools for combining lower-level elements into higher level IP Cores.

While the low-level tools came first, they are still necessary (and in common use) for developing optimized or tweaking existing high-level IP Blocks.

Some designers were frustrated with this "Bigger is Better" approach to design tools, and started fresh. SystemC was born a top-down design approach modeled after software programming. While this approach is similar to using Hardware Description Languages (HDL) already in general use, SystemC's roots in the C programming language give a much broader variety of structuring options. SystemC is a set of software libraries implemented in C++ that can be compiled to an arbitrary mix of software and hardware targets. This not only allows much simpler software design and simulation than is available in existing HDL simulators, but provides flexibility in regarding CPU vs. dedicated hardware design trade-offs. The picture below presents the shift in the standard of the design methodology for the hardware cores, and it would appear that SystemC can offer the best solution for AIN design needs. Unfortunately, this option had to be eliminated due to the lack of a cohesive, mature SystemC toolset at this time.

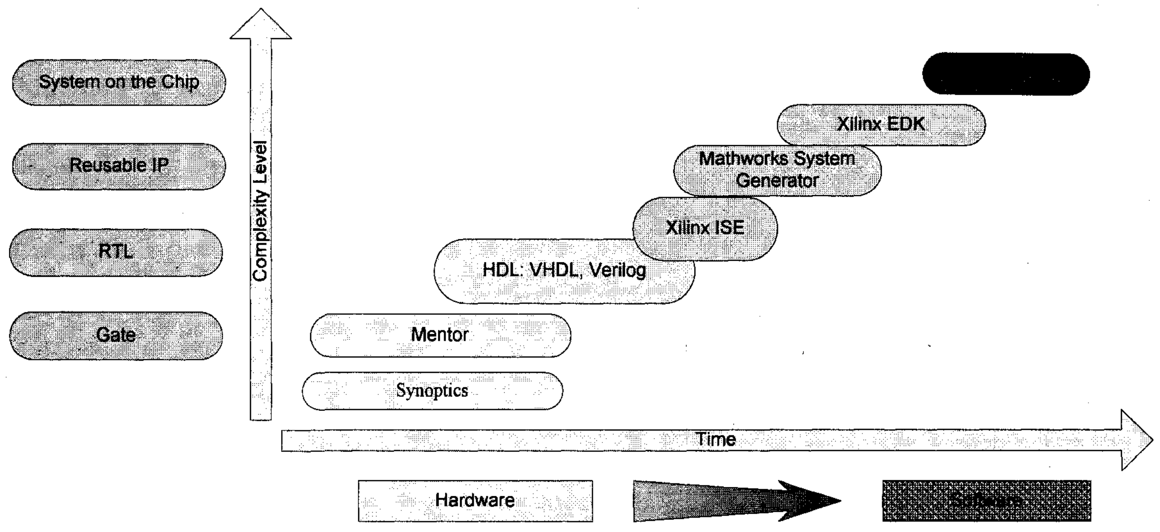


Figure 1: History of Design Tools

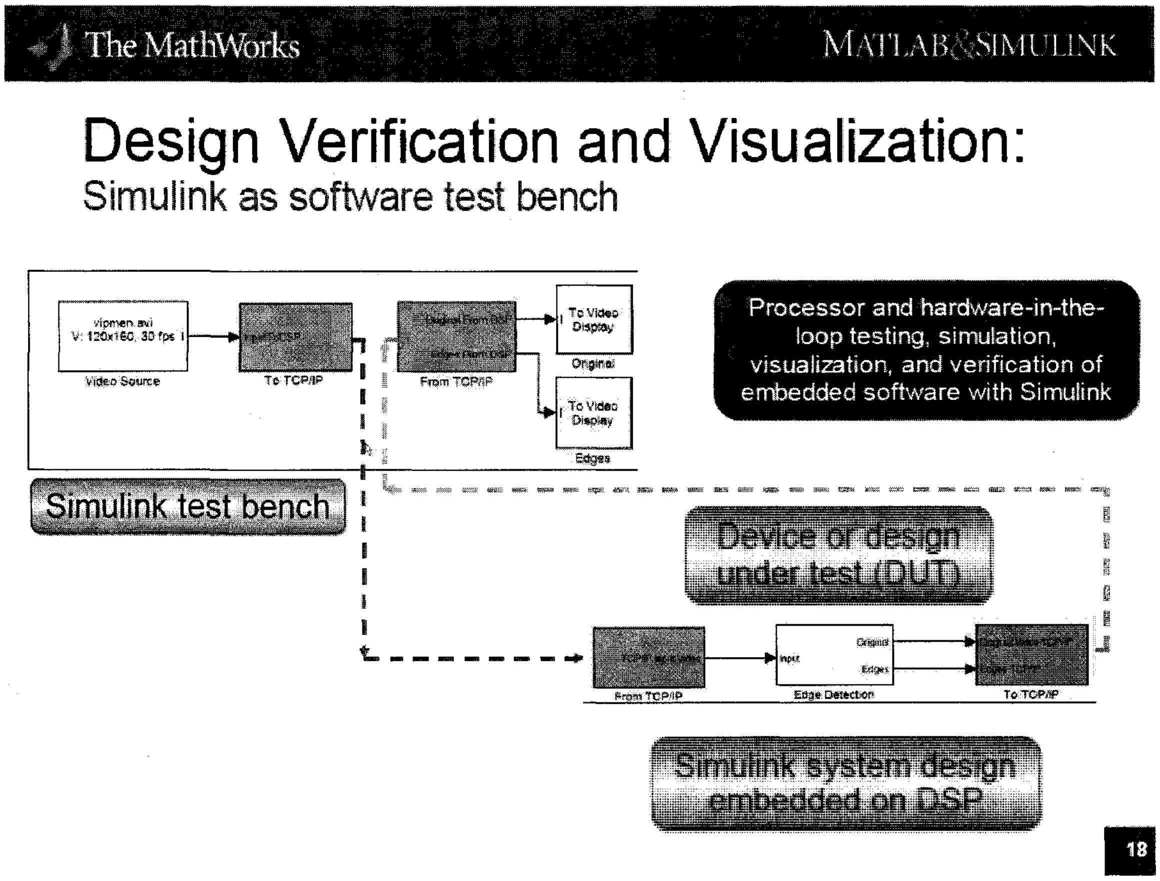
2.2 IP Blocks

There are two main options for creating a function block (IP Core) for an FPGA, and designers often chose to mix these methods. The low-level method, HDL coding offers the choice of two Hardware Description Languages: VHSIC HDL (VHDL) and Verilog. Since Verilog was originally designed as a Design Verification Test (DVT) programming language, VHDL is more focused on hardware design -- and will be the language of choice in the examples presented here. The high-level method, schematic design, is supported by a variety of tools available from FPGA vendors such as Xilinx and Altera, as well as CAD companies such as Mentor Graphics and Synoptics. High-level schematic design packages all produce low-level VHDL/Verilog code output, which can optionally be optimized by hand. Either way, the low-level code must be synthesized into a bitstream before it can be loaded onto an FPGA device. The advantages and disadvantages of HDL methods are detailed in the following table:

Advantages:	Disadvantages:
Portability – the code can be adapted by almost all off-the-shelf implementation and integration tools	Development time increases exponentially with the design complexity.
Best design visibility and troubleshooting for small projects	Worst design visibility and troubleshooting for complex projects
Complete control of the design	Steep learning curve.
Best optimization	

Table 2: Trade-offs of HDL Design methods

Due to these tradeoffs, it is common practice to rough-out a new IP Block in the Schematic tools, then fine tune the resulting VHDL code by hand. With this approach in mind, Xilinx developed a tool called System Generator. It allows the designer to utilize the schematic block approach similar to the StateCAD tools, but said modules are written in HDL. The schematic block approach is a very easy way for designers to generate more complex hardware designs without controlling many lines of source code. The next evolution is this tool's inclusion into an even higher-level modeling environment such as The MathWorks Simulink® software. This combination of tools gives the designer single-click access to very powerful DSP design, test, and tune-up blocks, system/design visibility, and open architecture. The design can be quickly verified and tweaked in the software simulation. The design tools flow easily to the hardware level through the System Generator "Hardware-in-the-Loop" design verification process, as shown in the following Simulink® design process example[2]:



Design Verification and Visualization:

Simulink as software test bench

Processor and hardware-in-the-loop testing, simulation, visualization, and verification of embedded software with Simulink

Table 3: Simulink® Hardware-in-the-Loop Example

When it comes to schematic design specialized for Digital Signal Processing (DSP), The MathWorks Simulink® software appears to be the only well-developed tool on the market at this time.

Let us take a moment to reflect on the significance of the DSP component in AIN design. The table below presents several common examples of the acceleration provided by a DSP over an equivalent General Purpose Processor (GPP) executing the same operation in software.

Application	Software Simulation Time (seconds)	Hardware Simulation Time (seconds)	Speed-up
Image Filtering	676	6	112X
QAM Demodulator + Extension	1203	18	67X
5 x 5 Image Filter	170	4	43X
Cordic Arc Tangent	187	27	7X
Additive White Gaussian Noise Channel	600	80	7.5X

Table 4: Software and Hardware simulation time comparison

With the limited resources available on AIN Nodes, hardware acceleration is a key component to producing useful designs. With this in mind, it was necessary to select a set of design tools that included tight integration with Simulink®. This choice came with several other trade-offs in our favor, most notably Simulation and Design Verification. The trade-offs of using the MathWorks Simulink® & Xilinx System Generator approach are detailed in the following table:

Advantages:	**Disadvantages:**
High abstraction level modeling	Cost: both Simulink® and System Generator applications must be purchased.
System level schematic design with resource estimation.	Block level troubleshooting only.
Simulation is inherently rolled into each Simulink® Model design – Simulink® was originally built as a simulation tool.	Designs must use Xilinx the blocks which are included in Simulink®.
Verification is inherently rolled into each Simulink® Model design using the included "hardware-in-the-loop" process.	No bidirectional bus is supported

Table 5: Trade-offs of the Simulink® and System Generator DSP Design Approach

2.3 Implementation of IP Bocks

The process of compiling an FPGA design from human-readable HDL or schematic form into a functioning FPGA is called, "Implementation." Many tools for FPGA Implementation are available at prices ranging from free to tens of thousands of dollars from a variety of vendors, but they all follow the same basic steps:

1. **Translate:** Merge multiple design files into a single netlist
2. **Map:** Group logical symbols from the netlist (gates) into physical components (slices and IOBs)

3. **Place & Route:** Place components onto the chip, connect the components, and extract timing data into reports

The compilation process creates many output files, most notably including:

1. Logs (for debug)
2. Floor Plans for reference or manual optimization.
3. Bitstream the file that can be directly loaded into the FPGA.

Most FPGA vendors offer the *hardware compiler* layer of the design software tools for free, for example, Xilinx's Integrated Software Environment (ISE) synthesis program. Commercially available solutions such as Synplify and Precision promise faster compile times, highly optimized bitstreams, etc. Our example AIN Node design is compiled with Xilinx ISE because it is currently the only synthesis tool that is fully integrated with The MathWorks Simulink® software. ISE is a superset of System Generator that manages all of these steps in one application using the single click approach. This tool provides fine-grain visibility into the implementation process in order to provide the option of hand-optimizing any single step in the process. The ISE implementation process is broken down as follows:

1. Translate
2. Floorplan
3. Assign package pins
4. Map
5. Analyze timing
6. Place & Route
7. Analyze timing

8. Floorplan
9. FPGA Editor
10. Analyze power
11. Create simulation model

These steps are handled behind-the-scenes by Xilinx tools, which have been integrated with The MathWork tools to allow for a huge advancement in MDA design. Designers have an option to run Simulink® systems in software or FPGA hardware, with single-click ease of execution either way. This is because Simulink® software automated the controls to the System Generator making hardware compilation, allowing for very rapid experimentation and verification cycles (or design and experimentation loops) during DSP design. This terrific automation only works under certain connections to, and under direct control of, Simulink®. There are many more steps required to take a finished DSP design to full deployment in a product.

2.4 Hardware and Software Integration

The combination of DSP, hardware, and software IP cores is very complex, requiring the use of tools designed for embedded systems design. These are ***not*** available for free from any vendor; Xilinx was kind enough to donate EDK and System Generator to UNH for the implementation of the node named Global Network Academic Test (GNAT). The EDK design environment presents the designer with a single interface for connecting and configuring all IP Blocks to be used in the embedded system ensuring a consistent implementation of busses interconnecting various IP Blocks.

Pertinent information about IP Blocks are shared as necessary among blocks to allow a smooth end-to-end *system* compilation process as shown in the following Xilinx figure:

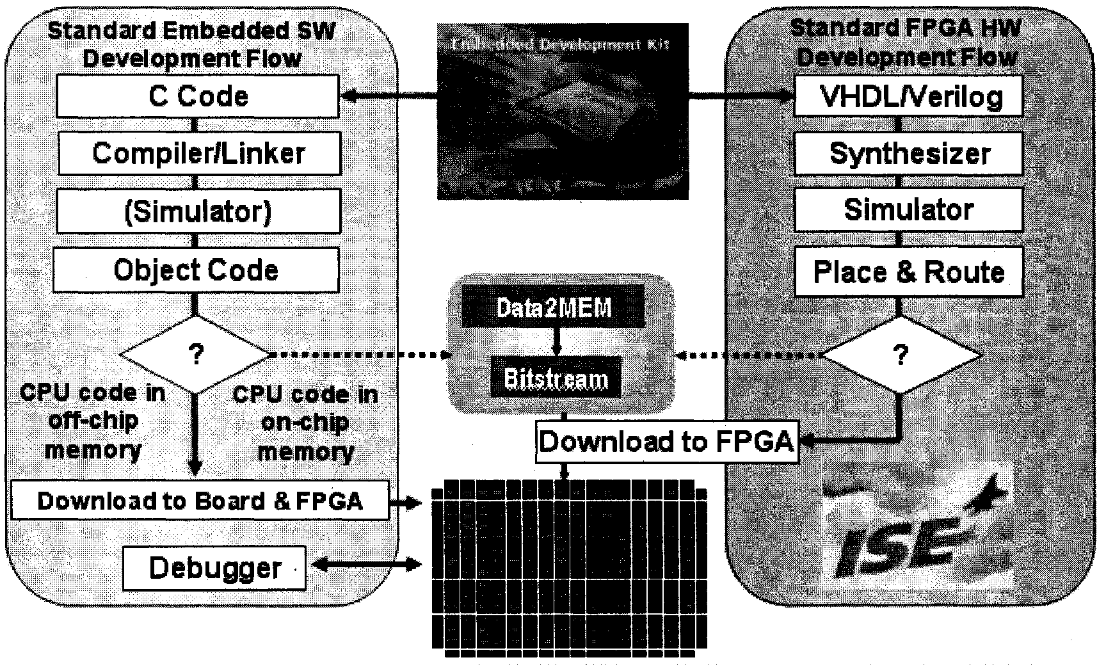


Figure 2: Xilinx EDK Flow Diagram

EDK recognizes and maintains software dependencies of hardware components, organizing software components of mixed-software/hardware IP Cores. EDK allows the hardware designer to easily interface hardware components, and feeds information about the resulting interconnect into the software compilation process. EDK manages source code and even kicks off software compilation, but the overall software solution is left completely up to the system designer.

2.5 Software

Gone are the days when a user interface could be implemented as a hardware state machine. Network-connected hardware must be able to communicate using complex, standards-based protocols, protocols that are occasionally updated, or patched. Some network appliances perform this function with a small dedicated application (App). Most, however, use an application that sits on a larger foundation of software called an Operating System (OS). While there are many choices in operating systems, and resulting tradeoffs such as architecture, cost, availability, and support, there is one universal advantage an OS has over an app. An Operating System includes software support for a very wide variety of common functions, such as networking, memory management, error handling, I/O, logging, encryption, and security. Since Software is a big part of the Node design, let us take a step back and look at the big picture for a moment before going into more detail about the software aspect of the GNAT design.

CHAPTER 3

Node Design

This chapter will present the steps to configure a simple node design using the methodology and tools previously described. The first step in the overview is a list of tools in the development environment. This list is not intended to scare potential designers away from this design methodology; the main thrust of this thesis is to provide a neatly packaged, well integrated environment. Like most embedded designs developed in industry, this project required expertise in too many specialties to be successfully completed by one person. The GNAT design environment, however, can be used by any systems engineer with elementary C programming experience to integrate sundry IP blocks into a complete system. Note that all shareware and GPL content (except for i386 RedHat 6.0) is packaged in the GNATDVD distribution.

Components of the Ambient Intelligence Node Design Environment are:

- MATLAB® 2006a R14.1, R14.2, or R14.3 with Signal Processing Blockset™ (Commercial Product)
- Xilinx v8.1 EDK (Commercial Product)
- Xilinx v8.1 ISE Foundation with latest service pack (Free download)
- Xilinx v8.1 System Generator for DSP
- sysgen2opb.m wrapper script from WARP/Rice University (Free download)
- EDK OPB Export Tool (Free download)
- MontaVista Linux 2.4 (Free download)
- XUP Virtex-II Pro board with Power Supply (Commercial Product)

- XUP Board Support Package from Digilent (Free download)
- Standard off-the-shelf USB cable (Commercial Product)
- 512 MB PC2100 SDRAM (Commercial Product)
- PC (Windows XP) (Commercial Products)
- PC (Linux, i386 RedHat 6.0 distribution recommended) (Commercial Hardware, Free download OS)
- 256MB Compact Flash (Commercial Product)
- Busy Box (Free download)
- Crosstool (Free download)
- MkRootFS (Free download)
- Tera Term Pro (Free download)
- Tcl interpreter (Free download)

This chapter will define the components of the GNAT, along with the design trade-offs involved in the chosen architecture. This will be followed by a brief overview of the steps involved in creating the Node. After that, the technical details and procedures for each step will be described. Many methods and tools were developed during the creation of the GNAT, and these will be detailed along the way in the order that they are needed in the process. Conforming to the global economy model in which AIN Nodes are expected to be designed, the GNAT was built using Common Off-The-Shelf (COTS) IP Blocks wherever possible. System designers who want to dive right in may want to skip ahead to APPENDIX VI: Ambient Intelligence Node Development Tools Set Quick Start Guide.

3.1 Node Architecture

The GNAT design is submitted as a reusable IP building block. Before drilling down into the details and tradeoffs made to reach this goal, let us first take a top-level overview of the Node:

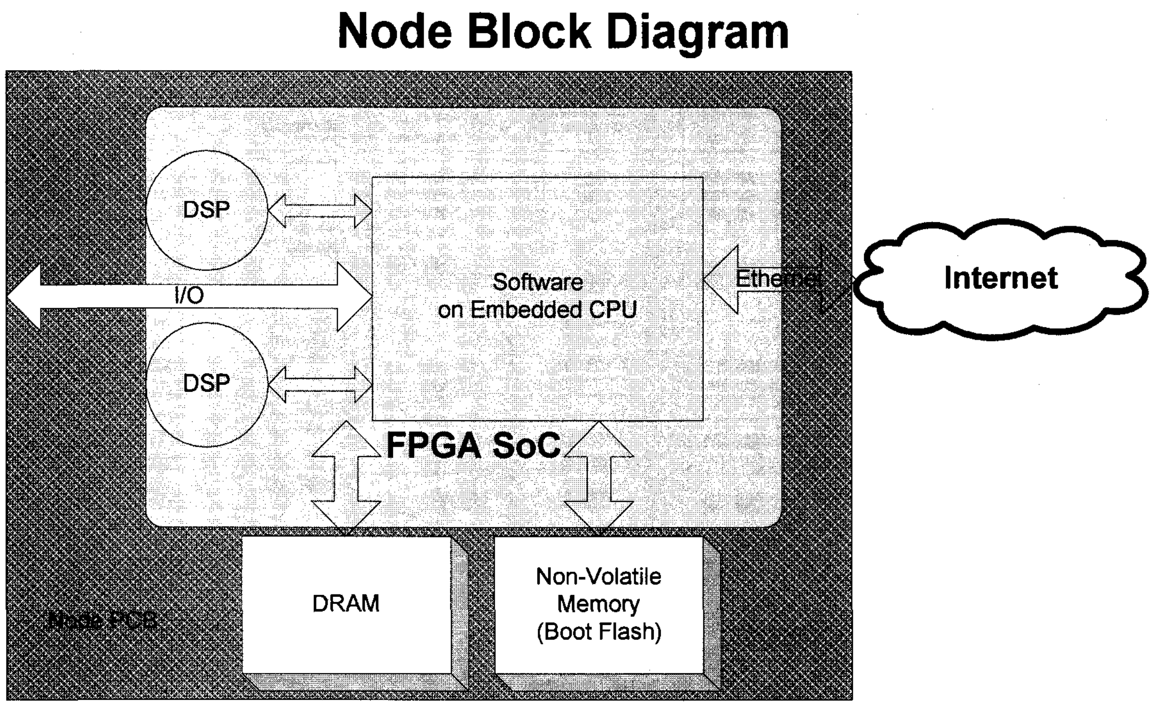


Figure 3: Node Block Diagram

The development process for this block was not focused on speed or power, but the design flexibility. The option to include one or more DSP cores synthesized as hardware in the FPGA gives this node design a much broader range of applications than a standard embedded computer. The inclusion of DSP IP cores brings with it the advantages of speed, power, and efficiency inherent to special purpose DSP. A modular means of interconnecting system components simplifies the design work and enables the designer to meet time-to-market demands. While hardware connections among the DSP core(s) would be more efficient than software, this type of interconnect carries a heavy customization burden for each system change. Software interconnect among system components is thus an excellent choice in reusable node design.

3.2 Node Software Overview

When it comes to placing software on an embedded system, there are two levels of abstraction available. A special-purpose application can often be compiled small enough to fit in FPGA memory blocks (BRAM on Xilinx processors), but the ultimate in pre-canned flexibility is provided by an Operating System (OS). Rather than code in or rewrite functionality in application code, one can reconfigure an operating system with simple configuration tools. Several embeddable OS's are available for designers:

- MontaVist Linux
- μClinux
- VxWORKS
- Xilinx OS
- Windows CE

MontaVista Linux 2.4 was selected for this design because it is available for free under General Public License (GPL). Linux comes with most of the networking support required for the GNAT node right out-of-the box:

- Network Hardware Driver with optimized interrupt handling
- TCP/IP stack
- DHCP/Routing

- Basic Web Server

While the embedded operating system provides much of the required functionality, there is still a role in the Node design best suited for an Application. Fundamental changes to the operating system must be compiled into the kernel, which involves a significant amount of risk and effort. The application space is better suited for transient or variable system components, as illustrated in the figure below;

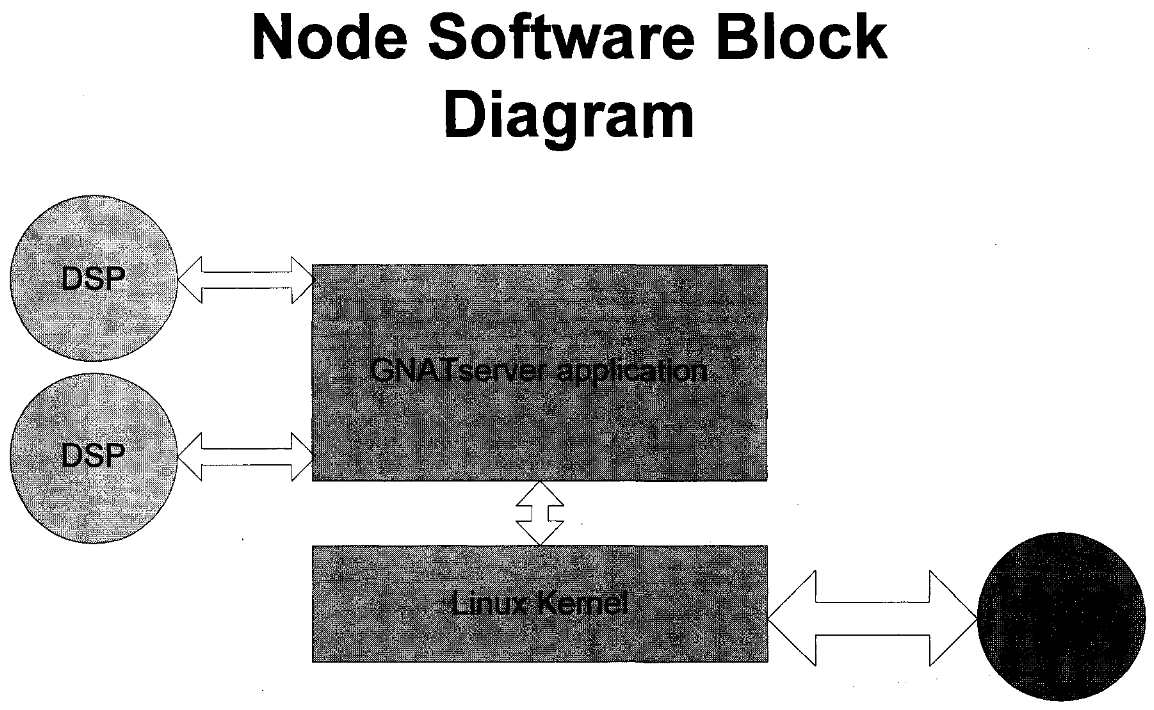


Figure 4: Node Software Block Diagram

Payload items, such as the DSP IP cores and the network connection, are represented in purple. The reconfigurable Server Application (blue) middleware manages communications among DSP cores and the network from a top its foundation, the static Linux Kernel (green). The relationship between the software and hardware components is depicted in the following block diagram:

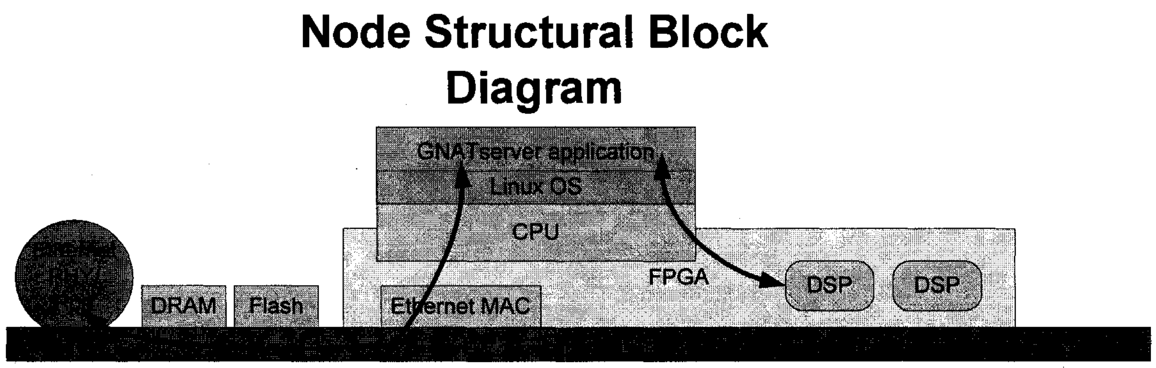
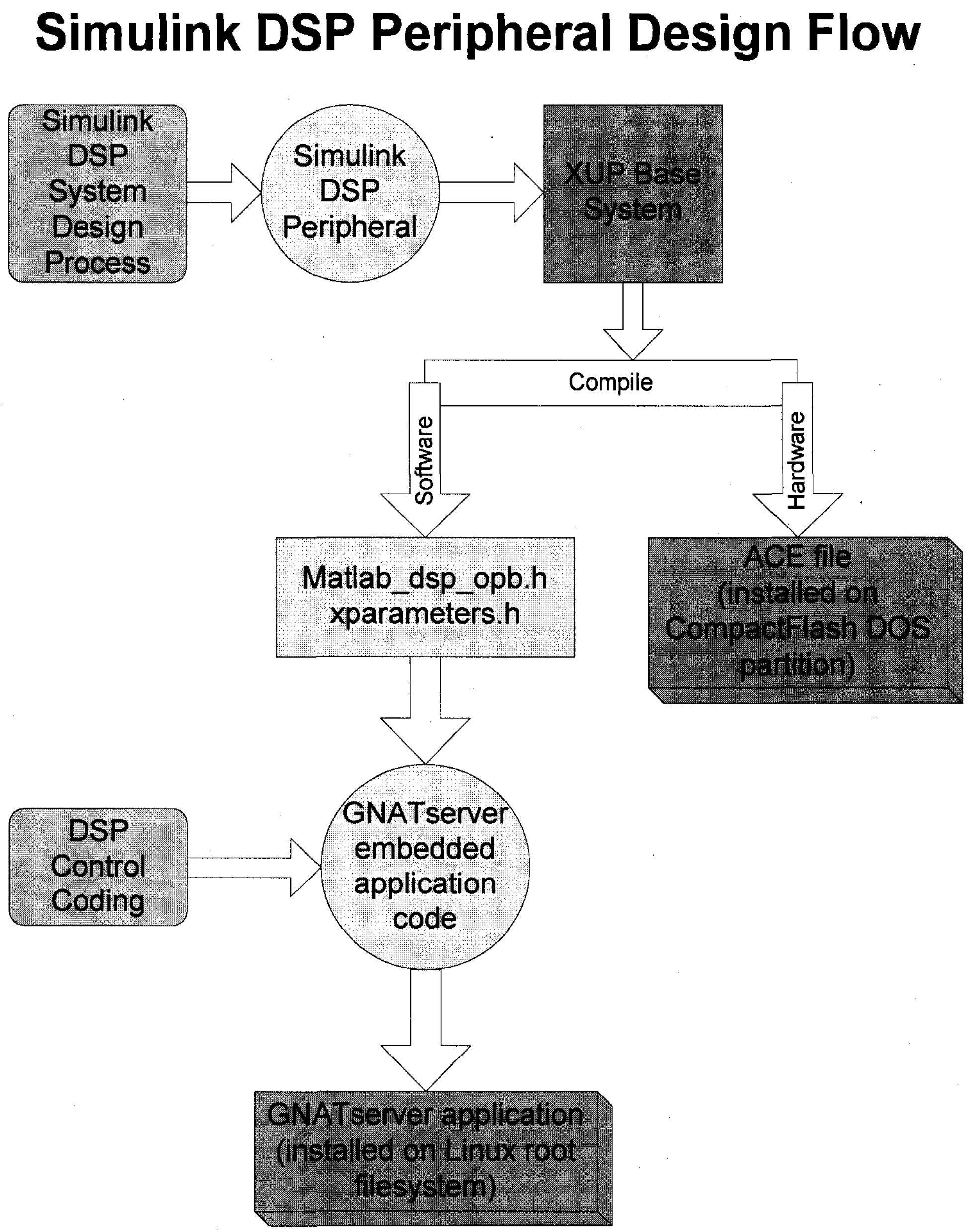


Figure 5: Node Structural Block Diagram

The GNATserver application resides on top of the Linux Operating System, which in turn executes on the CPU. The CPU is one of many cores in the FPGA; external dependencies such as DRAM and Flash out on the PCB are managed via the PLB bus by the Linux OS, and are excluded from this diagram for simplicity. An internet packet enters the GNAT via the Ethernet PHY, which puts signals on the PCB into the FPGA. The Ethernet MAC translates these signals into data on the OPB bus ready for consumption by the CPU, where it is interpreted by the Linux OS. GNAT requests are routed to the GNATServer application, which communicates back to the internet with relevant information from the DSPs. In order to communicate with the DSPs, the GNATServer application passes data to an abstracted interface in the Linux OS. As with Ethernet communications, the OS handles hardware off the CPU over the OPB bus to the DSPs.

The main concept of this architecture is it reusability. The design components that would normally be the most time labor intensive, DSP IP Blocks, can be done easily by any systems engineer via Xilinx's System Generator software in the Simulink® environment. These blocks can be swapped in and out based on the specific application, with little impact on the other system components. At this point it is worth repeating that the Simulink® -- System Generator environment provides excellent software and hardware DSP simulation capability as well as synthesis. This AIN design environment allows compilation of Simulink® models directly into standardized DSP Peripherals, with hooks into the processor at the hardware level and into the middleware application at the software level. This Simulink® DSP Peripheral design flow is illustrated below.



Simulink DSP Peripheral Design Flow
Simulink DSP System Design Process
Simulink DSP Peripheral
XUP Base System
Compile
Software
Hardware
Matlab_dsp_opb.h
xparameters.h
ACE file (installed on CompactFlash DOS partition)
DSP Control Coding
GNATserver embedded application code
GNATserver application (installed on Linux root filesystem)

Figure 6: Simulink® DSP Peripheral Design Flow

The design path begins with the DSP modeling in the Simulink® – System Generator environment. The model is first designed and simulated in native software within Simulink®. It can optionally be re-verified using the Simulink® Hardware-in-the-loop process before it is compiled for full deployment as an OPB Peripheral. From there, insertion of the Simulink® DSP Peripheral into the XUP Base System is a point-and-click process under EDK. Further automated processes produce updated hardware and middleware components. If the system designer wishes to make structural changes affecting the way that peripherals, in the system, communicate with each other, he does this by modifying the C source code for the GNATserver middleware. All of this compiled object code is installed on the flash drive, the installation of which into an XUP board makes a serviceable node. While remote updates to a networked Linux system are inherently simple, initial tests should always be performed on a local system before being deployed en masse via the network, to guard against unrecoverable failure

Chapter 4

DSP Block Design

DSP functionality is what makes the GNAT Node more than an ordinary embedded computer – more powerful, more efficient, and more flexible. As previously mentioned, our DSP IP Core design tool of choice is a combination of Xilinx System Generator and The MathWorks Simulink® software. A single click in this toolset translates a mathematical flow chart into VHDL code, which is piped into System Generator and then iMPACT tools to compile and upload a DSP core to the FPGA for iterative at-speed trials easily throughout the design process. Until recently, hand-crafted system interfaces were required to take Simulink® VHDL output to full deployment on an embedded system. Fortunately, new tools allow the finalized DSP design to be automatically wrapped into an OPB peripheral for integration of the DSP core into the microprocessor-based system solution. The finalized Simulink® model is then transformed into a Simulink® DSP Peripheral by a simple compilation process for full deployment!

4.1 DSP Connectivity Overview

The DSP functions designed in high-level Simulink® diagrams are compiled into FPGA hardware that lies alongside the CPU core in the same die. An IP Block provided by the Rice University WARP Project[3] presents the DSP hardware interface to the CPU as a set of registers on the OPB bus with compile-time configurable options for hardware pipelining. This DSP Peripheral is addressed from software running on the CPU; either an OS driver or a userland app. Either way, the software routes signals as desired for the given system. In the GNAT, the userland server app that provides Web Services also handles routing of signals to and from (or among) DSP elements.

4.1 DSP Block Design Detail

The steps in the Simulink® DSP Peripheral design process include:

- DSP schematics design with Simulation
- Hardware model generation
- Project verification (Hardware in the Loop testing), resource estimation/tune-up
- OPB-compliant peripheral generation
- Integration of Simulink® DSP Peripheral into EDK project

It is worth pointing out that the first step in this process is a universal design not dependent on the targeted board. Of course, the toolset provided here does add the requirements that the target hardware includes a Xilinx FPGA and embedded CPU. The first steps require knowledge of Digital Control Systems. For simplicity, the GNAT implements a 32-bit Multiplier DSP block.

The most important parts in the Simulink® Xilinx block-set are the "Gateway In" and "Gateway Out" blocks. These specify the bounds of the design that fall within the realm of the Xilinx System Generator. An extensive set of blocks developed by Xilinx can be used inside these bounds and implemented into the FPGA; analog environmental design tools like spectrum scopes and signal sources are ordinary Simulink® blocks that cannot be compiled into the FPGA. Once a model is designed, simulated, and operating satisfactorily in Simulink® software, the next step is to translate this model into a hardware implementation. This process requires several steps like synthesis, place & route, etc. These steps are managed and performed by Xilinx ISE, and triggered by the System Generator block. In order to optimize the output, it is important at this stage of implementation to define the hardware target. Simulink® ensures timely selection of a compile target by the structure of the System Generator block controls. Before System Generator will compile, all relevant fields of the following dialog must be filled in:

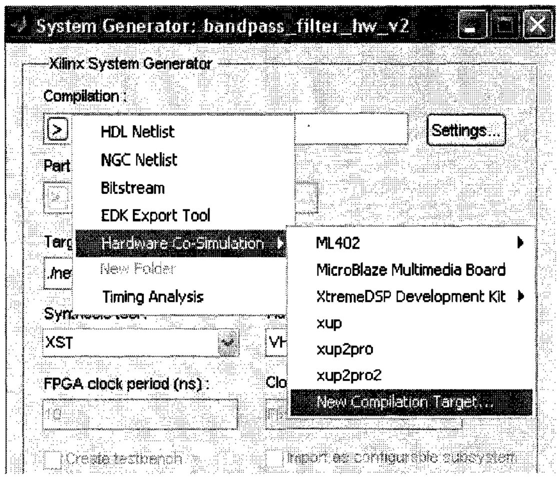


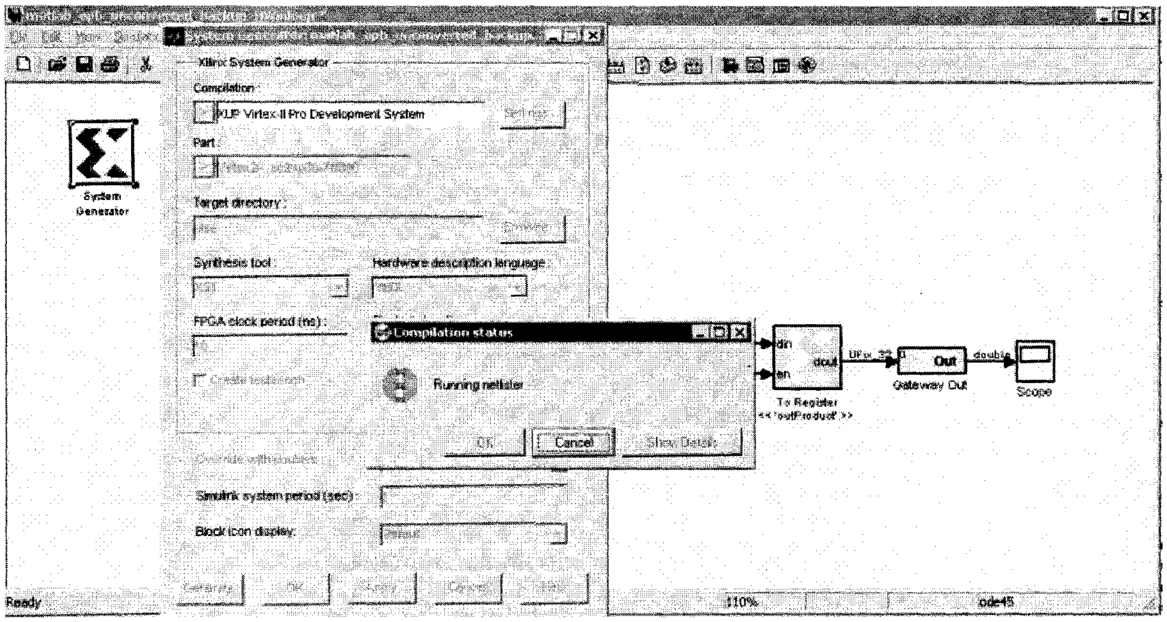
Figure 7: Xilinx System Generator Setup

The next logical step after software simulation is Hardware Co-Simulation. If a Board Support Package (BSP) is not loaded, then a new board can be defined by selecting $\rightarrow$ Co-Simulation $\rightarrow$ New Compilation Target. Several parameters need to be provided by the board designer:

- Core clock frequency in MHz
- Pin Allocation
- The name of the board and type

- Boundary Scan Position

The JTAG options are automatically detected from the board. Note that the board must be connected to and detected by the PC via "Xilinx JTAG Debug" USB cable before the Simulink® software is opened. With all of these parameters defined, hardware compilation is triggered from this System Generator block menu with single click of the Generate button as shown below:


Xilinx System Generator
Compilation: XUP Virtex-II Pro Development System
Part:
Target directory:
Synthesis tool:
Hardware description language:
FPGA clock period (ns):
Create testbench
Override with doubles
Simulink system period (sec):
Block icon display:

Compilation status
Running netlister

Figure 8: Xilinx System Generator Running Netlister

The compilation steps include:

- Synthesis Options Summary
- HDL Compilation
- Design Hierarchy Analysis
- HDL Analysis
- HDL Synthesis
- HDL Synthesis Report

- Advanced HDL Synthesis
- Advanced HDL Synthesis Report
- Low Level Synthesis
- Partition Report
- Final Report
- Device utilization summary
- TIMING REPORT

Design Verification Test (DVT) is very easy in this environment because the software and hardware implementations of the same model can be against the same Simulink® source, and the outputs can be compared on Simulink® scopes and spectrum analyzers.

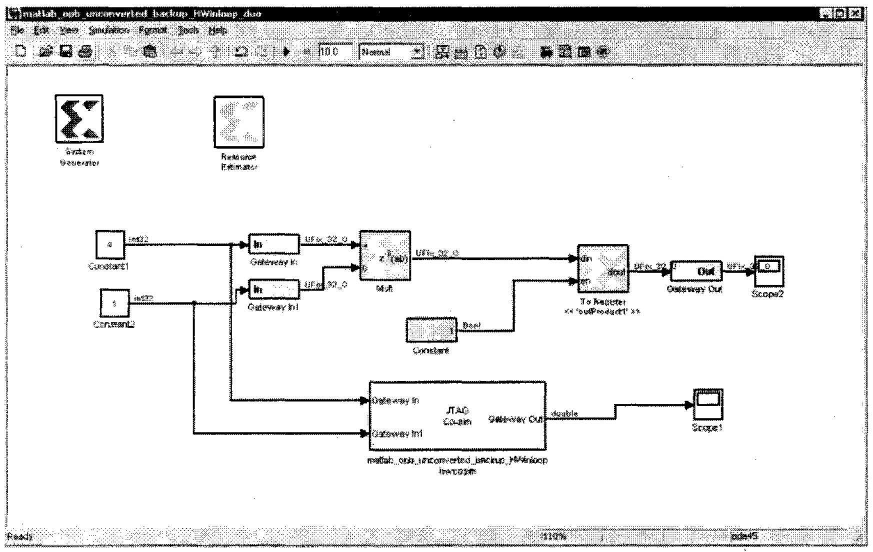
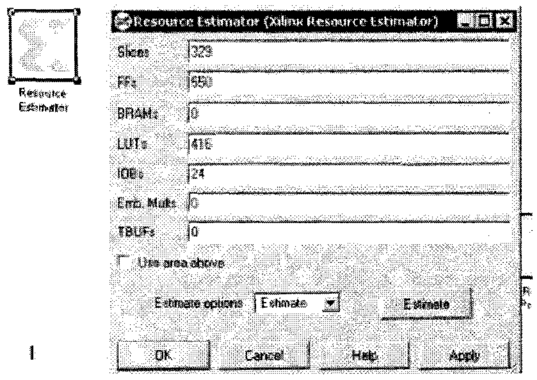


Figure 9: Simulink® software – Hardware in the Loop testing

Another very useful feature of the System Generator block is the Resource Estimator, which determines the physical FPGA resources required by the model.



Slices	329
FFs	550
BRAMs	0
LUTs	416
IOBs	24
Emb. Mults	0
TBUFs	0

Use area above

Estimate options: Estimate

Figure 10: Xilinx System Generator – Resource Estimator

When the design is completed and verified, the next step is to wrap the model in an OPB peripheral structure. In order to connect the IP Block to the GNAT CPU, the block must be an OPB-bus compatible peripheral. While Xilinx's OPB Export Tool provides fine-grain control of the OPB Bus Interface from the Simulink® software, it requires the additional purchase of The MathWorks Stateflow® tools. In order to overcome this limitation, a MATLAB® script provided by the Rice University WARP Project was used to meet this OPB structure needs; this script is "sysgen2opb.m"[5]. The exact steps required to use this tool are demonstrated in "WARP Lab2: Introduction to sysgen2opb"[4]. This script is used to further abstract the Simulink® DSP Block before it is processed by Xilinx's OPB Export tool, eliminating both fine-grain control of the OPB Bus Interface and the need for Stateflow® tools. When the new peripheral appears in the EDK project the last hardware steps are to attach the Simulink® DSP Peripheral to the OPB bus and generate addresses.

4.2 DSP Connectivity Detail

Several cores that come with an OPB interface were found freely available from The MathWorks, Xilinx, and other sources. The OPB interface in the Xilinx EDK toolset is based on an IBM processor bus standard, and allows IP cores to present themselves as hardware peripherals to the processor (whether that be a hard-core PPC or a soft-core MicroBlaze). Each of these peripherals presents itself primarily as a range of addresses in memory, and includes sample application code.

The key aspect of embedded system design that makes an AIN node most useful is network connectivity to the DSP core(s). In the GNAT Node the cores are managed by the very same application that handles network communications, the *gnatserver* – a SOAP server that handles data-structure transmission over the internet. This application-level code does not quite match up with the sample code provided with each IP core, as the cores come with sample Real-Time Operating System (RTOS) Applications. These were ported into the *gnatserver* application using the methods described in an article titled, "Porting RTOS Device Drivers to Embedded Linux" [5]. Unlike RTOS environments, Linux has a memory manager, which abstracts physical memory into multiple virtual memory address ranges. The memory manager resides in the kernel, and maintains **exclusive** access to all physical addressing ranges. The Linux OS is architected this way for several reasons, including security and scalability. The physical addresses of the Simulink® DSP Peripheral in the system enumerated in "xparameters.h" are mapped into virtual addresses in the space of the calling application by the Linux "*mmap*" system call. Register read and write functions required to access OPB peripherals are provided in Xilinx's sample memory test program for the XUP board. The relevant source files have been copied from *ppc405\libsrc\cpu_ppc405_v1_00_a\src* and *ppc405\libsrc\common_v1_00_a\src* without modification into the *gnatserver* source directory[6].

As is true at every other level of this system's architecture, there are trade-offs involved in the selection of this peripheral management structure. The application-level support for the Simulink® DSP Peripheral provides the developer with the greatest possible flexibility. The kernel does not need to be recompiled nor packed into a new ACE file when a functional change in data routing is desired. Said changes can instead be made in one C file and compiled into the top-level application. The drawback to application-level hardware support is that it precludes the use of hardware interrupts. The Linux architecture does not provide interrupt access to the user application space; instead all interrupt drivers must be installed in the kernel. However, the selected Simulink® OPB peripheral compiler currently does not support interrupts, so a kernel-level peripheral driver is not necessary at this time.

The Simulink® DSP Peripheral continuously operates on the input registers, independent of the application software, operating system, or CPU. Input registers are processed into a result in a fixed amount of real time, regardless of the operands – this is a feature of hardware execution. In this example case, the DSP is faster than the software code, so no software delays are necessary. The software reads the output register immediately after writing the input registers, and the result is ready.

Chapter 5

Node Software

The details of the node design span numerous design disciplines. As previously mentioned, an operating system was selected to manage the network interface. The entire operating system must be cross-compiled targeting the Node CPU, with kernel components linked according to the system hardware. While Xilinx tools do produce the bulk of the configuration inputs necessary to generate said kernel, their investment into the Linux OS compilation process is evidently minimal. The baseline configuration of FPGA hardware, Linux Kernel, and OS file system needed for a node *without* a Simulink® DSP Peripheral are described in a recent publication titled, "Porting MontaVista Linux to the XUP Virtex-II Pro Development Board"[7]. It was very difficult to replicate this work due to the breadth and complexity of the tools, so a different approach was taken to packaging the IP in this thesis. The FPGA portion of the Sample Node IP core is submitted as a pre-packaged, already working EDK project. The software portion of this sample IP core is submitted as both source and object code packaged along with a complete, configured cross-compiler environment. A copy of this set of cores (1.1GB) is in UNH's IP core repository.

5.1 Operating System

Several automation tools were developed to streamline the process of getting Linux onto an XUPV2P board. These are all run from a Linux PC that is set up with the cross-compiler environment included in the GNATDVD. The first of these tools is one which embeds a selected MAC addresses into the Linux Kernel. Unfortunately, the XUPV2P board does not come with a hardware MAC address; this must instead be compiled into the kernel in order for the XUP board to talk on a standard Ethernet network. The "setMAC.tcl" script in the Linux Compile Server's *xupv2p* directory updates the network driver to use a hard-coded value specified by the operator in the command-line parameter [MAC ADDRESS].

The kernel contains all of the hardware drivers for the board, but EDK was designed with this requirement in mind. Any time the hardware is reconfigured and compiled, EDK updates the contents of the Board Support Package (BSP) directory. The Linux Kernel compiler expects to find this directory under the main *montaVista_2_4_devel* directory, and uses the hardware configuration information and driver code contained therein. EDK can be instructed to place this directory on a network file share so that this occurs transparently. In practice it was simpler to do this step manually, in order to prevent unexpected changes to the kernel source code. In order to put these hardware changes into effect, the kernel must be recompiled.

Another automation tool was written to expedite kernel compiles. "*updatekernel.sh*" can be run from the Linux Compile Server's *xupv2p* directory with no command-line arguments. This script combines the static parameters and steps of the compile command, preventing mistakes. The finished kernel file is found under a soft link at "*xupv2p\zImage.elf*" on the Linux machine. This kernel must be further compiled along with the FPGA bitstream into an ACE file by EDK. Once again, contentious use of network file shares can make this step transparent to the user, but a manual process prevents unintentional changes. While "*zImage.elf*" is the actual binary executable, it cannot run on the FPGA in this form. A script called "*genACE.sh*" is run from the EDK command line to compile the kernel "*xupv2p\montaVista_ELF\zImage.elf*" and the latest EDK bitstream, "*implementation\system.bit*" into "*xupv2p\montaVista_ACE\system.ACE*". This file, once copied to the DOS partition on the CompactFlash drive, is used by the systemACE chip on the xupv2p board to load the Virtex FPGA bitstream and boot the embedded PowerPC Processor cores to the enclosed Linux kernel.

5.2 Application Software

The application layer of network communications in the GNAT Node is handled by custom middleware, an IP Block called GNATserver. This application formats and encapsulates data according to Service Oriented Architecture Protocol (SOAP) standards so that bidirectional communications across the internet can be conducted in a platform-independent manner, with data values interpreted correctly on all sides. The gSOAP C/C++ Web Services and Clients Toolkit[8] was selected as the development tool in which to write this network application. The GNATServer IP Block is a small C++ program that can be easily configured by anyone familiar with the C programming language. This program is compiled into a W3C compliant, multithreaded gSOAP server using the gSoap open-source toolkit. The gSOAP toolkit itself is easy to build and run from both Linux and Windows, and the source code generated cross-compiles easily. The version of the GNATserver code discussed in this chapter can be found in APPENDIX III: GNAT Node (Server) Demo Code.

The GNATserver application contains the software interconnect that binds the components of the GNAT Node together. This is achieved using automatically generated output of EDK, starting with a file called "*xparameters.h*", as previously mentioned. This file defines hardware-related values such as XPAR_MATLAB_OPB_OPBW_0_HIGHADDR and XPAR_MATLAB_OPB_OPBW_0_BASEADDR, the high and low addresses of the first DSP core. EDK generates this file into "*XUPV2P\ppc405_0\include*", and it is needed in "*xupv2p/mkrootfs/gnat*" (the GNAT source directory) on the Linux compile server – in this case, manually copying the file is required. These addresses are virtualized by the Linux Kernel by the mmap system call on line 21 of *gnatserver.c* because Linux does not permit direct access to physical memory from userland programs. The mmap register access method is the simplest way to access register-driven peripherals. For high-speed data links such as streaming video, an interrupt service routine or direct memory access driver may be preferable. This would require a kernel-level Linux driver, and would be an excellent addition to the GNAT IP Library.

The Simulink® OPB IP Blocks generated with the WARP MATLAB® OPB Peripheral generator include C header files defining data types and I/O routines for communicating with the hardware devices at the lowest levels. This is the reason for the inclusion of the header file on line 5 of *gnatserver.c: "matlab_opb.h"*. "*XUPV2P\drivers\matlab_opb_opbw\src\*" is the location in the EDK project from which this file was drawn. Additional source code referenced in this header file, provided by

Xilinx, defines register access routines required to talk to these peripherals. As described in Section 4.3, DSP Connectivity Detail, these dependencies have already been placed in the Linux "*xupv2p/mkrootfs/gnat*" soap server compilation directory. The final 20 lines of the main() function in *gnatserver.c* start the SOAP web service, listening on the port specified at the command-line (port 8080 is used in the GNAT Demo configuration).

The SOAP service within gnatserver manages bidirectional data flow between the network and the Simulink® DSP. Such data transactions could take many forms; the SOAP architecture allows the developer to specify an arbitrarily complex (or simple) data structure for a given transaction, as well as an arbitrary number of transaction types. These transactions and data structures are defined as SOAP services in *gnat.h*, shown here in APPENDIX III: GNAT Node (Server) Demo Code. In this example, four inputs and one output are defined.

The *gnatserver.c* main code orchestrates this transaction in the *ns_gnatevent* function on line 52. The two datapoints, *a* and *b*n are fed into the input registers of the DSP, and the result is read out immediately. The program then compares the result to one of the control values, *c*, which represents a threshold of relevance. If the result is determined to be relevant (greater than the threshold value), it is reported to another computer on the network, according to the Universal Reference Locator (URL) supplied in the fourth parameter. The data is formatted in the same *ns_gnatevent* SOAP type so that it can be received by another node, which for the demonstration, was a PC that displayed incoming events to the crowd.

The *gnatclient* program allows a user or another program to interact with any gnat node over a network from a PC command line. This short program may be found in

APPENDIX IV: GNAT Client Source Code. It takes 5 parameters: Node URL, operand A, operand B, threshold C, and forwarding URL. The reason the client and server programs are so short and simple is that all of the code dealing with network sockets, multithreaded servers, and SOAP protocol is automatically generated by the gSOAP precompiler. All of this is done based on the transaction structure described in gnat.h (one line of code), allowing the developer to focus on architecting his node's communication structure in the event function(s) of "*gnatserver.c*".

The example *gnatserver* DSP core performs a simple multiplier function. While there are three DSP cores implemented in this sample, only the first is used, as this is not intended as a demonstration of distributed computing. While a node could be used as network-attached DSP Appliance such as a wire-speed MPEG CODEC or encryption/decryption engine, it is expected that the more common application will be for multiple sensors and/or signal outputs to be interconnected, with reporting to, or remote control and observation from the network.

The *gnatserver* application is organized in the Node IP package such that it is recompiled and installed along with the BusyBox shell in to the Node's flash filesystem (a non-DOS partition) along with the non-kernel portion of the Linux OS by the "*mkrootfs.sh*" script. This script was originally provided by Jonathon Donaldson as part of the Linux Baseline Configuration toolset, but was expanded to compile and install the GNATserver. This script must run from "*xupv2p*" on the Linux Compile Server **while the Node's CompactFlash card is connected**. It is important to note that the *gnatserver* is run on the node at boot-time due to its configuration as a service as specified in "*xupv2p\mkrootfs\etc\init.d\rcS*". It is important to note that ALL FPGA, DSP, Software, and Configuration components to the Node are stored on the CompactFlash card. Repurposing a Node is as easy as changing the flash card, either by physically swapping out the card, or by updating the files over the network.

A demonstration of the GNAT Node was performed at UNH's December 2007 ECE Final Design Review. In this demonstration, three Nodes were networked together – all three had identical Hardware and Software loads, but unique network addresses. A Linux GNATclient program called by a script streamed control and simulated sensor data to all three nodes for processing. The data structure for this demonstration is as described above:

1. Operand A – a 32-bit integer
2. Operand B – a 32-bit integer
3. Threshold C – a 32-bit integer
4. Report target Universal Reference Locator (URL) – a string indicating the internet address to which relevant results should be delivered by the processing Node.

Each node multiplied operands A and B, then compared the product to C. Any time the product of A and B was greater than C, it was deemed "important" and forwarded to the URL in the first parameter. For the demonstration, a Monitoring Console consisting of a GNATServer running on a Windows PC was used to receive and view this data. Expansion of the node to this level from a simpler operand-operand-result data structure and construction of this multi-node demo was completed with only one man-day of work!

CHAPTER 6

DEVELOPMENT ROADMAP

Given a working collection of nodes, there are two axes along which the nodes may be combined. The first axis is networking, and the networking standard chosen for the GNAT Node is part of a higher level framework invented for organizing such devices. The significance of this framework will be described below. Completely orthogonal to the networking axis is the adaptability axis. Field-reconfigurable nodes come with great potential, but the means to fully exploit this potential lie in automated test methods. An overview of the automated test methods for expansion along this axis is also described below.

6.1 Self-Organization of Node-Based Systems

In order for a collection of nodes to form a useful system, there must be a purpose. Purposes are best distributed by hierarchical systems, so the problem of organization falls to how a hierarchy can be built *autonomously* from an ad-hoc collection of nodes. It would be desirable to have the node at the top of the hierarchy be one that is particularly well suited to management functions. Desirable features might be a large network bandwidth, a central point of communications, proximity to an operator who defines and changes purpose, or even a weighted combination of factors. None of these factors can be determined, however, until order is established.

The first order of business for a node is to establish communications with its neighbors. Whether this is a mesh network, LAN, etc. there are standard protocols such as Dynamic Host Configuration Protocol (DHCP) for requesting a position on the network. What if a given node is the first on the network? Nodes with the capacity to do so can be given a default behavior to set themselves up as DHCP servers on a default network. As communications are established, nodes register their addresses on the network as they are assigned. Following that step, the next order of business is for each node to forward a description of its capabilities and services via Web Service Description Language (WSDL) to a central point. So far, the only central point is the DHCP server, so it must have a Universal Description, Discovery and Integration (UDDI) service as well, to catalog each entry. All nodes advertising leadership capacity will be eligible for an election (randomized) to process the WSDL data collected by the UDDI in search of a purpose. Until a purpose is discovered, each node will assume its default behavior (if defined).

Once a purpose has been determined, the management node can remain manager or elect another node better suited to managing the task at hand. Once proper management has been established, the management node can begin assigning tasks and configurations to each node on the network, instructing nodes as to their interrelationships to form an organized system. For example, if one of the purposes of the system is to detect the epicenter of an earthquake, all nodes with vibration sensors can be synchronized, and given appropriate frequency filters. Central nodes on the network with the proper processing or DSP capacity are then instructed to subscribe to earthquake notification services on the detection nodes using SOAP messages. Said central points can change out a load of DSP firmware if necessary, or simply run a trig program on the CPU to perform echolocation on the data will be forwarded immediately to them on detection of earthquake events. These central points must, of course, also be instructed on what to do with their findings.

This structure is by its self-forming nature, a self-healing (fault-tolerant) structure. If the management node fails, no one will notice (not even a human operator) until a new purpose is raised. Any time a purpose exists without a management node, it causes an election. The system reforms as necessary. If any peripheral nodes are taken out of service cleanly, they will communicate this up the chain to the management node, who will compensate. If any node should fail unexpectedly, it will be noticed next time another node tries to interact with it, and this will be communicated up the chain. The management node will again compensate as needed. In the case that a new node with superior management capabilities is introduced to the system, that too will trigger an election.

In order to prevent all nodes from conglomerating into a single monster system, the notion of ownership must be instantiated. This can be done politely (naively) by assigning a system name to each node before it is deployed. Of course, a diabolical management program could easily spoof the advertised name to repurpose the node. For most applications, strong authentication will be required – likely in combination with data encryption over any channels not established as "trusted networks". A variety of software IP Blocks are freely available for the Linux OS to tackle these issues. How will the management node determine how to best carry out the assigned purpose using the available system of nodes? This matter is up to the designer who defines the purpose.

Using the design methodology contained demonstrated here, the designer gets to solve the same problems as before. The difference is that his toolbox now contains an army of nodes rather than an array of gates.

6.2 Autonomous System Test Specification for Node-Based Design

This is a test specification for AIN systems such as the GNAT node. Such a system is comprised of standardized *nodes*, each of which includes processing and communication. In addition, each node has attached zero or more other *elements*, such as sensors or storage. Nodes are expected to be interconnected directly with hardwired connections and/or indirectly via wireless networks. Wireless protocols such as the ZigBee Mesh network standard (extensions of IEEE 802.15.4) already handle seamless adding and removing of nodes. With the wireless aspect of the system continuity handled by software, that leaves quite a bit of hardware that must still be tested.

One motto of the PCB test camp has long been, "If the components and all of the interconnects among them are good, then the board is good." This board-test principle of compositionality is applicable at the system-level as well, even when the system spans a continent. This means that in order to have confidence in a distributed system, one must be able to test all nodes and physical interconnects – including IP Blocks, node-to-node interconnects, and node-to-element interconnects. Such tests permit us to avoid reliance on malfunctioning circuits, as well as opportunities to redefine non-functioning parts of the system. It is important to bear in mind that it is not acceptable to simple deploy known-good (factory tested) components. Permanently deployed nodes must be able to periodically test their subsystems to determine when, not if, degradation has occurred.

6.2.1 Element Test:

Let us first examine the area of element test. Elements are attached to nodes, which contain, among other functions, computing power. In order to operate the elements in a useful manner, the nodes must already have some knowledge of how to use them. For a hardware element example, a thermistor is connected to an A/D port on a node. Each hardware element must contain a software interface, such as a definition of valid sampling rates, and a method to convert the sampled signal into a temperature value for use elsewhere in the system. If still more knowledge is added about the thermistor to its IP core, it can run a Design Verification Test (DVT). This kind of test can be run against any given element to produce a result that, when compared against a signature, provides a high probability that the element is working as intended in the system. The A/D block can be exercised using BOLBO to produce a signature used to verify that the required timing was instantiated correctly on the FPGA. Once confidence is established in the A/D, a simple thermistor read will yield a value, which should fall within a range known to be valid for said sensor. Calibration is the next step, and may be performed by applying a standard current to the thermistor and looking for an expected temperature increase. A variation from this test response in a new device may indicate that a correction factor must be applied to account for component or environmental variation. The specifications of the sensor can be coded into a test rule and response table that can be applied to the Node to maintain a known operating characteristic and a definitive end of the element's operating life. Ideally, each element should contain the circuits necessary to run such self-tests so as to present a standard interface to the node. This way, the element is operated only during its useful life, and is discarded or flagged for repair when its lifetime is over.

How can the node know so much about each element in a dynamically configurable environment? Any IP instantiated separately from its test patterns must carry a sufficient identifier so that the proper steps to establish trust can be pulled from a database. Make, model, and revision numbers are sufficient to match hardware IP with its software counterpart, but a unique Serial number is required to allow an element's calibration record follow it when it moves in a system. For elements with static connections to nodes, any required history can be stored on or related to the attached node. However, any elements that move among nodes must present identifiers in a standard fashion on whatever interface the element uses to connect to the node (SPI, USB, parallel). This is generally done by commercial PC components, in compliance to several Plug-and-Play (PnP) standards, and is already being standardized in the sensor industry under IEEE 1451. The required codes are presented to the interface as part of the element's power-up initialization sequence, causing the attached node to load the correct software IP (hardware drivers and applications). It is already common practice for nodes to forward any un-recognized codes to a storage or command node to retrieve the necessary IP to utilize each attached element, with the most prominent example of this being Microsoft Windows Update. At any level of the system that uses TCP/IP, the best practice is that such transactions take place in the XML self-documenting file format.

6.2.2 Node Test:

The next area of focus is the node test. Because nodes can operate independently (without physical connection to anything else in the system), they should implement BIST (Built-In Self-Test). JTAG-based BIST techniques assume a test-master, which can drive vectors through a system (one or more chips) in test mode. Test masters, such as IEEE1149.6 MTM controllers, can often run vectors and responses against a store in flash memory, or across a system-wide test bus. Since our nodes are pressing towards single-chip solutions, it may be necessary to invert this methodology in order to accomplish our task. Rather than running functional test vectors which must be manually created for each IP core, the FPGA can reboot into a second, test-only, IP core to run structural test vectors. This core would instantiate the IEEE 1149.1 TAP controller (generally available from the device's standard library) and an internal test master, with vectors loaded as data into FPGA memory areas. During SoC design, board netlist and BSDL(s) for the node PCB are compiled into a test vector file using standard JTAG test tools, with element ports masked off. This file is incorporated as data into the IP compilation using standard FPGA tools. The test master has a "hardware" component allowing the onboard processor (or processor soft-core) to take over the TAP controller (minor modifications to the off-the-shelf TAP controller IP may be required to allow these connections). The test master also includes a software component that runs on the embedded core, processing the test vectors and translating any errors into flags stored for later use by the FPGA's functional IP core. The software portion of the Test Master consists primarily of IP commercially available on the market today.

When the Node comes up in BIST mode, it may read flags from a non-volatile register to determine what level (or flavor) of tests to run. However, it is expected that it will run primarily JTAG vectors. What does this accomplish? In infrastructure test verifies correct operation of board's test bus, ensuring valid results from any tests performed using this bus. An interconnect test then verifies the connections on the board are present (and that no unwanted connections are present). Paths between physically interconnected nodes can be tested using the board-to-board interconnect standard. Failing vectors can be run against diagnostics software, commercially available from the same companies that make the embedded vector generators. These diagnostics may optionally be categorized as recoverable faults if there are alternative IP configurations that do not use the damaged portion of the Node. Non-recoverable failures indicate an end-of-life for the node.

The easiest method for implementing a clean EOL process in a dispersed system is to provide a known timeout duration for the BIST for each variety of node. When a node indicates (or is told to) that it will go out of service for a BIST, its failure to recover from that BIST can be the indicator that it encountered a non-recoverable error. With this in mind, the BIST can be designed to simply store any failure data for use by a technician and power down. Recoverable errors can be assigned by the designer along with accompanying automated repair algorithms, described in another document. Any status flags generated during JTAG BIST should be stored in an area of the node that will not be erased during loading of the functional IP. This could be a write-protected sector of the FPGA or an off-chip memory.

The bootstrap code on the functional IP should read the test-flag memory on each boot up in order to determine appropriate action. In positive cases, a successful interconnect test may be followed by a memory test that is best executed from software on the functional IP core, for example. Alternatively, a failure in a node-to-node parallel cable connection may result in a flag indicating that a serial interface IP Block must be loaded instead of the preferred parallel-interface IP Block. For FPGAs that do not support partial reloads, this may result in a request for retrieval of alternate IP images from a storage node. Furthermore, certain error flags must be communicated upstream to the remainder of the system. The serial-fail-back from a parallel interface, for example, must be communicated to the neighboring node in order to restore connectivity – whether through a timeout or through an alternate interface. Also, any reduced functionality shall be communicated upstream (if any interfaces are available) so that the system can account for available resources and contact repair personnel as necessary.

Once trust is established in the node hardware, the next step is to validate any new combinations of instantiated IP on said node. If the particular IP combination of hardware, software, and firmware have been previously verified, DVT is not required. When deploying a known good design on a known bad board, however, a bit of functional test remains prudent. In order for either DFT or Functional test to be possible, each IP Block must include self-test capabilities, such as a BOLBO signature and/or a smattering of sample inputs & outputs. On success, the Node can report its new functionality back to the system and receive a suitable assignment. At the discretion of the Management Node, the system may email a technician with a request for replacement parts, or a mobile component may even drive itself to the closest repair depot.

CHAPTER 7

GNAT System Evaluation

7.1 Results

The achievement in the GNAT Node – an internet-enabled hardware multiplier – took over a year to develop. In the commercial market, this would have been an utter failure. The goal of this design, however, was not to bring a product to market and yield a commercial success. The final product yielded by this academic research is not the GNAT Node, but the design methodology and development environment built along the way. The GNAT Node itself is simply a proof of concept. A hardware design (Simulink® DSP) was integrated onto an existing hardware platform with accompanying interface software using a Model Driven Architecture. The IP Blocks used in this design were selected according to the design methodology and integrated into a product effectively with minimal design effort.

This design methodology provides a simple software-based interconnect framework for IP Blocks. Not only may IP Blocks on the same Node intercommunicate, but IP Blocks on disparate nodes may also communicate with each other or end-user applications across the internet. The tools environment assembled allows arbitrary DSP functions to be implemented and interconnected with controlling logic and network connectivity with ease. This tools-set allows for the efficient use of design time necessary to develop high-level AIN systems, and should be considered the absolute minimum design environment capable of producing AIN Node products.

The following tools required to complete the set did not exist in the commercial marketplace, and were created for this effort:

- Coherent FPGA and Software Development Environment for Linux on Virtex
- GNAT Client/Server SOAP-Standard Web-to-DSP interface and management application

This research effort was a tremendous success, yielding a commercially viable design environment and methodology that can be used to churn out new products at an astounding rate.

7.2 Identified Gaps

The GNAT Node demonstrated in this project lacked any physical-world interface. Most commercial applications for Ambient Node designs will require sensors such as thermistors and cameras or outputs such as motor controllers and displays. IP Cores for these interfaces are being developed in follow-on work by students enrolled in UNH's ECE993 course in Embedded Systems Engineering[9], detailed in APPENDIX VIII: Ongoing Research. The 100MHz OPB bus can quite easily handle bidirectional streams of 44kHz stereo audio (0.09% bus utilization) and even VGA video 640x480 x8bpp x30fps (5% bus utilization). High-efficiency kernel drivers are even available for many such devices under the Linux OS. The operating system naturally buffers input coming from such streams, and the input status of these buffers can be used by the GNATServer software to drive data through the DSP without data-loss. For longer-running or continuous I/O without kernel drivers, it will be necessary to implement timing controls in the DSP Peripheral, such as Pipelining with two-way depth pointer registers. In either case, the system will perform more efficiently at low power or high stress with the ability to suspend DSP operation on buffer over/underflow.

The major downside of the userland app structure of the GNATServer software is a double-burden incurred on the system bus. The software reads data from one peripheral, then writes it to another. A bidirectional video stream consuming 7% of the bus only gets the video out of the camera and into the DSP. If processed video then needs to come out of the DSP and stream onto the network, another 7% bus bandwidth is consumed. In order to alleviate this doubling of the bus-burden using this same approach, it would be necessary to implement a bus-mastering version of the Simulink® DSP Peripheral and accompanying Linux kernel driver. An even more elegant approach would be to integrate I/O peripherals directly into associated Simulink® DSP Peripherals. This would allow pre- and post- processing to be done in the DSP core without any CPU penalties.

7.3 Deficiencies

The Node design methodology presented here should be used in conjunction with an IP Core library. A large number of IP cores were evaluated in the selection of the cores used in the GNAT Node. Most options were eliminated due compatibility limitations. In order to take full advantage of this methodology, each compatible IP block considered should be tracked in a repository. The GNAT Node is the first set of IP blocks submitted to such a repository under development at UNH. Any library, of course, must be managed and maintained. Overhead is incurred validating IP blocks in the library for interoperability.

The toolsets used in the development environment also present a tremendous IT challenge. Linux kernels and cross-compilers change regularly, requiring a modest effort to be undertaken whenever a new version of compiler or new kernel code base is to be used. The open-source nature of these tools and cores makes these problems relatively tractable and quick to solve. Xilinx and The MathWorks, on the other hand, are closed-source commercial entities without a close-knit relationship. These companies have jointly ventured to make their products interoperably, and at the time of this writing, there are exactly two pair of Simulink® and EDK platforms that will interoperate. These versions cannot be mixed, nor are they forgiving. The departments responsible for the interoperation of these products are small and relatively un-influential parts of their respective companies. This makes version continuity across the entire tools-set as costly as it is imperative. Further cooperation among vendors would greatly improve accessibility to this design practice.

This tools-flow works only on Xilinx FPGAs. While Simulink® software can produce generic VHDL code that can be used by any FPGA synthesis tool for any silicon, the integration hooks such as hardware-in-the-loop are only compatible with the Xilinx synthesis tools at this time. Furthermore, the highly optimized, bit-accurate, cycle-true Simulink® blocks provided by Xilinx produce Xilinx-Only obfuscated VHDL code. Use of this process on another brand of silicon would require use of generic Simulink® blocks, which will produce less optimal hardware implementations. If use of this Node Design

Methodology becomes widespread, of course, all the major silicon vendors will be forthcoming with similar Simulink® compatibility.

REFERENCES

1 "Ambient Intelligence Node Prototyping using SOC Modular Design: Methodology and Implementation", thesis by Tomas M. Jankowski. Published concurrently with this thesis.

2 MATLAB® and Simulink® for Embedded System Design Pieter J. Mosterman Tutorial, Design, Automation and Test in Europe Conference & Exhibition, "Simulink® for Design and Programming Multiprocessor SoC" tutorial, Nice, France, April 16, 2007.
http://moncs.cs.mcgill.ca/people/mosterman/presentations/date07/tutorial.pdf

3 WARP OPB Export Tool, 2007, RICE University, http://warp.rice.edu/trac/wiki/sysgen2opb. A copy of the exact version used in this example is also available in the "*Windows\Source\Matlab Add-Ons*" directory of the GNATDVD.

4 "Lab 2: Introduction to sysgen2opb", 2007, RICE University, http://warp.rice.edu/trac/attachment/wiki/Workshops/Rice_2007March/Files/WARP_WorkshopExercise_2_sysgen2opbIntro.pdf, a copy of which is available in the "*Doc*" directory of the GNATDVD.

5 "Porting RTOS Device Drivers to Embedded Linux", 2004, Bill Weinberg, http://www.linuxjournal.com/article/7355. A copy of this article may also be found in PDF form in the "Doc" directory of the GNATDVD.

6 *xio.c*, *xio.h*, *xstatus.h, xbasic_types.h*, 2005, Xilinx, C source files for Xilinx OPB Bus register access.

7 Porting Monta Vista Linux to the XUP Virtex-II Pro Development Board, by Jonathon W. Donaldson. August 22, 2006. Located online at: https://rm-rfroot.net/files/courses/xupv2p/docs/carithers/report.pdf. A copy of this publication may be found in PDF form in the "Doc" directory of the GNATDVD.

8 "gSOAP: C/C++ Web Services and Clients Toolkit" , 2007, http://www.cs.fsu.edu/~engelen/soap.html. A copy of the toolkit, "gsoap_linux_2.7.9d.tar.gz", used in this project is found in the Linux\Source directory of the GNATDVD.

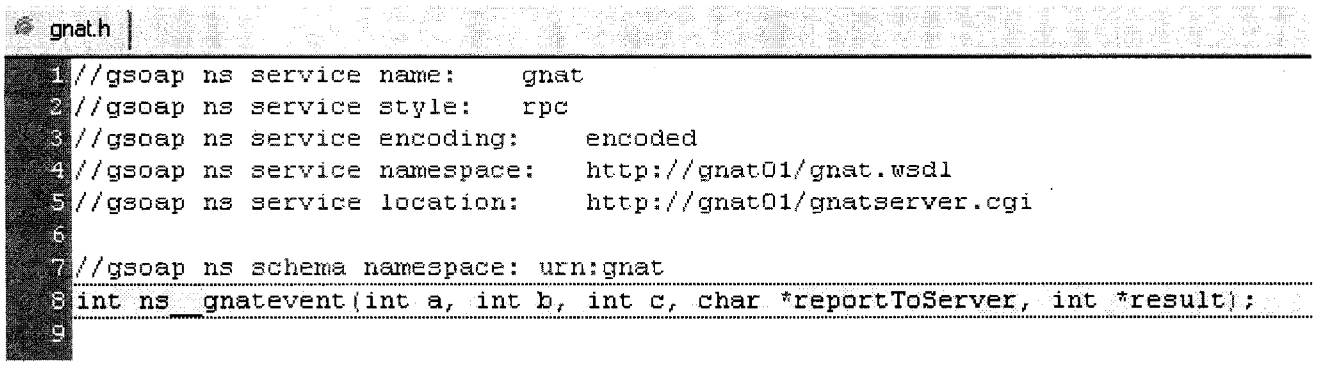
9 "GNAT: Global Network Academic Test Initiative" Datasheet generated by ECE993 class describing their ongoing work, http://www.cidlab.unh.edu.

APPENDIX I: Installation Procedure for Commercial MathWorks-Simulink®/Xilinx-EDK environment

Total time ~ 3hrs

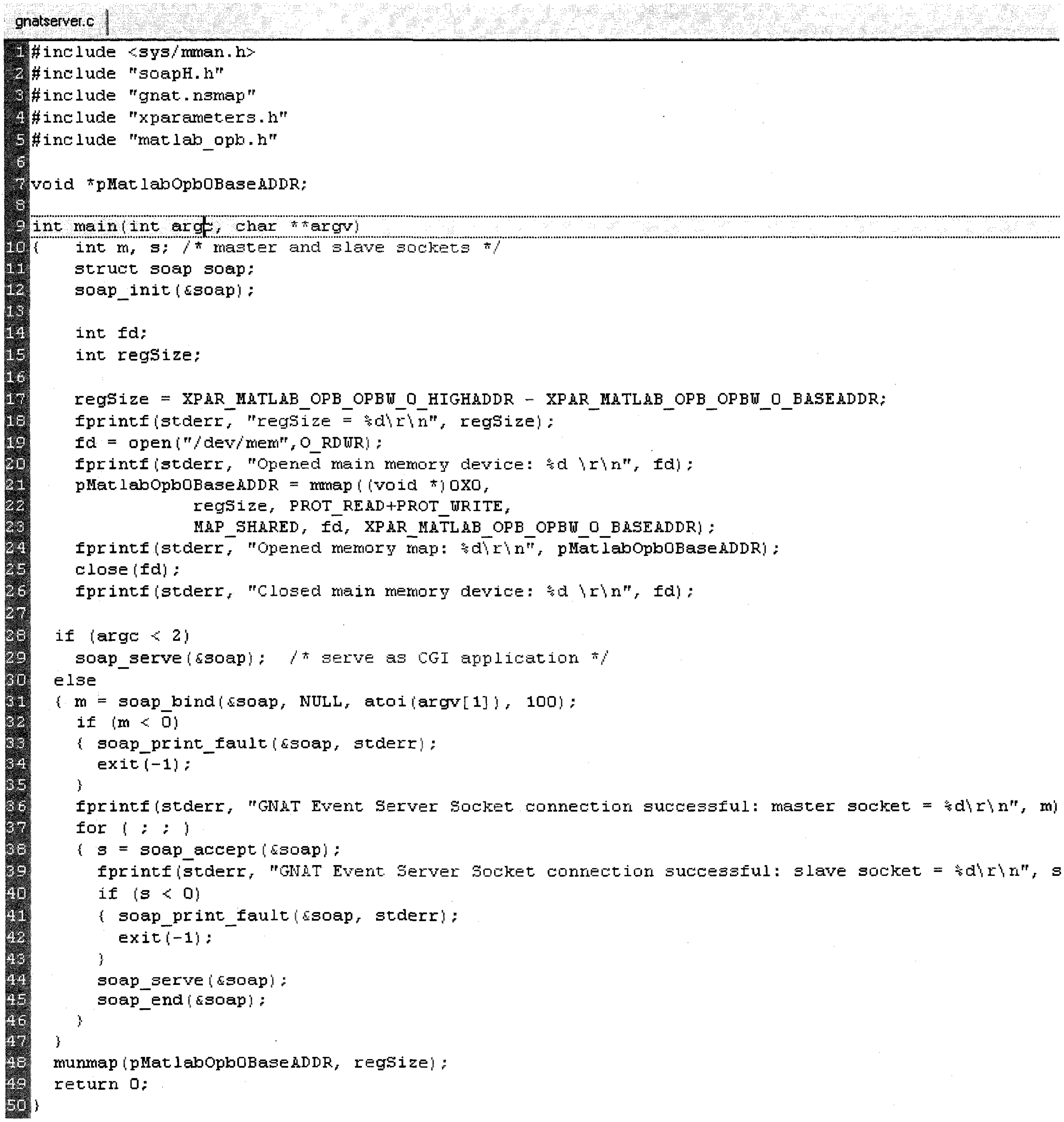
1. The MathWorks Simulink®
   a. Insert MATLAB® 2006a Disk 1
   b. Select Custom Install
   c. Change installation directory to "C:\MATLAB" – nothing else will work with Xilinx!
   d. Insert MATLAB® 2006a Disk 3
   e. At conclusion of installation, MATLAB® software will recommend that you download several updates. DO NOT! These will break Xilinx compatability!
   f. Replace C:\MATLAB\bin\win32\comcli.dll with the comcli.dll from the Xilinx System Generator CD.
2. Xilinx ISE 8.2i – if this was already installed on your system before you installed Matlab, too bad! Take it off, reboot, and reinstall it. ISE installs additional hooks if it sees Matlab in place during installation.
   a. Insert Xilinx ISE 8.2i Disk a
   b. Accept defaults.
   c. At conclusion of installation, reboot as instructed.
   d. Insert Xilinx ISE 8.2i Disk b – Yes, it appears to be identical.
   e. Accept defaults.
   f. At conclusion of installation, reboot as instructed.
   g. Insert Xilinx ISE 8.2i Update Disk.
   h. Run 8_2_02i_win.exe; accept defaults.
   i. Ignore the RocketIO wizard on that disk.
   j. Insert Xilinx Core Generator disk
   k. Run the "setup.exe" within ise_82i_ip_update.zip.
   l. Insert Xilinx System Generator disk
   m. Accept defaults.
   n. Insert Xilinx EDK disk.
   o. Accept defaults.
   a. Reboot.

APPENDIX II: gnat.h SOAP Service Definition



```
//gsoap ns service name:    gnat
//gsoap ns service style:   rpc
//gsoap ns service encoding:    encoded
//gsoap ns service namespace:   http://gnat01/gnat.wsdl
//gsoap ns service location:    http://gnat01/gnatserver.cgi

//gsoap ns schema namespace: urn:gnat
int ns__gnatevent(int a, int b, int c, char *reportToServer, int *result);

```

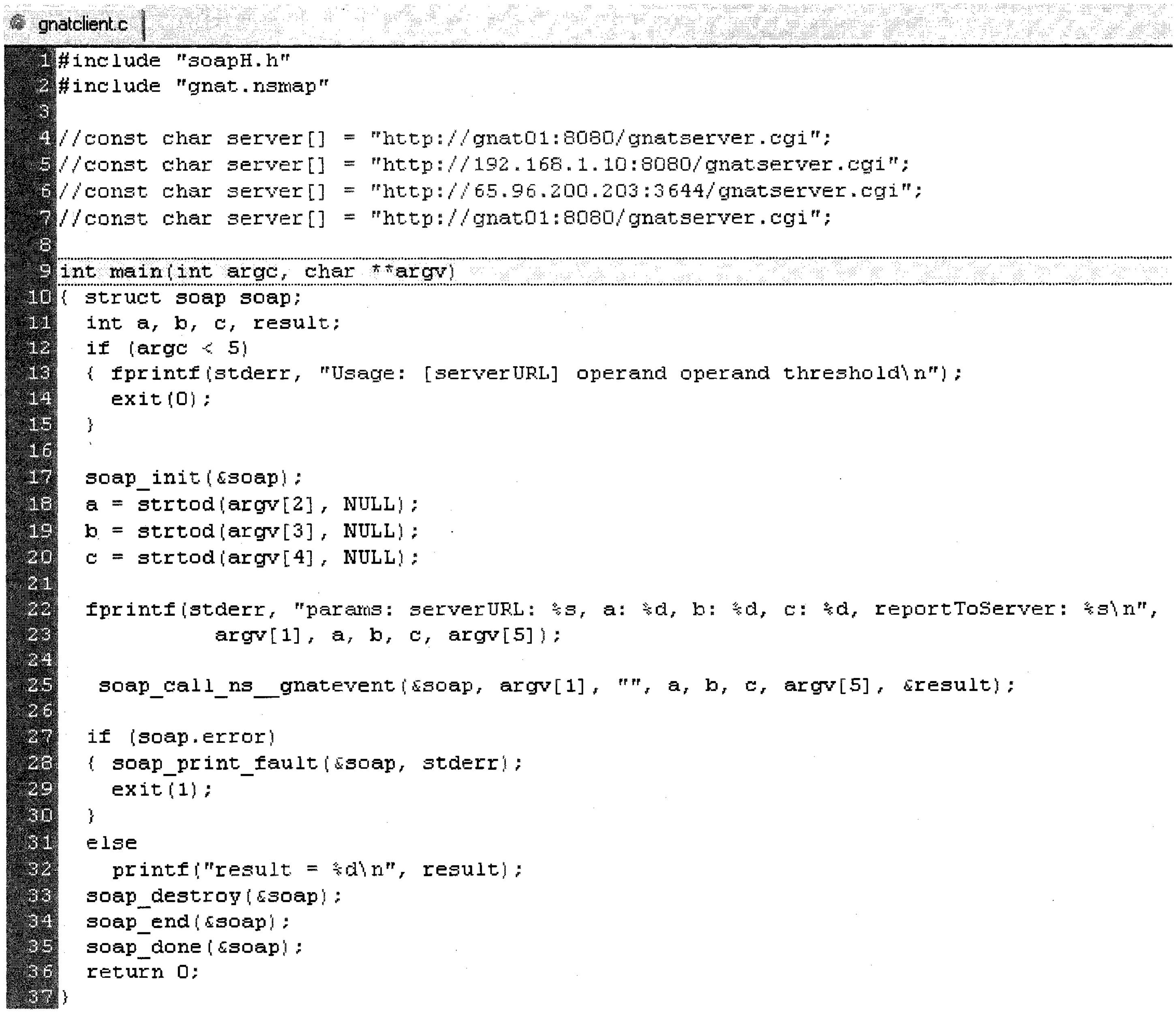
APPENDIX III: GNAT Node (Server) Demo Code


gnatserver.c

```c
#include <sys/mman.h>
#include "soapH.h"
#include "gnat.nsmap"
#include "xparameters.h"
#include "matlab_opb.h"

void *pMatlabOpb0BaseADDR;

int main(int argc, char **argv)
{   int m, s; /* master and slave sockets */
    struct soap soap;
    soap_init(&soap);

    int fd;
    int regSize;

    regSize = XPAR_MATLAB_OPB_OPBW_0_HIGHADDR - XPAR_MATLAB_OPB_OPBW_0_BASEADDR;
    fprintf(stderr, "regSize = %d\r\n", regSize);
    fd = open("/dev/mem",O_RDWR);
    fprintf(stderr, "Opened main memory device: %d \r\n", fd);
    pMatlabOpb0BaseADDR = mmap((void *)0X0,
               regSize, PROT_READ+PROT_WRITE,
               MAP_SHARED, fd, XPAR_MATLAB_OPB_OPBW_0_BASEADDR);
    fprintf(stderr, "Opened memory map: %d\r\n", pMatlabOpb0BaseADDR);
    close(fd);
    fprintf(stderr, "Closed main memory device: %d \r\n", fd);

  if (argc < 2)
    soap_serve(&soap);  /* serve as CGI application */
  else
  { m = soap_bind(&soap, NULL, atoi(argv[1]), 100);
    if (m < 0)
    { soap_print_fault(&soap, stderr);
      exit(-1);
    }
    fprintf(stderr, "GNAT Event Server Socket connection successful: master socket = %d\r\n", m)
    for ( ; ; )
    { s = soap_accept(&soap);
      fprintf(stderr, "GNAT Event Server Socket connection successful: slave socket = %d\r\n", s
      if (s < 0)
      { soap_print_fault(&soap, stderr);
        exit(-1);
      }
      soap_serve(&soap);
      soap_end(&soap);
    }
  }
  munmap(pMatlabOpb0BaseADDR, regSize);
  return 0;
}
```

```c

int ns__gnatevent(struct soap *soap, int a, int b, int c, char *reportToServer, int *result)
{
    struct soap soapReporting;
    int confirmation;

    if (c == INT_MAX) {
        fprintf(stderr, "Event received: %d\r\n", a);
        *result=0;
        return SOAP_OK;
    }


    fprintf(stderr, "Event received: %d, %d, %d, %s\r\n", a, b, c, reportToServer);
//Write Reg Macro
    matlab_opb_WriteReg_inA(pMatlabOpb0BaseADDR, a);
    matlab_opb_WriteReg_inB(pMatlabOpb0BaseADDR, b);
    *result = matlab_opb_ReadReg_outProduct(pMatlabOpb0BaseADDR);

  fprintf(stderr, "Event solution: %d\r\n", *result);
  if (*result > c) {
      fprintf(stderr, "Forwarding event solution: %d\r\n", *result);
      soap_init(&soapReporting);
      soap_call_ns__gnatevent(&soapReporting, reportToServer, "", *result, 1, INT_MAX,
            reportToServer, &confirmation);
      fprintf(stderr, "Forwarding confirmation: %d\r\n", confirmation);
  }

  if (soap_errno == EDOM)   /* soap_errno is like errno, but compatible with Win32 */
  { char *s = (char*)soap_malloc(soap, 1024);
    sprintf(s, "Can't understand event of %d %d", a, b);
    sprintf(s, "<error xmlns=\"http://tempuri.org/\">Can't understand %d, %d</error>", a, b);
    return soap_sender_fault(soap, "Event rejected", s);
  }
  return SOAP_OK;
}
```

APPENDIX IV: GNAT Client Source Code



```c
#include "soapH.h"
#include "gnat.nsmap"

//const char server[] = "http://gnat01:8080/gnatserver.cgi";
//const char server[] = "http://192.168.1.10:8080/gnatserver.cgi";
//const char server[] = "http://65.96.200.203:3644/gnatserver.cgi";
//const char server[] = "http://gnat01:8080/gnatserver.cgi";

int main(int argc, char **argv)
{ struct soap soap;
  int a, b, c, result;
  if (argc < 5)
  { fprintf(stderr, "Usage: [serverURL] operand operand threshold\n");
    exit(0);
  }

  soap_init(&soap);
  a = strtod(argv[2], NULL);
  b = strtod(argv[3], NULL);
  c = strtod(argv[4], NULL);

  fprintf(stderr, "params: serverURL: %s, a: %d, b: %d, c: %d, reportToServer: %s\n",
            argv[1], a, b, c, argv[5]);

   soap_call_ns__gnatevent(&soap, argv[1], "", a, b, c, argv[5], &result);

  if (soap.error)
  { soap_print_fault(&soap, stderr);
    exit(1);
  }
  else
    printf("result = %d\n", result);
  soap_destroy(&soap);
  soap_end(&soap);
  soap_done(&soap);
  return 0;
}
```

APPENDIX V: GNATDVD File Index

I. Doc – documentation library
   A. Application Development Process for GNAT, A SOC Networked System (PDF of this thesis)
   B. Architecture GNAT, A SOC Networked System (PDF of accompanying thesis by Tomasz Jankowski)
   C. Porting MontaVista Linux to the XUP Virtex-II Pro Development Board (PDF of foundational OS work by Jonathan Donaldson)
   D. Porting RTOS Device Drivers to Embedded Linux (PDF of Article used in SW Development)
   E. WARP_WorkshopExercise_2_sysgen2opbIntro (PDF of a walk-through of the OPB Peripheral Wrapper for Matlab used)

II. Linux – software for use on the Linux Compile Server
   A. Distribution – Finished Works for the Linux Compile Server
      1. opt_crosstoolDistribution.tar.gz -- powerpc405 library source code used by cross-compiler.
      2. xupv2pDistribution.tar.gz -- This contains the whole xupv2p project directory.
      3. gnatDistribution_1.1.tar.gz -- enhanced gnatServer, capable of accpeting a "threshold" parameter and a URL of a gnatServer to which products exceeding said thresholds shall be reported.
      4. gnatLinuxPC_1.1.tar.gz -- the gnat client and server ported to PC running Linux, for demos.
      5. UpdatedScripts: -- directory of revised versions of scripts found in the xupScriptsAndLinks.tar.gz archive
      6. readme.txt – a text file with detailed information about the files in this directory, including installation instructions.
   B. Source – this is a collection of files referenced by Jonathan Donaldson's work on Porting Monta Vista Linux to the XUP Virtex-II Pro Development Board

III. Windows – Software for use on the Windows PC
   A. Distribution – Finished works for the Windows PC.
      1. gnatClient_VC_1.0.zip -- MS Visual Studio 2005 project compiling gnat command-line client.
      2. gnatClient_VC_1.1.zip -- MS Visual Studio 2005 project compiling gnat command-line client.
      3. XUPV2P_without_MatlabMultipliers_EDK9.1.zip -- XUPV2P project built according to Jonathan Donaldson thesis under EDK 9.1.
      4. XUPV2P_without_MatlabMultipliers_EDK8.2.zip -- XUPV2P project built without Matlab peripherals
      5. XUPV2P_with_MatlabMultipliers_EDK8.2.zip -- XUPV2P project built with Matlab peripherals

6. matlab_opb.zip -- Matlab model of a multipler (included in above)
7. readme.txt – a text file with detailed information about the files in this directory, including installation instructions.

B. Source – Design Environment tools and IP Cores that had to be downloaded separately.

1. Digilent_pack -- This is the Board Support Package (BSP) for the XUPV2P board.
2. xupGenACE scripts -- the xupGenACE.tcl script that came with EDK9.1 was broken, so included is the working one from EDK8.2.
3. Matlab Add-Ons -- these packages must be installed into Matlab in order to produce Xilinx OPB peripherals. Each zipfile contains its own installation instructions within.
4. readme.txt – a text file with detailed information about the files in this directory, including installation instructions.

C. Tools – Windows Shareware used in this project.

1. Crimson Editor -- an handy universal file editor
2. TclTK Interpreter -- tcl is installed with EDK, but this intpperpreter will be necessary if using tcl scripts (such as the gnatClient wrapper) on a Windows PC without EDK.
3. TeraTerm Terminal Emulator -- a handy serial/telnet terminal emulator
4. readme.txt – a text file with detailed information about the files in this directory, including installation instructions.

APPENDIX VI: Ambient Intelligence Node Development Tools Set Quick Start Guide

The GNAT Ambient Intelligence Node Development Tools Set includes two design environments spread across two separate computers. One of these computers must be running Windows XP or higher, with at least 1GB of RAM and 3GHz Pentium 4 CPU. The Simulink® and EDK tools must be installed precisely according to the procedure in APPENDIX I: Installation Procedure for Commercial . After the main commercial tools are installed, it is time to refer to the *windows* directory of the GNATDVD; further utilities required for this process are listed in APPENDIX VII: Notable Files and Scripts.

The second computer should be a Pentium III or higher with at least 256MB of RAM running a recent distribution of Linux; Fedora 6 was used in development of the original GNAT demo. The Linux tools environment has been archived in a pre-installed form, and should re-hydrate from the tarballs fully functional if done properly. The instructions and files for this process are in the *Linux/Distribution* directory of the GNATDVD.

Various Project files will need to be looped through both the Windows and Linux development systems, so file sharing between the two is advised. The original GNATServers were set up to share (with read/write access) the ece992 user's account directory under which the XUPV2P project is found (/home/ece992/xupv2p). This structure is recommended. Intelligent use of soft links may optimize this structure for Linux servers shared by multiple users.

The Windows\Distribution\XUPV2P_x_Matlab_Multipliers_EDKx.zip archives are EDK sample projects, descriptively named. Designers should choose from these EDK projects for a starting point for any Node designs using the XUPV2P board. While GNATDVD includes all of the components used in creation of the GNAT Node EDK project, it takes several hours to build the GNAT EDK project from scratch. These project were generated using the process laid out in Doc\Porting MontaVista Linux.pdf, but archived for use as starting points because this procedure is very difficult.

With all systems and projects set up, the Node development process is as follows:

1. Create, download, or purchase Linux or OPB-compatible IP cores for Node or Node-attached Hardware.
2. Create /update desired DSP block(s) in Simulink
3. Export OPB peripherals as specified in WARP_WorkshopExercise_2_sysgen2opbIntro.pdf.
4. Connect OPB peripherals in EDK, recompile bitstream.
5. Copy xparameters.h, bsp directory, and any special C files to Linux server.
6. Configure kernel-level support of Linux compatible peripherals (as necessary).
7. Assign MAC address with setMAC.tcl
8. Compile kernel with updateKernel.sh
9. Copy kernel (zImage.elf) from Linux to PC
10. Generate system.ACE file (Xilinx genACE.sh script)
11. Copy system.ACE file to CompactFlash DOS partition (directly from Windows or from Linux via usb mount point)
12. Update SOAP data structure in gnat.h
13. Update gnatserver.c to reflect desired Node behavior
14. Compile and load GNATServer to CompactFlash Linux Partition (usb3) using mkrootfs script.
15. Insert flash drive into XUPV2P board and turn on new Node.

APPENDIX VII: Notable Files and Scripts

Linux Side Notable Files:

mkrootfs – directory containing files to be copied into the root filesystem of the Node
mkrootfs/gnat – directory containing GNATServer source and object code
gnatserver.c – main GNATServer source code; this controls data flow in the Node
gnat.h – SOAP service definition for GNATServer
gnatclient.c – main source code for a command-line GNAT client program
mkrootfs/etc/init.d/rcS – boot time services configuration. Calls GNATServer and specifies port
usb – a mount point for the first (DOS) partition of the Node CompactFlash card
usb3 – a mount point for the third (Linux) partition of the Node CompactFlash card
montaVista_2_4_devel – root of embedded linux source code distribution
zImage.elf – kernel object code – this must be copied to Windows PC for inclusion into .ACE file through EDK.

Linux Side Scripts:

setMAC.tcl – hard-codes a specified MAC ADDRESS into kernel source
updatekernel.sh – a script that recompiles the kernel, producing zImage.elf

Windows Side Notable Files:

montaVista_ACE\system.ACE – this file, when copied to the DOS partition of the Node CompactFlash Drive, is used by the XUP board's ACE controller to load the FPGA and boot the CPU.

ppc405_0\include\xparameters.h – C file enumerating hardware resources generated by EDK. This file is required in Linux mkrootfs/gnat to provide correct peripheral address mapping in the GNATServer application (dynamic on recompiled FPGA).

drivers\matlab_opb_opbw\src\matlab_opb.h – C file describing Matlab OPB Peripheral interface, required in Linux mkrootfs/gnat to provide correct peripheral interface (static).

ppc405\libsrc\cpu_ppc405_v1_00_a\src – contains xio.c and xio.h, C libraries required for connecting to Xilinx peripheral interfaces (static).

ppc405\libsrc\common_v1_00_a\src – contains xstatus.h and xbasic_types.h, C libraries required for connecting to Xilinx peripheral interfaces (static).

Windows Side Scripts:

genACE – a Xilinx-provided script that combines the ".bit" FPGA bitstream with the zImage.elf kernel object code to produce an ".ace" Xilinx SystemACE FPGA boot file. This script expects the following file structure under the EDK project directory:

montaVista_ELF\zImage.elf (input)

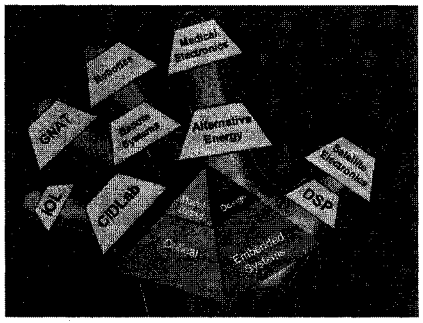
montaVista_genACE\xupGenACE.tcl script location and xupGenACE.opt options file

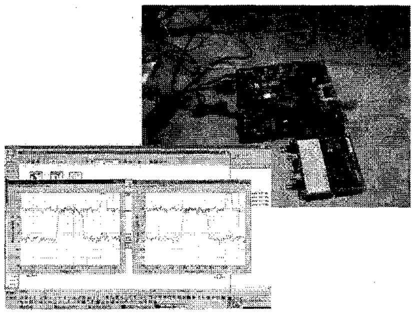
montaVista_ACE\system.ACE (output)

APPENDIX VIII: Ongoing Research

GNAT: Global Network Academic Test Initiative

Critical embedded system engineering computer architectures





Highlights: GNAT is an (inter)-networked collection of next-generation cost-effective computer architectures for security systems. It is a prototype of an innovative computing environment that is compliant with *Critical Embedded Systems* engineering and *Ambient Intelligence* (AmI) principles. GNAT is also a low cost object in a class represented by: RAMP, BEE2, CRAY XDI, SRC-7, SGI Altix/RASC and enterprise / grid / cloud computing (see IEEE Computer Jan 2008).

- *The GNAT Initiative encompasses creating prototypes, tools and components to develop and implement:*
  - *a global-range, scalable, organic, multiprocessor, networked computational environment*
  - *that can be heterogeneous, and reconfigurable, at all scales from the node to the global*
  - *the physical scale can range from the nm of the chip to the Mm of the globe*

GNAT enables embedded computation that is optimized to applications in critical system security applications. GNAT's architecture supports Computation with reconfigurable and self-optimizing Morphing Hardware that encompasses:

- Global Range
- Scalability
- Computing On Demand
  - Access to Embedded Distributed Processing and Storage
- Ambient Intelligence -- "self-awareness"
  - Anomaly Detection and "tracking"

GNAT Technical Data

- *Sensor Layer*
  - Cypress PSoC
  - Video Camera
- *Mobile Layer*
  - iRobot Create (GNATbot)
- *Node Layer Hardware*
  - Xilinx XUP Virtex-II Pro board with Power Supply, USB- JTAG programming cable (Digilent); XUP Board Support Package (Free download); 512 MB PC2100 SDRAM (Commercial Product); 256MB Compact Flash (Commercial Product)
  - Xilinx Spartan3AN board with Power Supply, programming cable, (Nu Horizons)

    - Xilinx Spartan3 Xtreme DSP board with Power Supply, programming cable, (Nu Horizons)
- *Node Layer OS Software:*
    - MontaVista Linux 2.4 (Free download);
    - PC (Windows XP) (Commercial Products);
    - PC (Linux, i386 RedHat 6.0 distribution recommended) (Commercial Hardware, Free download OS)
- *Node layer Development Tools:*
    - MATLAB® 2006 R14.1, R14.2, or R14.3 with Signal Processing Blockset™ (Commercial Product);
    - Xilinx v8.1 EDK (Commercial Product); Xilinx v8.1 ISE Foundation with latest service pack (Free download);
    - Xilinx v8.1 System Generator for DSP
    - sysgen2opb.m wrapper script from WARP/Rice University (Free download); EDK OPB Export Tool (Free download); Busy Box (Free download); Crosstool (Free download); MkRootFS (Free download); Tera Term Pro (Free download); Tcl interpreter (Free download)
- *Server Layer:*
    - Servers: Multi-Core Sun Blades Cluster in Rack
    - Hardware: IP 132.177.206.176, DELL laptop
    - Software: Linux, Windows
    - OS Development Tools
    - Operating Procedure – Quick Start Guide & User's Manual
    - Security
- *Communication Layer:*
    - Hardware: Wireless; Internet, e.g., protocols
    - Development tools: TechonLine

Application Development Track

- *Hardware Development Track using Model Based Design*: VHDL, Simulink, Nu-Horizon, IPs for SoC
- *Software Development Track*: Windows, Linux, Xilinx® EDK VHDL, MathWorks Simulink® software
- *Embedded Systems Development Track*: Hybrid (mobile and stationary) GNAT nodes
- *Networking/Wireless DeIPs for NoC; High speed; Wireless*: ZigBee, MIMO, Internet, e.g., TechonLine (IOL)

On Going Research and Development

- Global Transportation System (commercial application)
- "Take me to the Ball Game" (educational application)
- Earth Magnetic Field Monitoring (scientific application)
- Experimental Mine Monitoring PLUTO (AmI application)
- "Plant a Flag" – auto-configurable wind-field and other environmental data-field measurement system (commercial application)
- CMP/IP Repository: ASIC; FPGA (research and educational application)
- Model Based Design (research application)
- DSP (research application)

Contact: Dr. Andrzej Rucinski CIDLAB-US Director,
University of New Hampshire, Kingsbury Hall, Durham, NH 03284
Tel 603 862 1381 e-mail a.rucinski@unh.edu
www.cidlab.unh.edu