Simulation of an Implementation and Evaluation of the Layered Radio Architecture

by

James Neel

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

Approved:

Dr. Jeffrey H. Reed

Dr. William H. Tranter

Dr. Mark T. Jones

December 2002

Blacksburg, VA

Keywords: reconfigurable computing, ccm, stallion, colt, wireless, soft radio, dsp, software radio, wcdmam layered radio architecture, TMS320C6201, custom computing machine, object oriented simulation , oop

Simulation of an Implementation and Evaluation of the Layered Radio Architecture

James Neel

(Abstract)

Software radio is a radio that is substantially defined in software and whose physical layer behavior can be significantly altered through changes to its software. As a primary goal of software radio is the ability to support existing and future wireless protocols, software radio necessitates the use of a rapidly reprogrammable baseband processing solution. However third generation wireless protocols represent a significant increase in complexity over second generation protocols. Due to the natural performance sacrifices that must be made when moving an application from an Application Specific Integrated Circuit (ASIC) to a general purpose processor or a digital signal processor, it is feared that reprogrammable processing solutions may not suffice for the emerging wireless protocols, which would significantly hinder the realization of software radio, particularly in the handheld domain where power consumption and chip area are critical.

Recently, the Configurable Computing Lab at Virginia Tech developed a new breed of reprogrammable processor which they called "custom computing machines" (CCM). Representing a dramatic departure from traditional architectures used for baseband processing solutions, CCMs utilize a large number of optimized and programmable processing cores connected through a programmable mesh. Due to this architectural approach, CCMs have been promoted as supplying a level of processing power and power efficiency similar to ASICs while providing a level of reconfigurability similar to that of a DSP. Subsequently, Dr. Srikathyayani Srikanteswara proposed a new software radio architecture, known as the Layered Radio Architecture, which is intended to facilitate the inclusion of CCMs into a software radio.

The primary goal of the research presented in this thesis is to demonstrate how a particular CCM, Stallion, can be used within the Layered Radio Architecture to provide sufficient processing performance, power efficiency, and reconfigurability to meet the constraints of the handheld domain through implementations of a single user adaptive receiver with adaptive complex filtering and a W-CDMA downlink rake receiver. These metrics are measured from a detailed simulation of Stallion and the Configuration Layer of the Layered Radio Architecture using advanced object oriented programming techniques that facilitate the inclusion of statistics gathering routines into normal operation. To provide perspective, these statistics are compared to the performance that could be expected from an implementation on a top-of-the-line DSP.

Acknowledgements

I would like to thank my advisor, Dr. Jeffrey H. Reed, for his continuing support, words of encouragement, as well as his admonishments to get out of the lab and attend class. I would also like to thank the rest of my committee, Dr. Jones and Dr. Tranter, for their involvement in my thesis. I would also like to thank Dr Athanas and Dr Ha for their help understanding the intricacies of hardware implementations.

I would like to specifically express my appreciation to Dr. Srikathyayani Srkanteswara who managed my portion of the GloMo project and was both a mentor and a friend. Further, I would like to thank Santiago Leon, Maneesh Soni, and Michael Hosemann who also worked with me on this project and Dr. Bob Boyle who coordinated the GloMo project.

Finally, I would like to convey my gratitude to my family and my girlfriend for their support as well as their patient, persistent reminders to write this thesis.

Table of Contents

Listing of Figures

Listing of Tables

Chapter 1

1 Introduction

1.1 Software Radio

Beginning in 1979 with the Air Force's Tactical Anti-Jam Programmable Signal Processor (TAJPSP) program, there has been a migration of radio architectures from those that implemented functionality with analog and integrated circuit (IC) components to architectures that use microprocessors (μP), digital signal processors (DSP), and field programmable gate arrays (FPGA) to perform the bulk of the operations. The waveforms of radios designed using these components are no longer defined by hardware; rather, it is the software loaded onto the μP, DSP or FPGA that determine the radios' waveforms. The recognition of this paradigm shift – from hardware to software implementations – led to the coining of the term "software radio" by Joe Mitola in 1991 [1] to differentiate radios that are implemented primarily in software from those implemented principally in hardware.

Accompanying improvements in technology, the term "software radio" has expanded to encompass many new concepts since its initial introduction. These include the following:

- "Multi-band Multi-mode Radio" (MBMMR) – a radio that can operate over multiple center frequencies and bandwidths with different modulations
- "Future Proof" Radio – a radio that is designed in such a way that it supports incremental upgrades, particularly to just its software, that prevents the radio's obsolescence
- "Ideal" or "Ultimate" Software Radio – a MBMMR where the digitization boundary occurs at the antenna and *ALL* of the processing is performed in the digital domain (implicitly with software)

- "Cognitive Radio" – a radio that is 1) knowledgeable of its environment and how changes to its operating parameters will impact its performance and 2) capable of altering its operating parameters.

There are further subtler refinements to the software radio concept including Open Systems Architecture, object oriented architecture, and over-the-air updates that describe how a software radio should be implemented and features it should support. Others include extensions to the basic software radio concept to form fine gradations to evaluate how well a radio satisfies the software radio concept. For instance, the SDR Forum has established a five-tier system to describe different levels for being a software radio [2].

To avoid confusion, this thesis will consider software radio to be as defined in [3]: *"a radio that is substantially defined in software and whose physical layer behavior can be significantly altered through changes to its software."*

As software radio holds the potential to revolutionize wireless communications, software radio is a hot topic for research, the focus of intense efforts by many governments and multinational consortiums. Currently, the research in software radio is being led by the US military as part of their Joint Tactical Radio System (JTRS) program and the Software Defined Radio Forum (SDR Forum), a multinational consortium of corporations and research institutes. JTRS and the SDR Forum are currently developing a Software Communications Architecture (SCA) that is intended to be usable across all domains [4]. Domains are *"specific physical environments with unique operating constraints that should be considered when constructing a radio"* [3]. For instance the handheld domain, *e.g.* a cell phone, has very different operating constraints – power consumption, form factor, price – from the fixed domain, *e.g.* a base station. One of the key challenges to developing a SCA applicable to all domains is the impact the size and power constraints of the handheld domain has on the baseband processing elements. The effect of the combination of these constraints has led to the concession that some of the functionality implied in the SCA may still need to be implemented using application specific integrated circuits (ASIC) as μP, DSP, and FPGA solutions alone cannot meet the demands of the handheld domain. However, this is in opposition to the fundamental goal of software radio, as the use of ASICs leads to a radio that is substantially defined in *hardware* and whose physical layer behavior *cannot* be significantly altered through changes to its software.

Recently, the Configurable Computing Lab at Virginia Tech developed a new breed of processor which they called “custom computing machines” (CCM). A dramatic departure from traditional processing solutions, CCMs have been billed as supplying processing power and power efficiency similar to ASICs while providing reconfigurability similar to a DSP. The goal of the research presented in this thesis was to show how one particular CCM, Stallion, could be used to provide sufficient processing performance, power efficiency, and reconfigurability to meet the constraints of the handheld domain.

1.2 Traditional Processing Solutions for Software Radio

The software radio concept is built upon the use of reconfigurable (reprogrammable) hardware whose operation can be changed through software modifications. Traditional processing devices used in software radios include Digital Signal Processors (DSP), Microprocessors (μP), and Field Programmable Gate Arrays (FPGA). Most software radio architectures also make provisions for the use of Application Specific Integrated Circuits (ASIC) when a combination of μP, DSP, and FPGA devices does not provide sufficient computational density (computations per unit volume) and/or power efficiency.

When selecting the processing solution for a handheld software radio there are six key parameters that affect the performance of a software radio: computation power, computation density, reconfiguration time, power efficiency, application flexibility, and hardware flexibility. There are many other significant factors that may be considered when selecting a chip, such as price and the chip’s development cycle, but these are not central to the radio’s performance. For the purposes of this thesis, only performance parameters were considered.

Computation power is related to a chip’s clock speed as computation power is inversely related to the time required to perform some operation and for identical processor architectures, faster clock speeds directly translate to increased computation power. However, since different chips have different architectures and support vastly different operations, a simple clock comparison or even a comparison of MIPS (million instructions per second) or FLOPS (floating point operations per second) does not fully capture the actual computational capacity of a chip. This is a well-recognized fact in the world of signal processing, thus companies such as Berkley Design Technology Inc. (BDTI) design have developed tests modeled on commonly implemented algorithms, *e.g.*, FIR, FFT, which they implement and optimize on a number of chips and then measure and “score” based off the actual time required to complete an operation. BDTI’s scores for 2002 are shown below in Figure 1.1.

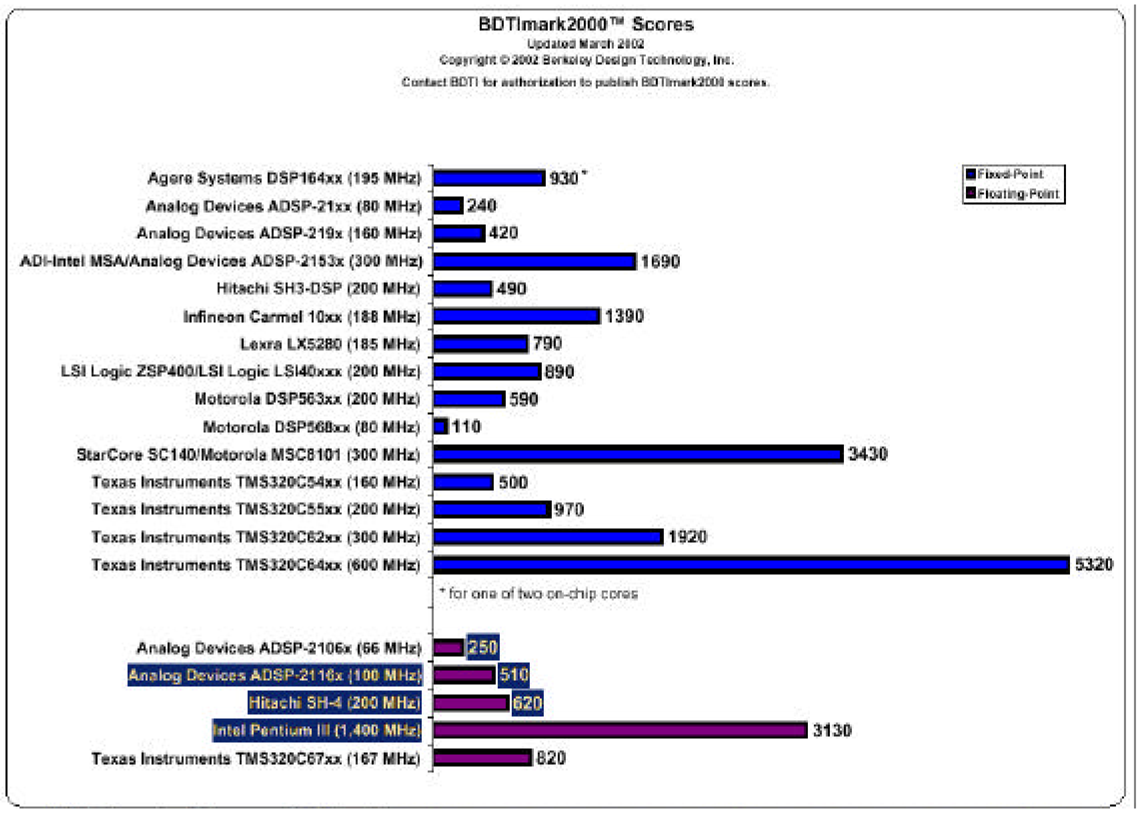


Figure 1.1 BDTI Scores [7]

Notice that there is only a loose relationship between any chip’s speed and its corresponding score. In order to support third generation (3G) wireless standards as well as the wide range of potential waveforms, higher computation power is generally better for a software radio.

Computation density is the raw computational power per unit volume. In the handheld domain, space constraints may dictate the use of a chip with a higher computation density and lower overall computation power. However, in general both higher computation density and computation power is desirable for a software radio implemented in the handheld domain.

Reconfiguration time is the amount of time required to effect a change in the behavior of the device. Due to the expectation that software radios will be involved in vertical handoffs, handoffs between cells using different waveforms, handheld software radios will need to be able to completely alter their behavior within a very short period of time. Thus a shorter reconfiguration time is desirable for handheld software radios.

Power efficiency is the ratio of a chip’s computation power to the amount of power consumed to perform the operations. Although a concern in all domains, power efficiency is especially critical in the handheld domain where the average user closely associates battery life with the value of the radio. Thus heightened power efficiency is desirable in the handheld domain.

Application flexibility is the level to which significant changes can be made to the application being run by the processor and the level of granularity available for change. For instance, a chip that facilitates the insertion/removal of individual software modules, control of its operational parameters and can also completely change waveforms is more flexible than a chip that only supports changes that involve a complete change to the chip. For a software radio, a processor with greater application flexibility is always desirable.

Hardware flexibility is the level to which significant changes can be made to the computational hardware and the level of granularity available for change. Hardware flexibility reflects a processor’s ability to “tune” its computation hardware to the current application. Hardware flexibility may be reflected in an ability to change between functions performed in hardware, *e.g.*, Galois arithmetic circuitry to FFT circuitry, or the circuitry used to calculate an operation, *e.g.,* changing from 16-bit to 8-bit arithmetic. Hardware flexibility permits a processor to achieve high performance levels for many diverse applications.

These six parameters can be envisioned as comprising a solution space where each of the parameters represents a different basis vector for the space. A solution space is a handy way to envision the capabilities a device presents. For the purposes of simplicity, this thesis uses the inverse of reconfiguration time instead of the reconfiguration time, so that any increase along a basis function represents a more desirable result. As described, the solution space for the handheld domain can be envisioned as shown below in Figure 1.2 . Note that none of these parameters can reasonably assume a negative value.

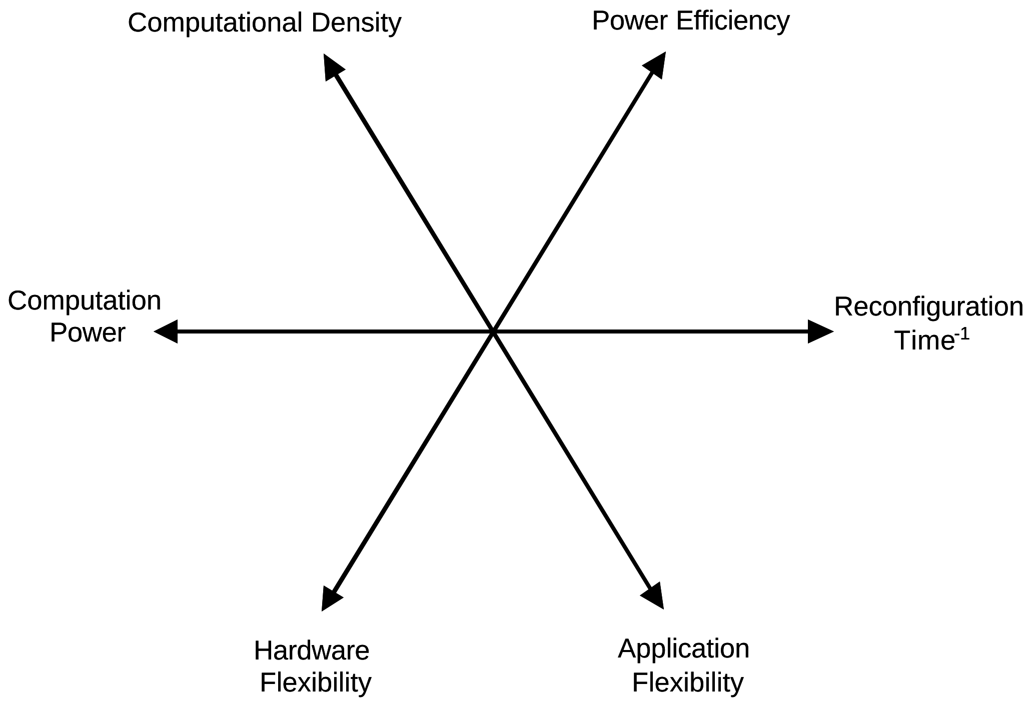


Figure 1.2 Handheld Domain Processor Solution Space

The ideal processing solution for the handheld domain would maximize its operating parameters along each of these vectors. This is shown below in Figure 1.3.

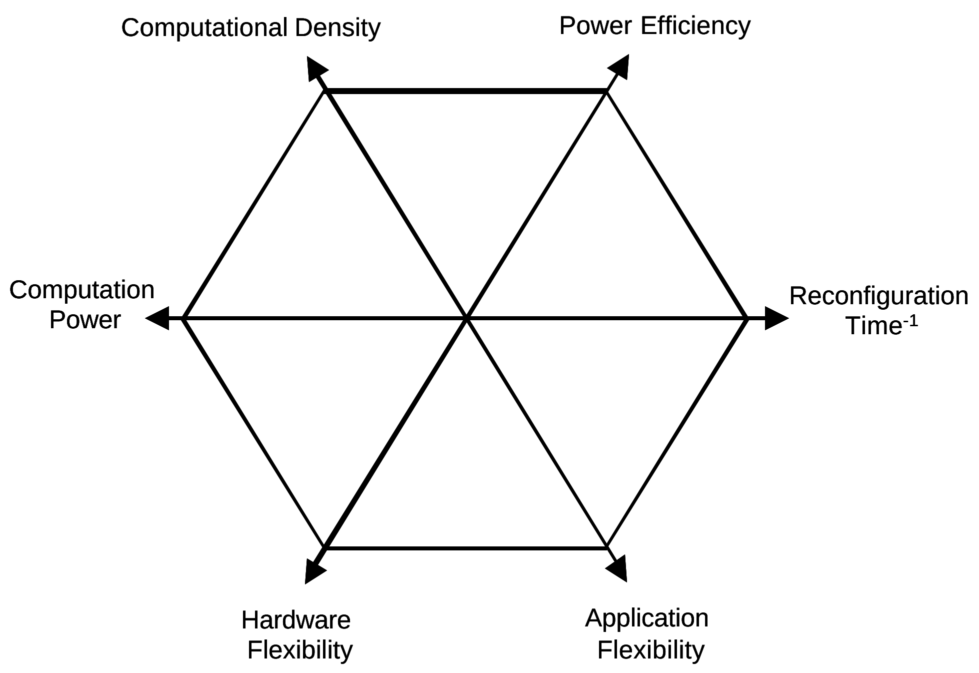


Figure 1.3 Ideal Solution Space

The following reviews how traditional processing solutions are generally implemented and how the solutions compare to each other and to an ideal processor in the solution space. Note that there will be great variation between specific implementations of devices within each category of processing solution. All information should be treated as generalizations on the conceptual approach to creating that class of processing solution.

1.2.1 Application Specific Integrated Circuits

Application Specific Integrated Circuits (ASIC) are chips designed by laying out individual gates and traces on a silicon substrate to implement a particular application. Because the entire chip is designed with a particular application in mind, it is possible to optimize an ASIC for computation power, power efficiency, and computation density, achieving performance levels which is generally used to set the threshold for the ideal handheld processing solution's performance. However, an ASIC's potential to achieve optimal levels of performance comes at a cost; an ASIC cannot be reconfigured to perform another application. Thus an ASIC has no flexibility and infinite reconfiguration time. The solution space for an ASIC is shown below in Figure 1.4. Note that none of the vectors have been given values as there can be great variation between designs and the solution space is intended to depict general trends for ASIC solutions. While no precise values have been given, the solution space can be used for the purposes of comparison. Also note that while an ASIC has the potential to achieve optimal levels, the actual selected chip may not. This is indicated by the filled-in region in the figure.

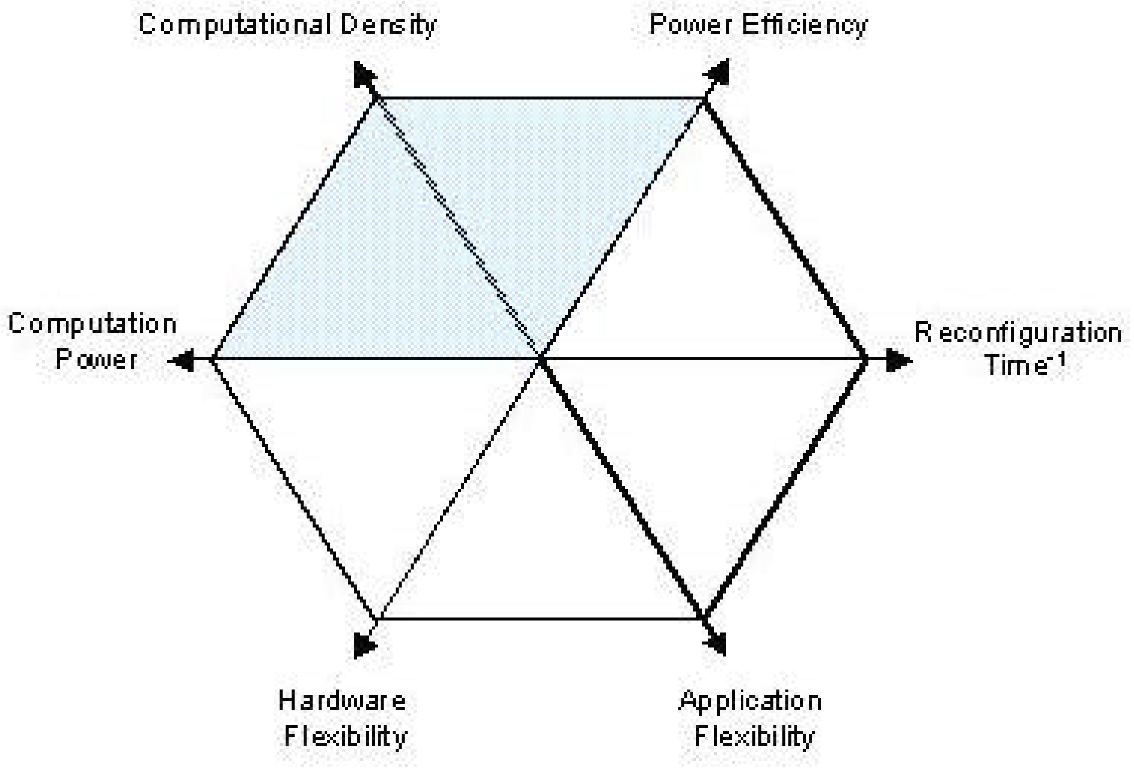


Figure 1.4 ASIC Solution Space

Although ASICs may seem to be a poor fit for software radios, they are still commonly included in handheld software radio designs because they can provide superior performance in a small form factor with a minimum amount of power consumption. Because of this confluence of advantages, current software radio designs still find it necessary to include ASICs to implement frequently used computationally intensive algorithms. Since most software radio designs anticipate an eventual migration away from ASICs, an ASIC is normally encapsulated into a common interface. For instance, the Software Communications Architecture (SCA) specifies the use of a specific form of encapsulation known as *Adapter Classes* to provide an interface to an ASIC capable of being accessed by software components [4].

1.2.2 Microprocessors

A microprocessor (μP) assumes the opposite approach of an ASIC: maximize reconfigurability at the expense of efficiency. A μP interprets instructions stored in memory to perform calculations on inputs and data stored in memory. The results of the calculations may be stored in memory, output, or used to modify the chip's flow of execution. The traditional μP consists of the following components:

- A single functional unit, typically an arithmetic logic unit (ALU), that can be rapidly programmed to execute one of a number of predefined instructions
- Memory which holds instructions and data
- Circuitry for fetching, decoding, and dispatching instructions to the functional unit
- Input and output (I/O) circuitry

In the traditional μP design, these components are organized according to the von Neumann architecture. A block diagram of the von Neumann architecture is shown below in Figure 1.5. Notice that a common bus and common memory is used for both program information and data. Modern μPs have evolved beyond the von Neumann architecture and many now include multiple functional units and specialized coprocessors.

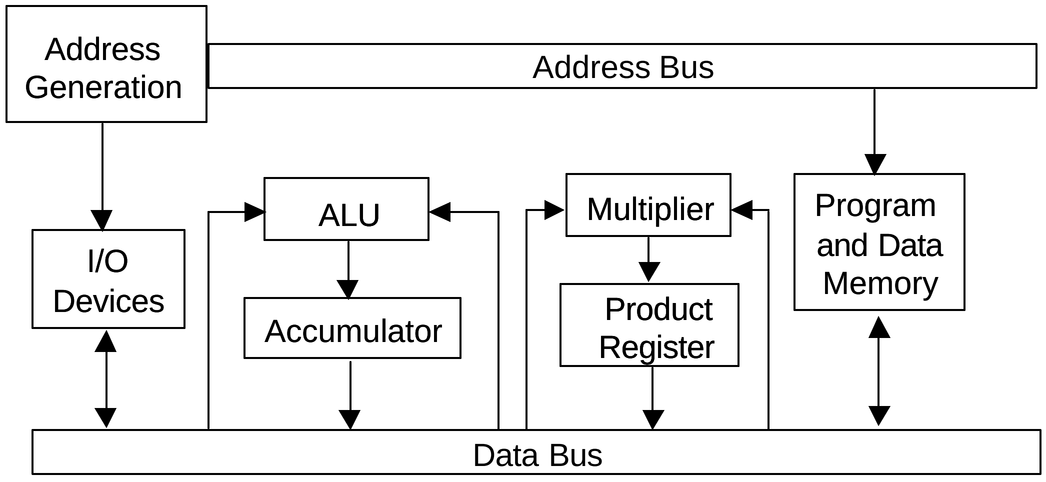


Figure 1.5 vonNeumann Architecture

Due to their design, μP achieve excellent application flexibility and very low reconfiguration times, as a change in application can occur merely by changing the location in memory from which instructions are fetched – potentially occurring in a single clock cycle. Due to the limited number of computational units, the computational power achieved by a μP is highly dependent on the processor's clock rate, and, although it can be considerable, it is generally less than that of an ASIC created with identical fabrication technology. Similarly, a μP's computational density and power efficiency is much less than an ASIC due to the extra circuitry and memory used to provide the μP's application flexibility. Most μPs also lack the ability to alter its hardware. Thus the solution space for a μP can be envisioned as shown in Figure 1.6.

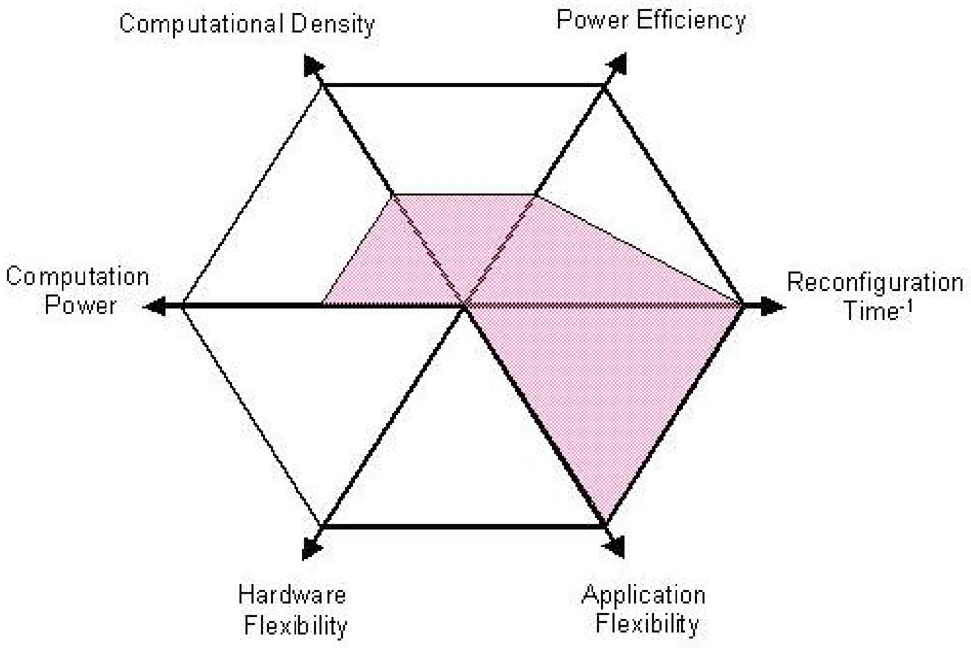


Figure 1.6 Microprocessor Solution Space

1.2.3 Digital Signal Processors

As the primary shortcomings of the μP are attributable to its application independent design, a digital signal processor (DSP) is an attempt to solve these problems by designing a μP that is optimized specifically for signal processing applications.

In signal processing applications, the most common operation is the multiply-and-accumulate (MAC) operation, which is fundamental to any filter implementation. A MAC operation can be characterized by the following equation $Z = \sum_{i=0}^{N-1} X_i Y_i$ where X and

Y are two vectors and *Z* is the result of the dot product of the vectors. In order to facilitate this operation in the shortest period possible, two variables need to be simultaneously loaded, multiplied, and the result accumulated. In the von Neumann architecture, the shared memory space and shared busses for program and data information only permit a single element to be loaded at a time – one variable or one instruction. Since two variables and one instruction (for the ALU) must be dispatched, the von Neumann architecture requires at least three sequential memory accesses to perform one leg of a MAC operation.

The Harvard architecture, shown in Figure 1.7, was developed explicitly to support the MAC operation. In the Harvard architecture, three different independent memory spaces are established and three different data and address busses established – one for programming information, and two for data termed *X* and *Y*. Thus in the Harvard architecture, all of the memory accesses can be performed simultaneously.

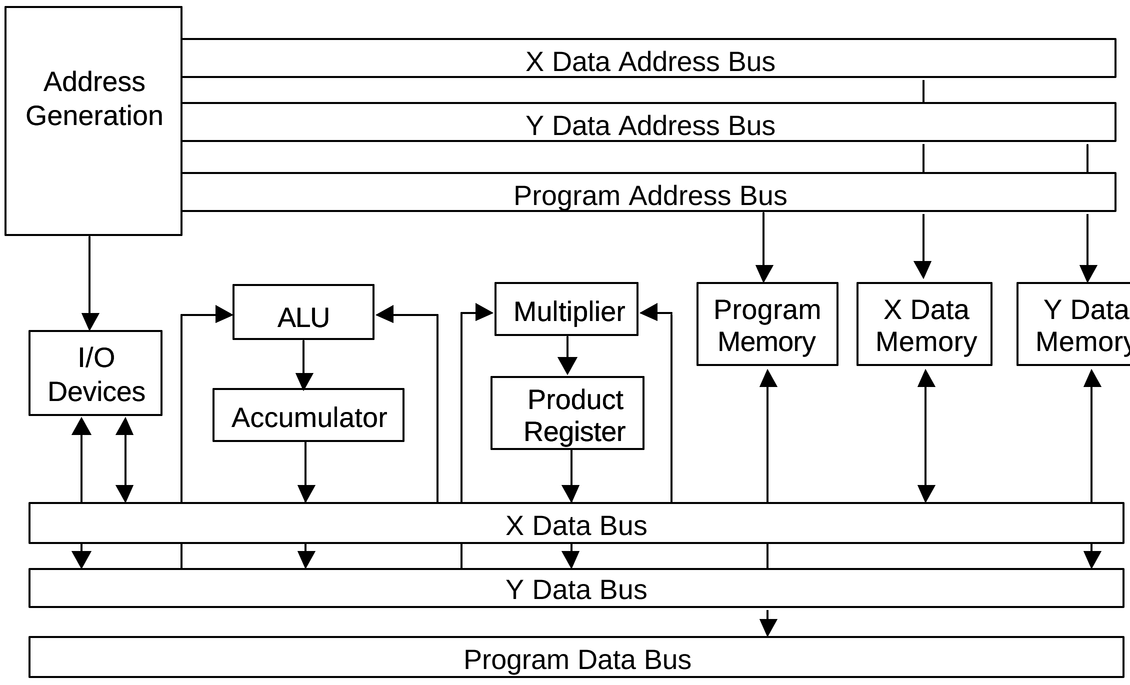


Figure 1.7 Conceptual Representation of the Harvard Architecture

The generalized Harvard Architecture has been modified to include many new features designed to enhance signal processing performance. Some of these features include deeper pipelines, specialized addressing modes such as bit-reversed to facilitate the FFT, very-long instruction word (VLIW), and single instruction multiple data (SIMD). In architectures that utilize VLIW, the DSP is capable of fetching multiple instructions at a time that are executed in parallel. An example of a DSP that uses VLIW is the TMS320C6701 which is capable of fetching and executing eight instructions simultaneously [10]. Architectures that use SIMD have functional units that are capable of treating its operands as a number of smaller independent operands. Then the functional unit performs the operation indicated by the instruction on the smaller independent operands. For instance in a DSP that can exploit SIMD, two 32 bit integers can be loaded that actually consist of four 16-bit words. SIMD is illustrated in Figure 1.8. A DSP that uses SIMD is the ADSP21060 from Analog Devices [9].

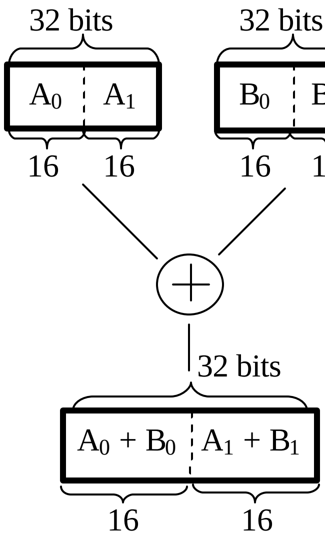


Figure 1.8 Single Instruction Multiple Data Addition

Although a DSP seems significantly different from a μP, in practice, most modern μPs have adopted so many of the DSP's optimizations that it can be difficult to distinguish between what is a μP and what is a DSP. For the purposes of this thesis, a DSP is treated as any processor that has been optimized for signal processing applications; a μP is a processor that is intended for general purpose applications. To emphasize this distinction, the term "General Purpose Processor" (GPP) is used to refer to a μP that is not a DSP for the remainder of this document.

A DSP, then, is a μP that has been tweaked to better perform signal processing applications. For the handheld domain, this results in processors that have slightly better computational density and power efficiency than a GPP, though significantly less than an ASIC. However, as shown in Figure 1.9, a GPP and a DSP achieve similar levels of computational power. Like a GPP, a DSP can be reconfigured in an extremely short period of time, and a DSP has complete application flexibility. A DSP may also have a small measure of hardware flexibility due to its support of differing addressing schemes and SIMD. Thus the solution space for a DSP can be visualized as shown below in Figure 1.9.

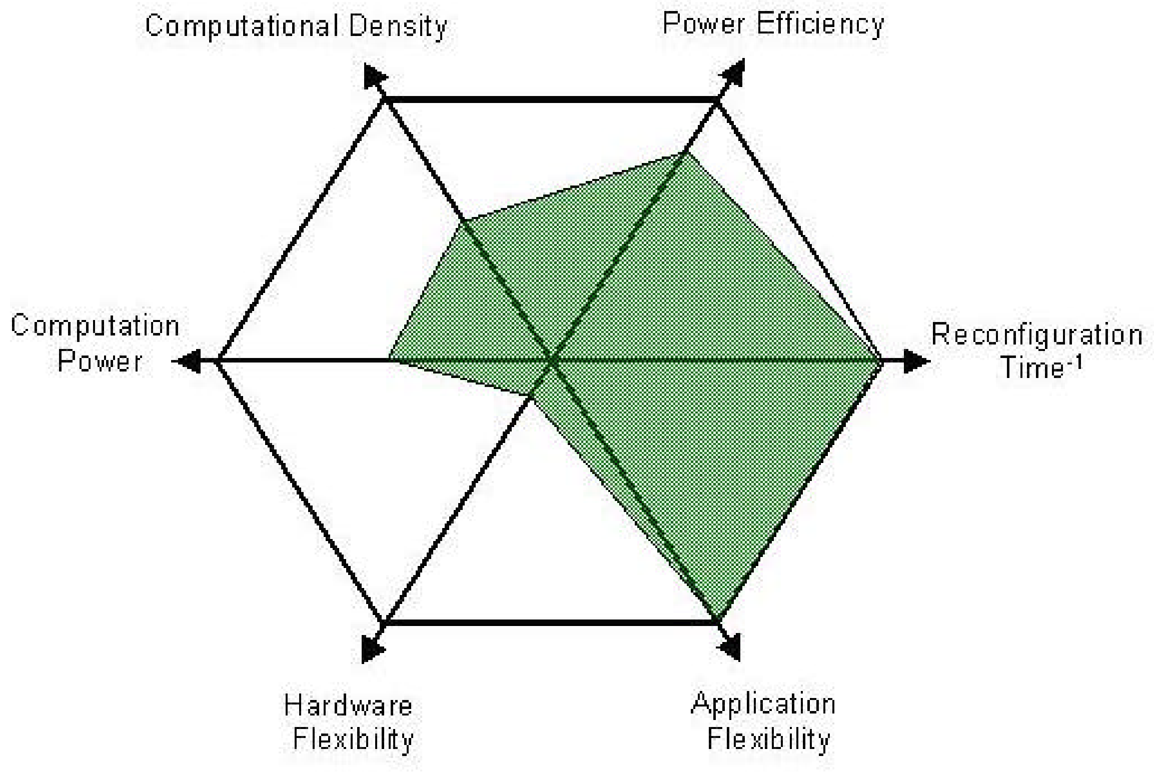


Figure 1.9 DSP Solution Space

1.2.4 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are chips whose hardware functionality can be reprogrammed. This is performed by changing the bit values stored in the memory elements that determine the operation of the components of the fundamental component of a FPGA – the configurable logic block (CLB). A CLB is designed to support arithmetic operations at the bit level that can be scaled to support operations across several bits. A CLB for a XCV1000 FPGA is shown below in Figure 1.10. Notice that this CLB has two primary inputs, or operands, primary and secondary outputs, as well as a simple carry input and output.

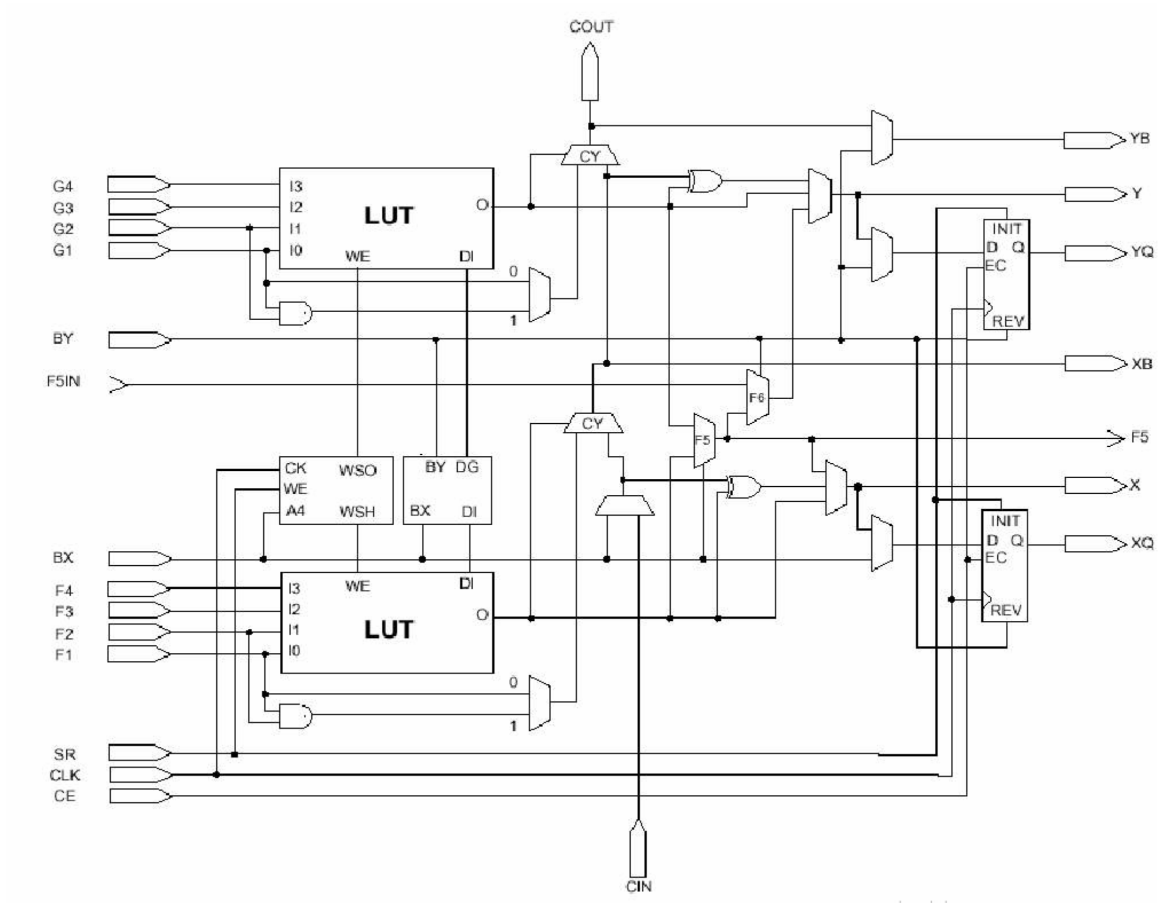


Figure 1.10 Virtex Configurable Logic Block [8]

A single FPGA can contain millions of CLBs [8]. Nominally independent, groups of CLBs can be programmed to work together to perform more complex operations using more bits. Most FPGAs provide direct connections to facilitate communications between adjacent CLBs. Additionally busses managed by switch matrixes are provided to transport information between greatly separated CLBs. The scheme for routing on a Xilinx 4000 FPGA is shown below in

Figure 1.11.

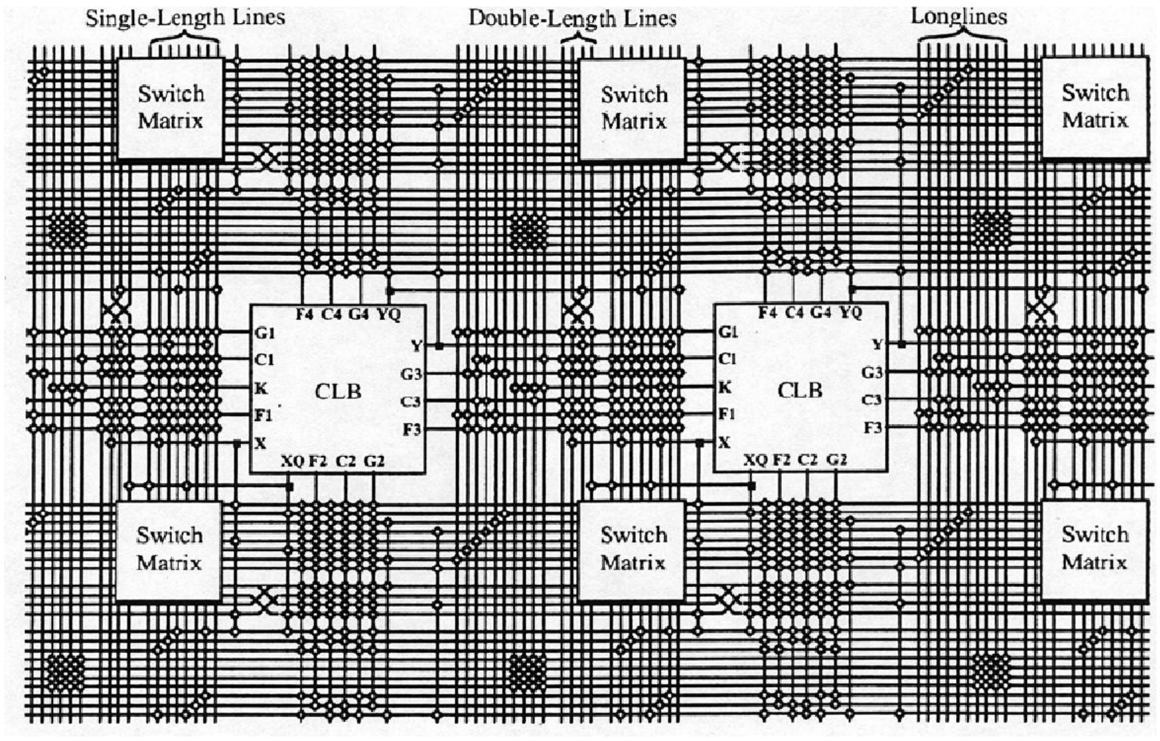


Figure 1.11 FPGA Routing Architecture [12]

This highly flexible structure allows a FPGA to be configured to implement almost any application with hardware that is tuned for the application. Thus, although application independent, a FPGA is able to achieve a computational power much greater than that of a DSP or a GPP. However, notice that for a FPGA a great deal of chip space and resources are allocated for routing signals between CLBs. Thus it cannot achieve the power efficiency nor the computational density of an ASIC. In fact due to the many lengthy cross-chip connections in a FPGA, the power consumption of a FPGA is generally inferior to optimized DSPs and GPPs. Conversely, since a FPGA does not have to support large internal memory caches (external memory is supported) or the overhead associated with address generation and instruction decoding, a FPGA generally has a higher computational density than a GPP or a DSP. Conversely, a modern FPGA requires megabits of information to be completely programmed, and typically requires at least several milliseconds to change operation [5]. Partial reprogrammability would help to alleviate this problem, but most FPGAs do not support this feature. Thus the solution space for a FPGA can be visualized as shown in Figure 1.12.

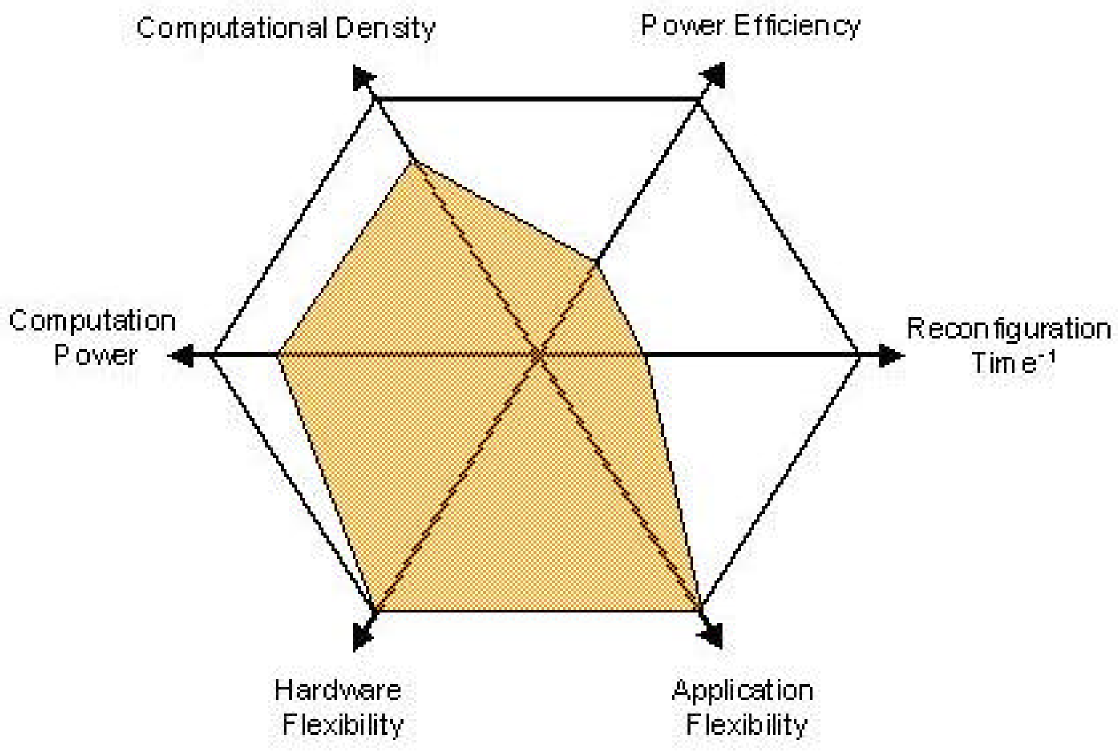


Figure 1.12 FPGA Solution Space

1.3 Custom Computing Machines

Upon examining the solution spaces of the traditional processing solutions, it can be seen that in order to achieve better performance along one axis, a device trades off performance along another axis. Thus none of the traditional processing solutions can adequately meet the demands of the handheld domain. One potential solution is to use a radio architecture that incorporates a mixture of all of these devices. However, this can be a cumbersome solution that may or may not satisfy form factor constraints and introduces difficult device management problems. Thus, it remains desirable to identify a single processor that approximates the performance parameters of the ideal handheld solution. Recently, the Configurable Computing Lab at Virginia Tech has developed a new breed of processors termed Custom Computing Machines (CCM) that holds great promise as a processing solution in the handheld domain. One of the primary goals of this thesis is to examine the applicability of CCM to the handheld domain and to figure out how a CCM solution compares to traditional devices and an ideal handheld solution.

CCMs are similar to traditional processors, but in other important aspects, CCMs represent a significant departure from traditional processing solutions. Conceptually, a CCM is similar to a FPGA. Both primarily consist of reprogrammable hardware blocks – CLBs in a FPGA and functional units in a CCM. In both CCMs and FPGAs, these blocks can be linked together in nearly arbitrary manners to facilitate the implementation of deeply pipelined algorithms or, alternatively, massively parallel instantiations. However, the hardware blocks in CCMs exhibit a much coarser granularity than seen in FPGAs – performing operations on words instead of bits. In the place of the adder and LUTs in the CLB, a CCM's functional unit consists of devices such as ALUs, barrel shifters, and dedicated multipliers. Each of these devices can be optimized for the device's operation. A functional unit from the Stallion Custom Computing Machine is shown below in Figure 1.13. Notice that its primary processing circuitry consists of a 16-bit ALU, a barrel shifter, and a conditional selection block.

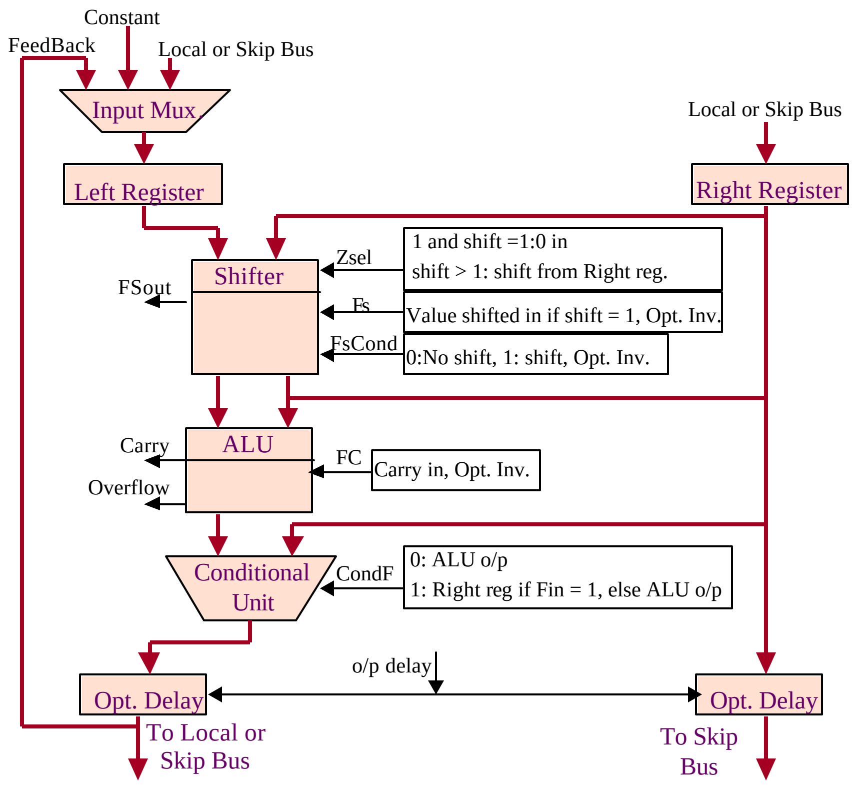


Figure 1.13 Stallion CCM Functional Unit

Interconnection between functional units is accomplished in a manner similar to a FPGA; direct connections between adjacent functional units and extensive busses and switching matrixes provided for connections between widely separated functional units.

1.4 Motivation and Contributions

The primary motivation of this work was to demonstrate the feasibility of CCMs in a software radio, particularly the Stallion CCM, within a layered radio architecture and to form qualitative performance comparisons between CCMs and traditional processing solutions. As part of the work of this thesis, a graphical environment was created for the development of applications on the Stallion CCM. The environment included a graphical programming interface, a programmable simulation of the Stallion CCM developed using object oriented programming methodologies that returns raw results and approximations of power consumption statistics, and a routine to calculate utilization statistics. To demonstrate Stallion's reconfiguration, flexibility, computational power, power efficiency, and role in the Layered Radio Architecture two different receiver structures – a single-user Direct Sequence Spread Spectrum (DSSS) receiver and a Wideband Code Division Multiple Access (WCDMA) receiver – were developed and simulated in the environment.

1.5 Document Organization

The remainder of this document is organized as follows:

- A discussion of material on which this work rests including details of the Layered Radio Architecture and the Stallion CCM as well as an overview of the object oriented programming methodologies used in creating the simulation tools
- A description of the Stallion development environment including a discussion of the simulator, the programming environment, and the statistic generator
- A description of the Single User DSSS receiver implementation and performance
- A description of the WCDMA receiver implementation and performance
- A summary of the results including important conclusions about design tradeoffs and insights that can be made about the solution space a CCM presents for the handheld domain.

Chapter 2

2 Background Material

2.1 Layered Radio Architecture

The Layered Radio Architecture was developed as a part of the dissertation of Dr Srikathyayani Srikanteswara [13]. It is built around two fundamental concepts: a layered architecture and stream based processing and tuned for use with software radios and CCMs.

2.1.1 Layered Architecture

In a layered architecture, functionalities are separated into distinct layers where each layer implements a different level of abstraction of the intended application. Though used for communications networks, a good illustrative layered architecture is the Open Systems Interconnect (OSI) model shown below in Figure 2.1. In the OSI reference model, functionality is clearly separated between layers, and each layer implements a different level of abstraction of the desired communications application.

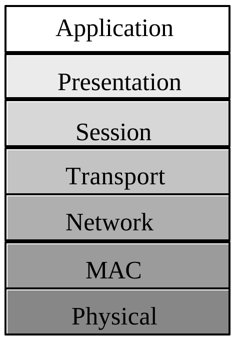


Figure 2.1 OSI Reference Model

The application layer holds the actual application, *e.g.*, voice, email, video, being run over the communications link. The presentation layer is responsible for reformatting the data types native to the application to a standard format usable by the lower layers. The session layer is responsible for managing the end-to-end communications link between two devices without respect to how this is accomplished. The transport layer is responsible for handling the end-to-end messages, again without respect to how this is accomplished. The network layer is responsible for data routing and addressing and must be concerned with the actual topology of the network. The MAC layer is responsible for error correction and multiple access. The physical layer is responsible for implementing the immediate link in the network.

Further in a good layered architecture, the interfaces between each layer are well-defined, and the actual operation of each layer hidden from the other layers. The combination of these factors results in layers that can operate independently of the operation of the other layers. Thus with a good layered architecture, the layers can be developed independently and modifications can be made to a layer without impacting the performance of the other layers. This has a tremendous benefit in a software radio where the layers may change numerous times over the lifetime of the radio.

2.1.2 Stream Based Processing

Also exploited in Stallion, stream based processing extends the concept of pipelining by allowing the pipeline to operate on a continuous stream of packets. Further in stream based processing, the incoming packets also contain information that can be used to dynamically reconfigure the operation of each stage in the pipeline. The stream based processing concept is conceptually illustrated below in Figure 2.2. Notice that there is a single path for both programming information and data. Each element in the path is capable of interpreting the programming instructions and accordingly reconfiguring its operations.

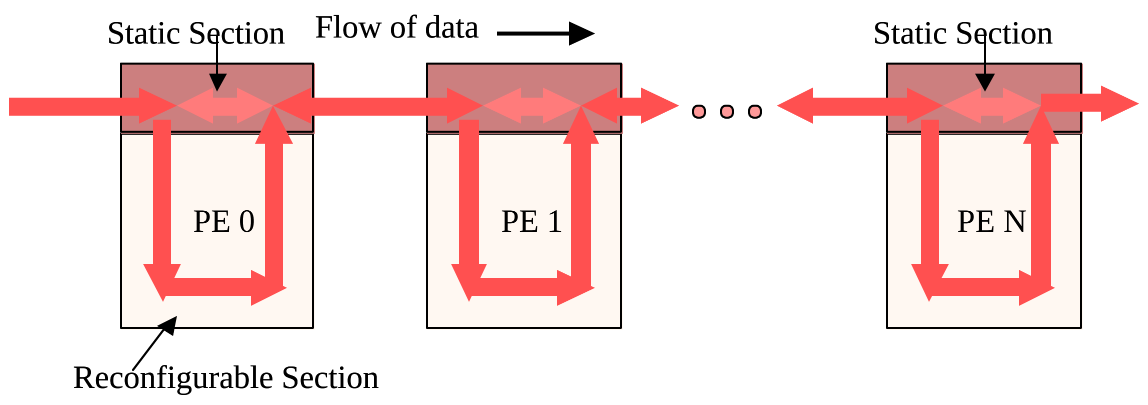


Figure 2.2 Stream Based Processing Example [13]

Ideally, each element in the stream is capable of interpreting the packet to determine whether to process the packet, to use the packet for configuration, or to leave the packet unchanged. The element then constructs packets for the next element in the pipeline. To support changes in configuration and for more advanced processing, each processing element also should have dedicated memory. An ideal stream processing element is shown below in Figure 2.3.

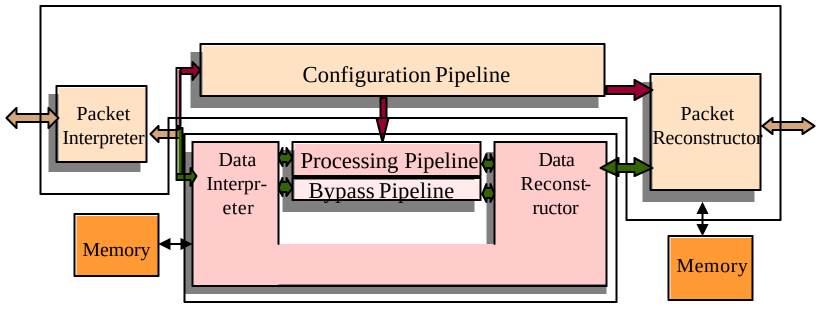


Figure 2.3 Stream Processing Element [13]

2.1.3 Layered Radio Architecture

Shown below in Figure 2.4 there are four layers in the layered radio architecture: application layer, soft radio interface (SRI) layer, configuration layer, and the physical layer. Whereas the OSI reference model implements a communications link, the layered radio architecture implements the baseband processing aspects of a soft radio. The application layer, which is conceptual and not strictly part of the Layered Radio Architecture, is responsible for the determining which wireless protocol to run dependent on the application and network availability. The application layer also conceptually generates the data for the baseband processing whether from the ADC or internally. The SRI layer is responsible for formatting the data and messages received from the application layer into stream packets and also for determining the major constituents (physical layer configurations) and their order of execution in order to implement a protocol specified by the application layer. The configuration layer is responsible managing the streams of programming information and data needed to actually implement the baseband processing of each configuration. Conceptually, each layer also traps various kinds of errors.

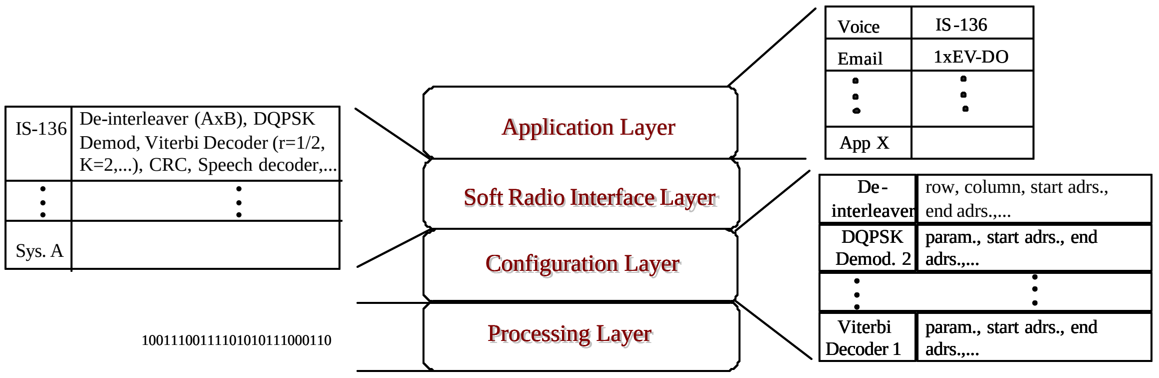

Voice
IS-136
Email
1xEV-DO
App X
IS-136
De-interleaver (AxB), DQPSK Demod, Viterbi Decoder (r=1/2, K=2,...), CRC, Speech decoder,...
Sys. A
Application Layer
Soft Radio Interface Layer
Configuration Layer
Processing Layer
1001110011110101011100011 0
De-interleaver
row, column, start adrs., end adrs.,...
DQPSK Demod. 2
param., start adrs., end adrs.,...
Viterbi Decoder 1
param., start adrs., end adrs.,...

Figure 2.4 Layered Radio Architecture

To further support the layered architecture, well-defined packet-based interfaces were developed. Communication between layers is implemented using streaming packets which would embed data, configuration information, and control signals. As an example of the structure used to build the interlayer interfaces, Figure 2.5 shows the interlayer communications packet used by the SRI layer to send messages to the configuration layer for a packet intended to modify a configuration stored in configuration memory. In this hierarchical representation of the 64-bit packet where the "meaning" of certain fields are dependent on the values in other bit fields. When subsequent fields are dependent on the value of a field, the bit fields associated with a particular value are displayed in a tree that extends from an oval that represents the value. For instance if bits 51-50 indicate that the configuration packet type is for the configuration header, bits 49-36 are discarded while bits 35-0 hold the configuration header information. However, if bits 51-50 indicate a stream packet, then bits 49-0 carry information for a particular stream packet. To support this streaming approach, both the configuration layer and the SRI layer are implemented like a stream processing element.

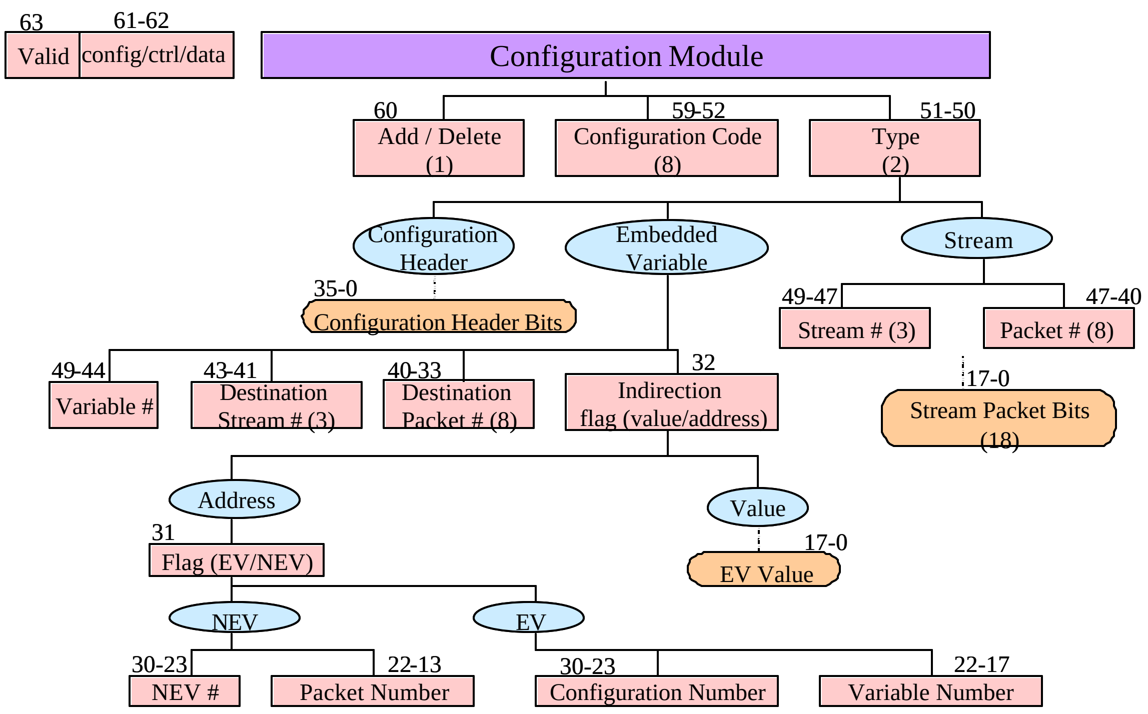


Figure 2.5 SRI to Configuration Layer Communication Packet

2.1.4 Configuration Layer Implementation

The implementations of the configuration layer and the SRI layer are conceptually similar. Both have separate memory spaces for configuration information and data and make extensive use of state machines. This section describes the details of the configuration layer which were used to serve as the basis for the controller chip simulation developed as part of this thesis. The configuration layer consists of the following components: data and configuration memory, memory controller modules, a layer controller module, a stream router, and interfaces with the SRI layer and the physical layer. A block diagram of the configuration layer is shown below in Figure 2.6.

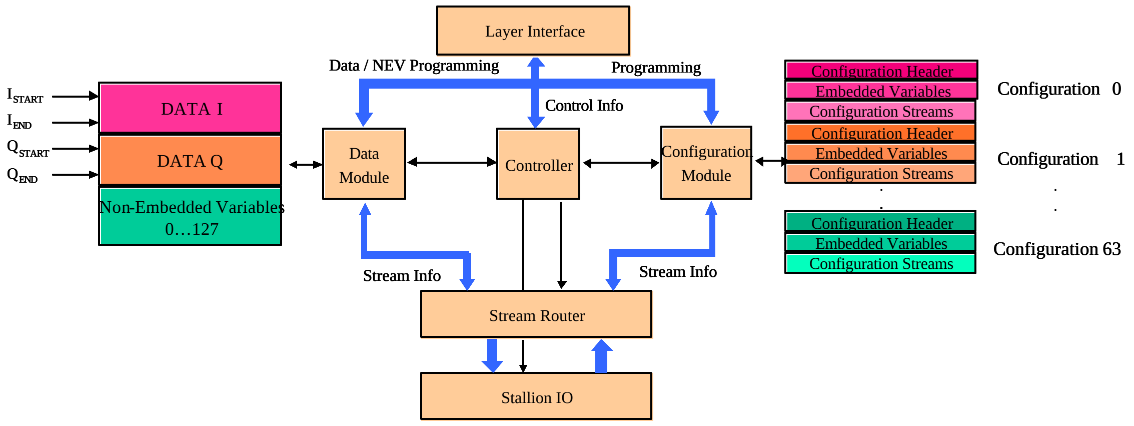


Figure 2.6 Configuration Layer Block Diagram

The data memory block is segmented into three blocks: Data I, Data Q, and Non-Embedded Variables (NEV). There are independent data pointers that are used to write data in (I_{end}, Q_{end}) and read data out from data memory (I_{start}, Q_{start}). The data module is responsible for managing the data memory, keeping track of which element of a NEV is active, and passing this data as packets to the stream router. As part of the management of the data, the data module would also generate error/status messages for about how much space data memory was occupying. This information could be used by higher layers to potentially adjust the sampling rate, when possible. To abstract away the exact addresses from the data module, the data module has a data memory manager responsible for interpreting instructions that specify loading or fetching data or a non-embedded variable. Once instructed by the data module, this memory manager has the capability of continuously loading or storing data for six different streams.

NEVs are intended for use as storage for the temporary variables passed between configurations. Since Stallion operates best with long streams of data, all NEV were organized as arrays. The data module also contains six counters that are initialized by information received from the control module and are used to specify the current address to be used by each of the six possible streams. Thus it is assumed that data for each stream is stored in a sequential manner.

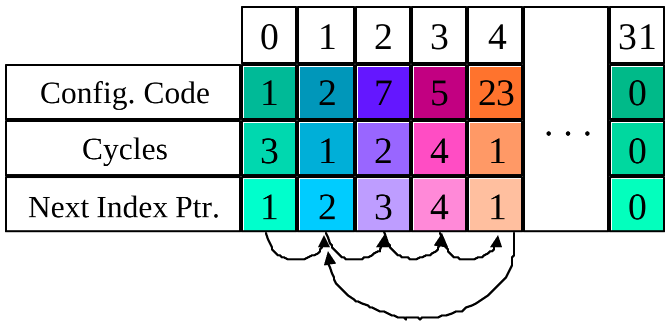
The configuration memory block is segmented into space for 64 Stallion configurations. Each configuration is divided into a configuration header, embedded variables, and configuration streams. The configuration header contains information that describes the higher level characteristics of a configuration. These include the specification as to which of the six streams are used, which are input streams, which are output streams defined in a six bit vector, the number of embedded variables needed by the configuration, how much to advance the data start pointers after configuration completion, the number of cycles required to program a configuration and the number of execution cycles for the configuration.

The embedded variables are intended to facilitate fine-grained changes to the operation of a configuration through the alteration of constants within a configuration. Depending on the configuration, a change in a particular embedded variable can be as simple as a change in spreading code for a despreader or cause a configuration to operate in an entirely different manner. These variables can be modified by commands issued from the SRI layer or, when configured as an indirect embedded variable pointing to a NEV, by the results produced by a particular configuration. Thus embedded variables support both directed alterations to a configuration as well as run-time adaptation of configurations in response to results produced by a configuration.

Each stream consists of a header and programming information. Depending on the context of the stream (as specified by the configuration header), the stream header is interpreted as an input stream header or an output stream header. If an input stream, the header indicates the source of the stream; this can be data I, data Q, or an NEV. If an output stream, the header indicates the sink for the stream; this can be an NEV, the SRI layer, or an embedded variable. Additionally, an output stream also includes information on a packet-by-packet basis as to whether to deliver the packet to the destination or to discard the packet. This approach was adopted as a solution to manage the potential differing latencies between streams as well as the possibility that a particular configuration may not continuously generate outputs.

The configuration module is responsible for fetching this information and passing the control information, to the control module. It also services the embedded variables and organizes packets for the stream router to indicate packet destination on a packet by packet manner through the aid of a configuration memory manager. It is the responsibility of the configuration memory manager to abstract away the actual addresses being used to store the details of a configuration. The configuration memory manager also supports modes for programming where once instructed to begin, it will continue to return programming packets. This reduces the number of messages that the configuration module has to send to its memory manager.

The control module is responsible for managing the configurations and the routing of information within the configuration layer. It collects error messages from the data module and the configuration module. It also generates an error message whenever an invalid configuration is requested for execution. Shown below in Figure 2.7, the control module is built around the programmable state machine which is filled according to instructions issued by the SRI layer. The state table contains the following information: the configuration code to specify one of the configurations stored in configuration memory, the number of times the entire configuration needs to be run (indicated by the Cycles row), and a pointer which is used to determine which column (and thus which configuration) to execute after the current one is complete. The inclusion of the Cycles row facilitates the reuse (typically with a different set of data) of a configuration without reprogramming. An example of when this is useful is with a sliding correlator being used for acquisition (refer to Chapter 4 for a configuration that uses such a configuration) where the same operations need to be performed repeatedly with slightly different sets of data. The inclusion of the Next Index Pointer allows the configuration layer to implement near arbitrary sequences of configurations as well as dynamically paging in additional configurations as instructed by the SRI layer.



	0	1	2	3	4		31
Config. Code	1	2	7	5	23	. . .	0
Cycles	3	1	2	4	1		0
Next Index Ptr.	1	2	3	4	1		0

Figure 2.7 Control Module Programmable State Machine

The stream router serves a collection point for the various packets from the data and configuration modules. As specified by the packets received from the configuration module during the initialization of a configuration, the stream router sends the different packets to the correct destinations within the configuration layer or to the processing layer. The stream router supports the simultaneous operation of up to six half-duplex streams of information.

The following describes the steps used by these components to load and manage a new configuration into the processing layer (Stallion). This is also illustrated in the Unified Modeling Language (UML) sequence diagram shown below.

1. The control module state table indicates a new configuration is to be loaded.
2. The control module halts the operation of the data module, configuration module and the stream router, ending normal operation. This also terminates the operation of the memory managers. The validity of the configuration is verified. If valid, initialization of the specified configuration commences, else an error packet is sent to the SRI layer.
3. If valid, the control module sends a message to the configuration module to initialize the specified configuration, unhalting the configuration module.
4. The configuration module fetches the configuration header information. This information is used to update counters in the control module and to configure the direction of the streams in the stream router.
5. The configuration module services the embedded variables. When a direct embedded variable is encountered, the value stored is written into the appropriate configuration stream packet. If instead, the embedded variable is indirect, then the value is fetched if internal to configuration memory, else the configuration module issues a message to the control module requesting the required data (such as from a NEV in the data module).
6. The configuration module fetches the stream header information for the streams. This information is then used to configure counters within the configuration module and the data module (through messages sent through the control module). This signals the end of the configuration initialization.
7. The control module then instructs the configuration module to begin loading the programming packets. The stream router is placed into programming mode wherein all outputs from Stallion are discarded. The configuration module proceeds to send configuration packets to the stream router which are subsequently sent to Stallion.

8. After the number of configuration cycles (n in the diagram) specified in the configuration header has passed (this may or may not correspond with the complete programming of all streams), normal processing begins. Although not depicted in the diagram, this corresponds to the control module unhalting the data module; the configuration module sending output stream information to the stream router, and the data module fetching or writing data as specified by each stream's counter and the stream configuration information it had earlier received and stored in the data memory manager.

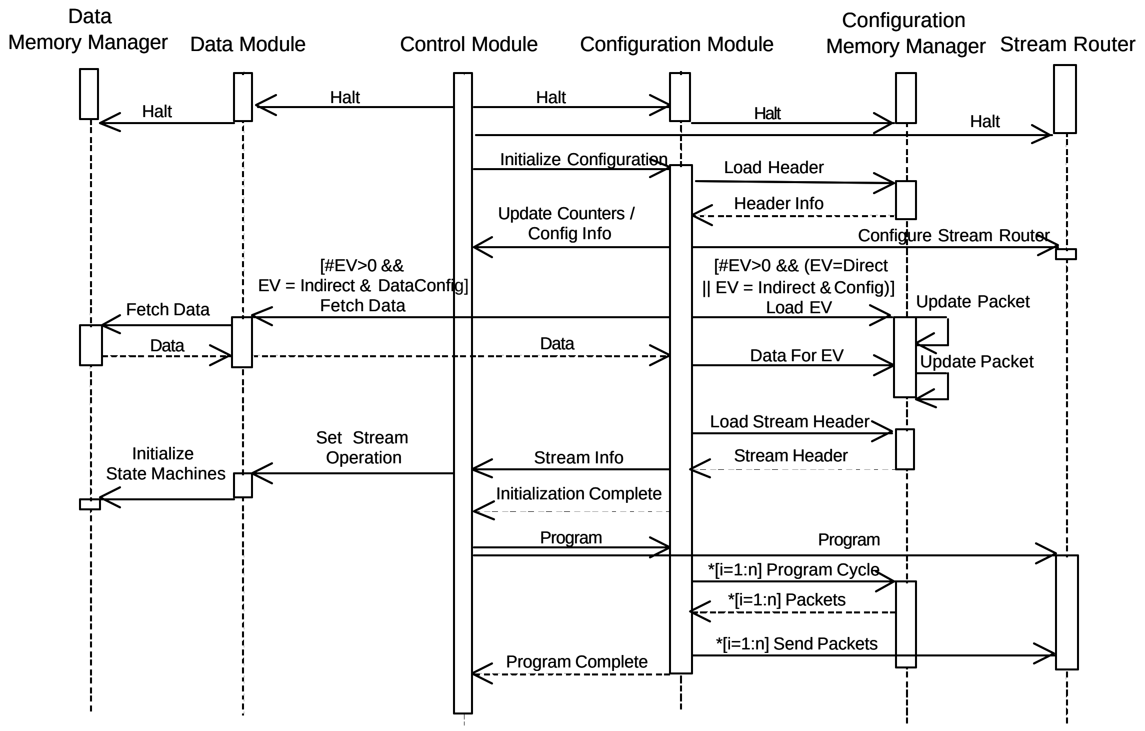


Figure 2.8 Interaction Diagram when a New Configuration is Loaded

Thus it is seen that the configuration layer expects to be rapidly reconfiguring the physical layer in order to implement the sequence of configurations that comprise a waveform, in a process known as *hardware paging*. Hardware paging is the process of rapidly and sequentially reconfiguring hardware to implement an application where each reconfiguration implements a different algorithm. Although hardware paging does not give the optimal performance when partial reconfiguration is possible, hardware paging was dictated by the combination of a relatively low bandwidth for its processing power, and short reconfiguration time offered by the processor chosen for the physical layer: Stallion.

2.2 Stallion

2.2.1 Overview

The Stallion CCM is an extension of the Colt CCM originally designed by Dr Ray Bittner [14] and is used to implement the physical layer of the Layered Radio Architecture. On Stallion, sixty interconnected functional units and four dedicated multipliers provide the processing resources 16-bit integers. These processing resources are arrayed into two meshes of thirty functional units and two multipliers. A data interface to Stallion is provided through the use of six bi-directional data ports. A fully interconnected crossbar provides data flow between these elements. These elements can be visualized as shown below in Figure 2.9. Note this is a conceptual representation, not a depiction of the physical layout.

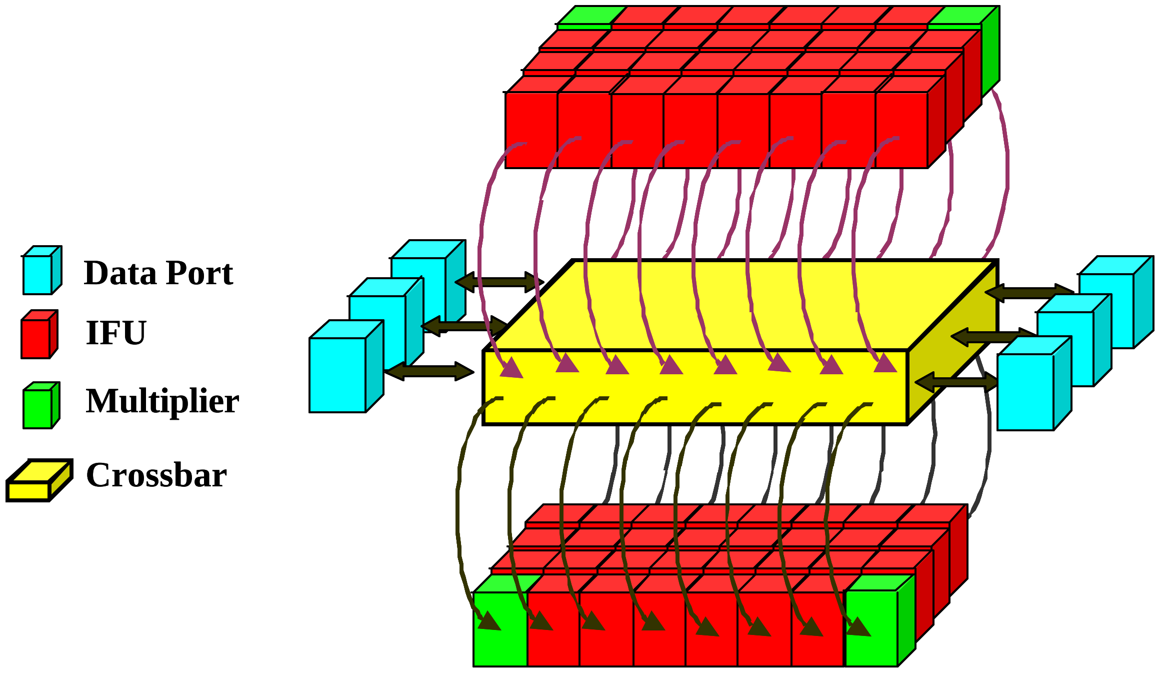


Figure 2.9 Stallion Functional Diagram

Processing on Stallion is performed using a concept known as *stream-based processing.* In stream-based processing streams (a run of consecutive packets) of programming information and data are passed through pipelined processing elements. When a packet containing programming information arrives at the desired element, the element assumes the specified configuration. If the packet contains data, then the element processes the data according to its configuration. Stream-based processing reduces the complexity of a processor by eliminating the need for separate paths for programming and data information. Stream-based processing also lends itself well to the construction of deep pipelines – a useful technique for maximizing the efficiency of computational resources.

For Stallion the packet consists of three components: a bit to indicate whether or not the packet is a programming packet, a bit to indicate whether or not the packet is valid, and a 16-bit payload. This is illustrated below in Figure 2.10. In general in stream based processing, when a packet is invalid the packet is discarded. However, the components in Stallion can be programmed to respond to the packet in a variety of manners; most commonly to convey control information.

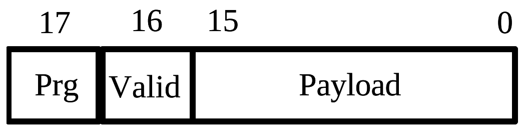


Figure 2.10 Stallion Stream Packet

To further facilitate the programming process, Stallion couples the concept of stream-based processing with *wormhole runtime reconfiguration*, illustrated in Figure 2.11. In wormhole runtime reconfiguration, programming packets also contain the information about the directions where processing elements should forward future programming information while in programming mode. Using this method, programming streams (“worms”) can “chew” a programming path through the interconnected processing elements. Note that this path need not be the same as the path eventually used for data flow. Wormhole runtime reconfiguration further extends the stream based processing concept as the stream itself contains all of the information necessary for programming. Wormhole runtime reconfiguration and stream based processing also simplify the scaling of an algorithm across multiple devices since when appropriately connected, the processing elements on other chips are just another element in the pipeline. It also facilitates independent reprogramming of parts of a processor as the programming independence inherent to wormhole runtime reconfiguration allows for one stream to be programming while another is processing. All of these benefits enhance a processor's capabilities.

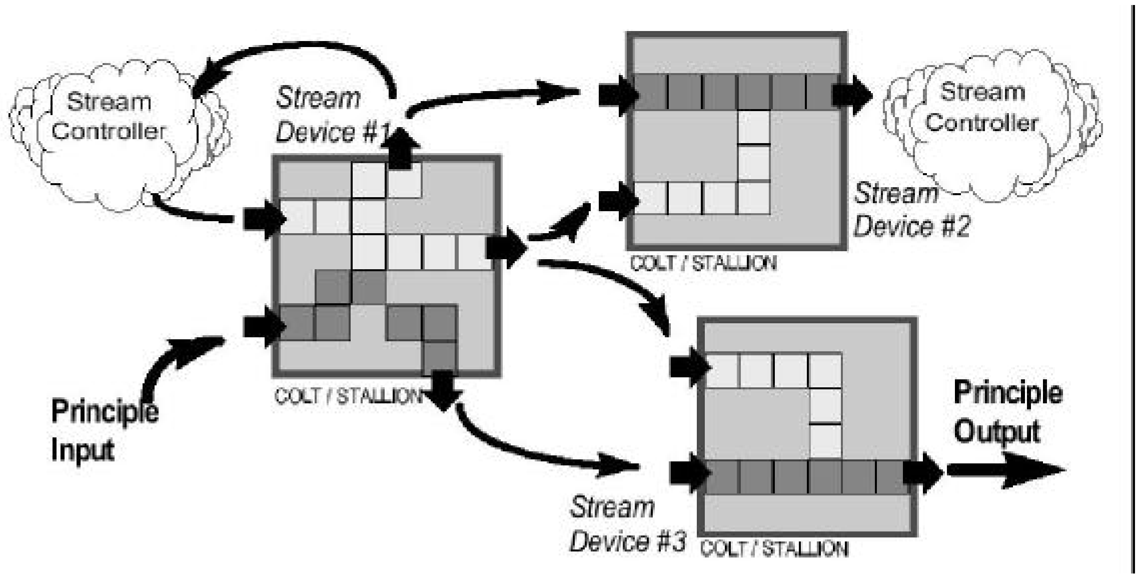


Figure 2.11 Wormhole Runtime Reconfiguration Concept [14]

Since the completion of work on this thesis, fabrication of the Stallion processor has been completed. It measures some 1.675 inches per square side and operates with a clock speed of 50 MHz and a voltage of 3.3V. In the report of its layout and implementation, Stallion was roughly estimated to consume 0.7 W of power [15]. As part of this thesis, this estimate is refined to show how the level of power consumption is dependent on an application's percentage of resource utilization and the randomness of its data.

2.2.2 Mesh

The computational elements of Stallion are organized into two meshes. Each mesh is arrayed in a 4 x 8 grid of two multipliers and thirty functional units. Each mesh is connected to the crossbar by sixteen inputs at the "top" of the grid and eight outputs on the "bottom" of the grid. Data may flow in any direction within the mesh, but in general flows from top to bottom due to the placement of inputs and outputs as shown in Figure 2.12. This is implemented through the use of local connections between mesh units and mesh-wide busses known as skip busses. The two multipliers and six of the functional units are placed at the top of the mesh. This arrangement facilitates the MAC operation, which requires a multiplier precede an ALU in a MAC pipeline.

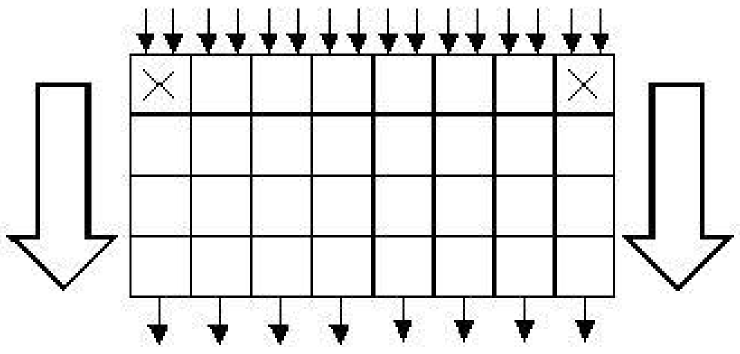


Figure 2.12 Mesh Organization and Generalized Data Flow

2.2.2.1 Mesh Addressing Scheme

The addressing scheme is similar to the addressing scheme which was employed in Colt [14] [17], the predecessor to Stallion. However, Colt only included 15 functional units and a single multiplier as opposed to the 60 functional units and four multipliers of Stallion. Thus Stallion's internal addressing scheme is a repeated version of Colt's scheme as shown below in Figure 2.13. Note that multipliers are not actually assigned an address. Addresses for components within each mesh are then assigned as shown in Table 2.1 according to the row (A-D) and column (1-4) of each mesh element. Note that the addressing scheme repeats within each mesh.

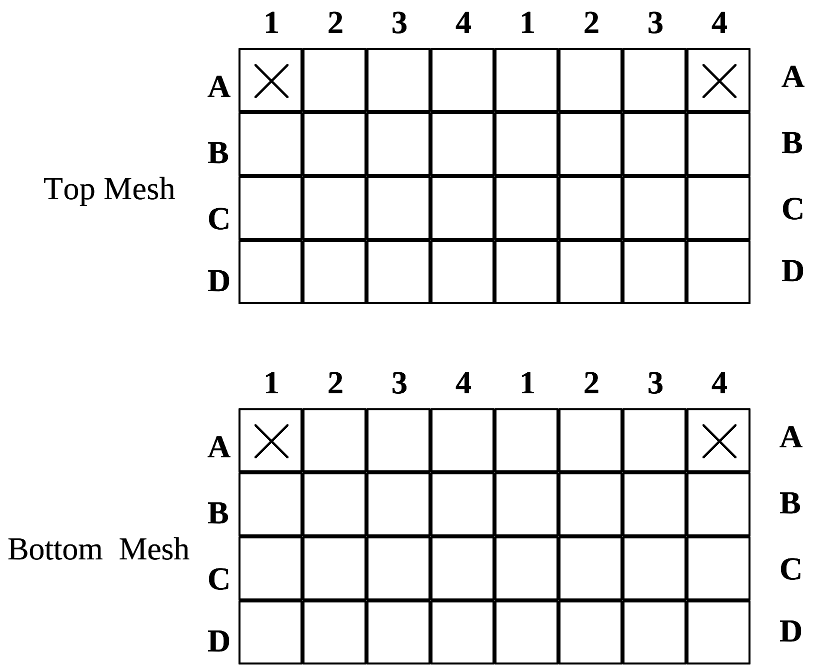


Figure 2.13 Stallion Mesh Addressing Scheme

	1	2	3	4
A		40	41	42
B	43	44	45	46
C	47	48	49	50
D	51	52	53	54

Table 2.1 Mesh Address Assignments

2.2.2.2 Functional Unit

Shown below in Figure 2.14, the functional unit (FU) is the primary processing element of the mesh. It performs calculations with a barrel shifter and an ALU on inputs held in two 16-bit registers. The inputs can be chosen from external sources, an internally generated constant, or feedback or the FU's output the previous cycle (this facilitates the MAC operation). The final output can be delayed for a single cycle to help satisfy latency constraints. The exact calculation performed by a FU can be dynamically influenced by combinations of control signals designated as Flag1 and Flag2. These control signals can be generated internally or externally in response to data, internal processes, or programmed constants.

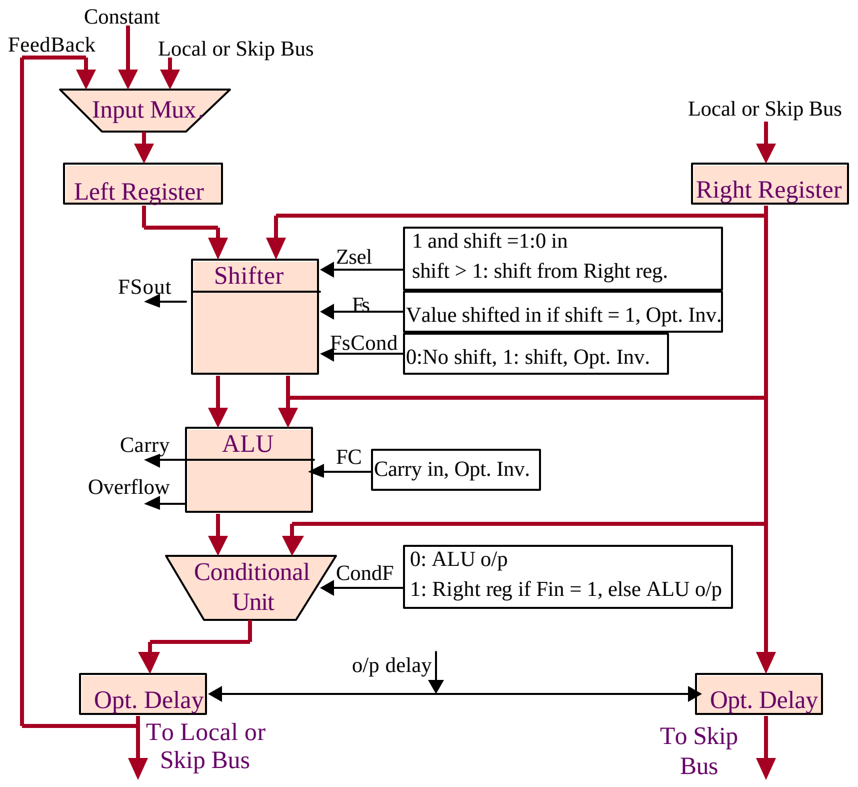


Figure 2.14 FU Functional Diagram

2.2.2.2.1 *Functional Unit Programming Options*

Most of the components within the functional unit can be configured to perform a number of different tasks or to use a number of different sources as an input. The following describes the programmable options for each component and the associated programming bits.

Input Data Register Sources

The programmable options for the components associated with the input data registers are listed below in Table 2.2. Note that the feedback enabled flag and the left register flag override the input source. Also note that it is possible to clear the left register through the use of Flag2; this is particularly useful when using the functional unit as an accumulator.

Left Data Source	Local North	000	Skip North	100
	Local East	001	Skip East	101
	Local South	010	Skip South	110
	Local West	011	Skip West	111
Feedback Enabled	False	0	True	1
Right Data Source	Local North	000	Skip North	100
	Local East	001	Skip East	101
	Local South	010	Skip South	110
	Local West	011	Skip West	111
Left Register Flag	Any Data	00	Only Valid	10
(Input Mux)	Zero if Flag2 = 1	01	Constant	11
Constant Valid Bit	False	0	True	1

Table 2.2 Input Data Register Programming

Data Skip Bus Inputs

The programmable inputs for the data skip busses are listed below in Table 2.3. Note that each skip bus must flow through the functional unit, thus only one of the opposing skip busses (north or south, east or west) is outputting data, and thus the programmable sources is only relevant for one of the skip busses. This direction of data flow is indicated by two direction flags, NS Dir Flag and EW Dir Flag.

South Skip Bus	Skip Bus to West	00	Aux Output	10
	Bus Output	01	Skip Bus to North	11
North Skip Bus	Skip Bus to East	00	Aux Output	10
	Bus Output	01	Skip Bus to South	11
East Skip Bus	Skip Bus to South	00	Aux Output	10
	Bus Output	01	Skip Bus to West	11
West Skip Bus	Skip Bus to North	00	Aux Output	10
	Bus Output	01	Skip Bus to East	11
NS Dir Flag	North	0	South	1
EW Dir Flag	East	0	West	1

Table 2.3 Data Skip Bus Input Programming

Shifter

The programmable inputs for each of objects associated with the shifter is listed below in Table 2.4. Note that when the shifter shifts left by one or right by one, the shifted in value is determined by the FS flag. If Flag 1 or Flag 2 are programmed appropriately, this can be used for a sign extension shift. When shifting by more than one position, the ZSEL Input is used. This facilitates scaling after a multiplication operation which produces a 32-bit output which requires storage across two registers.

Shift Operation	Left 1	000	Right 1	100
	Left 2	001	Undefined	101
	Left 3	010	Undefined	110
	Left 4	011	Undefined	111
Shift Condition	0	00	Flag 2	10
	Flag 1	01	FN	11
Invert Shift Condition	False	0	True	1
ZSEL Input	Right Register	0	zeros	1
FS Input	0	00	Flag2	10
	Flag1	01	FN	11
Invert FS	False	0	True	1

Table 2.4 Associated Programmable Shifter Components

ALU

The programmable options associated with the ALU are given below in Table 2.5. Note that the ALU is a Propagate – Generate – Result sixteen bit ALU as described in [14]. To perform many operations (increment, shift, negate) it is also necessary to program the P, G, and R registers and the carry in bit.

Operations	Defined by 4-bit P, G, and R registers			
Carry In	0	00	Flag2	10
	Flag1	01	FN	11
Invert Carry In	False	0	True	1

Table 2.5 ALU Programmable Options

Flag Skip Bus Inputs and Direction Flags

The programmable inputs for the data skip busses are listed below in Table 2.6. Note that each skip bus must flow through the functional unit, thus only one of the opposing skip busses (north or south, east or west) is outputting data, and thus the programmable sources is only relevant for one of the skip busses. This direction of data flow is indicated by two direction flags, NS Dir Flag and EW Dir Flag.

Flag 1 Skip Bus East	Skip Bus to South	00	F1 Output	10
	Skip Bus to West	01	F2 Output	11
Flag 1 Skip Bus North	Skip Bus to East	00	F1 Output	10
	Skip Bus to South	01	F2 Output	11
Flag 1 Skip Bus South	Skip Bus to West	00	F1 Output	10
	Skip Bus to North	01	F2 Output	11
Flag 1 Skip Bus West	Skip Bus to North	00	F1 Output	10
	Skip Bus to East	01	F2 Output	11
Flag 2 Skip Bus East	Skip Bus to South	00	F1 Output	10
	Skip Bus to West	01	F2 Output	11
Flag 1 Skip Bus North	Skip Bus to East	00	F1 Output	10
	Skip Bus to South	01	F2 Output	11
Flag 1 Skip Bus South	Skip Bus to West	00	F1 Output	10
	Skip Bus to North	01	F2 Output	11
Flag 1 Skip Bus West	Skip Bus to North	00	F1 Output	10
	Skip Bus to East	01	F2 Output	11
Flag 1 NS Dir Flag	North	0	South	1
Flag 1EW Dir Flag	East	0	West	1
Flag 2 NS Dir Flag	North	0	South	1
Flag 2 EW Dir Flag	East	0	West	1

Table 2.6 Flag Skip Bus Programming

Flag1 and Flag2 Input and Output Sources

Table 2.7 lists the various programming options for Flag 1 and Flag 2, two internal control signals. The flag inputs are used internally to the functional unit; the flag outputs are transported by the local busses and potentially by the skip busses. Note that both flag inputs can be set to the valid bit and sign bit <16:15> of either input registers.

Table 2.7 Flag Programmable Options

Flag 1 Input	0	00000	RightReg<16>	10000
	1	00001	Flag1LocalN	10001
	LeftReg<0>	00010	Flag1LocalE	10010
	LeftReg<1>	00011	Flag1LocalS	10011
	LeftReg<15>nor<14>	00100	Flag1LocalW	10100
	LeftReg<15>nor<14><13>	00101	Flag1SkipN	10101
	LeftReg<15>nor<14><13><12>	00110	Flag1SkipE	10110
	LeftReg<15>	00111	Flag1SkipS	10111
	LeftReg<16>	01000	Flag1SkipW	11000
	CondOutput<15>	01001	Undefined	11001
	FS	01010	Undefined	11010
	FSOut	01011	Undefined	11011
	ALU Output<15>	01100	Undefined	11100
	ALUOverflow	01101	Undefined	11101
	CarryOut	01110	Undefined	11110
	RightReg<15>	01111	Undefined	11111
Flag 2 Input	0	00000	LeftReg<16>	10000
	1	00001	Flag2 LocalN	10001
	RightReg<0>	00010	Flag2 LocalE	10010
	RightReg<1>	00011	Flag2 LocalS	10011
	RightReg<2>	00100	Flag2 LocalW	10100
	RightReg<3>	00101	Flag2 SkipN	10101
	RightReg<4>	00110	Flag2 SkipE	10110
	Not RightReg<15>	00111	Flag2 SkipS	10111
	Not RightReg<16>	01000	Flag2 SkipW	11000
	RightInpReg valid zero	01001	Undefined	11001
	FS	01010	Undefined	11010
	FSOut	01011	Undefined	11011
	ALU Output<15>	01100	Undefined	11100
	ALU Overflow	01101	Undefined	11101
	CarryOut	01110	Undefined	11110
	LeftReg<15>	01111	Undefined	11111
Flag 1 Output	Flag1 Input	0	FN Output	1
Flag 2 Output	Flag2 Input	00	ALU Overflow	10

	FS Out	01	Carry Out	11

FN Flag

The FN flag is an internally available flag that is used to facilitate arbitrary combinations of F1 and F2 as specified by the FN LUT (implemented as a 4x2 MUX). The options for generating these flags are listed below in Table 2.8.

FN LUT	4 bit input			
FN Output	F1	0	F2	0
FN Valid	1	0	FN LUT	0

Table 2.8 FN Generation Options

Other

The remaining programmable objects are listed in Table 2.9. The Valid Bit Mux determines the valid bit attached to the data output from the functional unit (Output<16>). It is also possible to indicate which directions the functional unit should send programming information after it has been programmed. The Only Program IFU flag is used to indicate that the functional unit will be needed to pass on programming information, but its current configuration should be retained. This has the potential to reduce programming time and provides some finer grained programming resolution. The delay elements can be used to delay output by a single cycle, potentially useful for maintaining timing between streams. Note that the flag delays are the only components that latch a control flag. The stall line is used to halt the processing in the functional unit as indicated by an external control line. The choice of multiple stall lines further facilitates the implementation of independently operating streams.

Valid Bit Mux	L<16> AND R<16>	00	L<16> AND Valid FN	10
	Valid FN AND R<16>	01	Valid FN	11
Output Program Stream East	False	0	True	1
Output Program Stream North	False	0	True	1
Output Program Stream South	False	0	True	1
Output Program Stream West	False	0	True	1
Only Program IFU	False	0	True	1
Enable Data Delay	False	0	True	1
Enable Flag 1 Delay	False	0	True	1
Enable Flag 2 Delay	False	0	True	1
Conditional Flag	ALU Out	0	FN Out Chooses	1
Stall Line Input	None	00	Line2	10
	Line 1	01	Line3	11

Table 2.9 Programmable Components

2.2.2.2.2 Functional Unit Programming Packets

To reconfigure a functional unit, eight 16-bit programming packets are required. These packets are stored in each functional unit's configuration registers. Below the packets are listed in the sequence required to program the functional unit. Note the address scheme for the functional units is listed in Section 2.2.2.1.

IFU Register

0	1	2	3	4	5
1	Only Program IFU	Output Program Stream North	Output Program Stream East	Output Program Stream South	Output Program Stream West

6	7	8	9 15
Constant Valid Bit	Constant <0>	Feedback Enabled	IFU Address

Configuration Register 1

0	1 15
0	Constant <1:15>

Configuration Register 2

0 1 2 3 4 5 6 7 8

0	Flag2 Output	Stall Mux	NS Dir Flag	EW Dir Flag	Conditional Mode Flag	FN Output

9 10 11 12 15

FN Valid	Valid Bit Mux	FN LUT

Configuration Register 3

0 1 3 4 5 7 8 9 10 11

0	Left Data Source	Enable Data Delay	Right Data Source	North Skip Bus	East Skip Bus

12 13 14 15

North Skip Bus	East Skip Bus

Configuration Register 4

0 1 2 3 7 8 9 10 11

0	Flag 2 Delay	Flag 2 EW Dir Flag	Flag 2 Input	Flag 2 Skip Bus North	Flag 2 Skip Bus East

12 13 14 15

Flag 2 Skip Bus North	Flag 2 Skip Bus East

Configuration Register 5

0 1 3 4 5 6 7

0	Shift Operation	ZSEL	Invert Carry In	Invert Shift In	Invert Shift Condition

8 9 10 11 12 13 14 15

Carry In Select	Shift In Select	Shift Condition	Flag 2 NS Dir Flag	Flag 1 NS Dir Flag

Configuration Register 6

0 1 2 3 7 8 9 10 11

0	Flag 1 Delay	Flag 1 EW Dir Flag	Flag 1 Input	Flag 1 Skip Bus North	Flag 1 Skip Bus East

12 13 14 15

Flag 1 Skip Bus North	Flag 1 Skip Bus East

Configuration Register 7

0	1 – 2	2 – 3	4 – 7	8 – 11	12 – 15
0	Flag 1 Output	Left Register Load	ALU R	ALU P	ALU G

2.2.2.3 Multiplier

The multipliers in the mesh implement a two-cycle pipelined 16 x 16 $\rightarrow$ 32 unsigned single-cycle multiplication [14][16]. Thus for this multiplier, two cycles are required for calculating a single output, but the process can be pipelined so that a continuous stream of outputs are created with a single cycle latency. In order to support the insertion of the multiplier into the mesh which uses 16-bit data busses, the multiplier's 32-bit output is split into two sixteen bit words. Horizontally, the upper sixteen bits are output onto the skip bus and the lower sixteen bits are output onto the local bus. Vertically, the upper sixteen bits are output onto the local bus and the lower sixteen bits are output onto the skip bus. This is illustrated below in Figure 2.15.

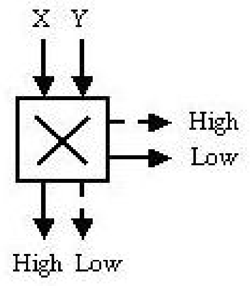


Figure 2.15 Multiplier Data Flow

2.2.2.3.1 Multiplier Programming Packet

Multipliers are not programmed as their operation is always the same. Therefore there is no multiplier programming packet and no addressing scheme assigned to the multipliers.

2.2.2.4 Intra-Mesh Connectivity

Connectivity within Stallion is provided in a manner similar to that of a FPGA; direct connections are provided to adjacent functional units; long distance connections are provided by a number of busses and switch matrixes. These connections are provided for data and control signals (Flag1 and Flag2). For Stallion, the direct connections are termed *local busses* and the long distance connections are called *skip busses*. Each local bus acts as a bi-directional full duplex connection. Local bus connections are provided to each adjacent functional unit as shown below in Figure 2.16.

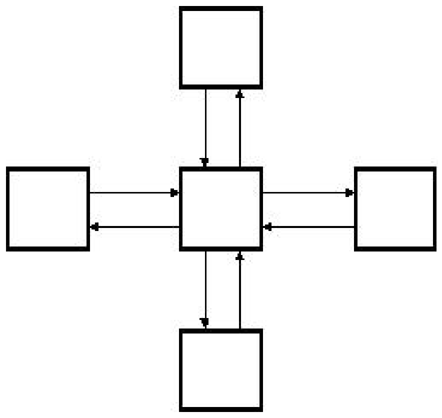


Figure 2.16 Local Bus Connections

Skip busses provide the mechanism for information to flow between any two functional units within the mesh, regardless of distance, in a single clock cycle. The skip busses are implemented as a series of half-duplex links routed through a number of switching matrixes that form short links between interconnected functional unit (IFU) as shown below in Figure 2.17. Two directional switches control the direction of North-South and East-West information flow across and into an IFU. For skip bus links carrying information away from an IFU, another switching matrix selects the source of the information from one of four possible locations: the opposite skip bus, the skip bus to the right, the primary FU output register and the auxiliary FU output register.

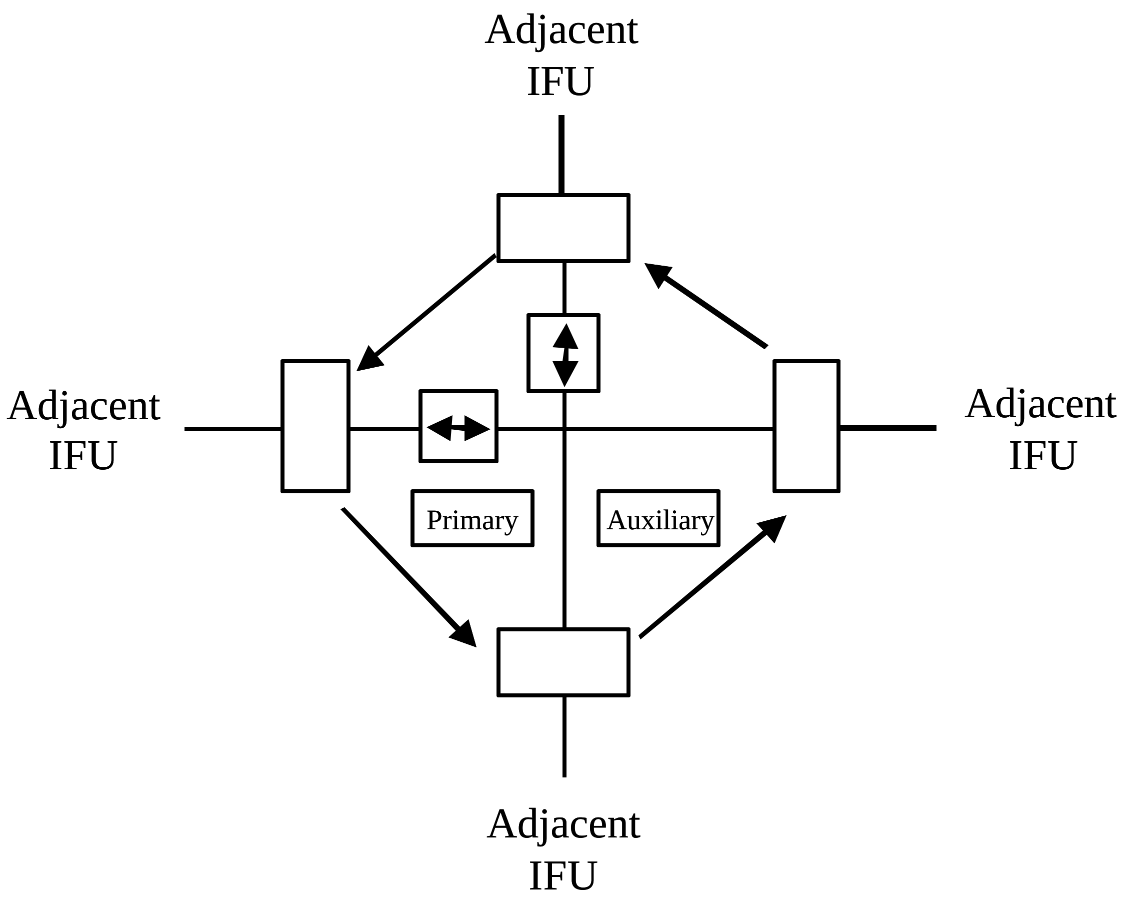


Figure 2.17 IFU skip bus links

By properly configuring the switching matrixes, links across an entire mesh can be created. For example in the submesh shown below in Figure 2.18, a skip bus is established from the functional unit in the top right (A4) to the functional unit in the bottom left (B1). In order to establish this link, in the A4 functional unit sets the N/S directional switch to south and the south skip bus multiplexer to accept an input from the primary output register. Functional unit B4 sets its N/S directional switch to south and the E/W switch to west. The west skip bus input multiplexer is set to accept the skip bus input from the north. In functional units B1-B3, the E/W directio nal switch to west, and in B2 and B3 the west skip bus input multiplexer is set to the east skip bus input. Then the functional unit B1 can use the output generated by A4 in the previous cycle as an operand.

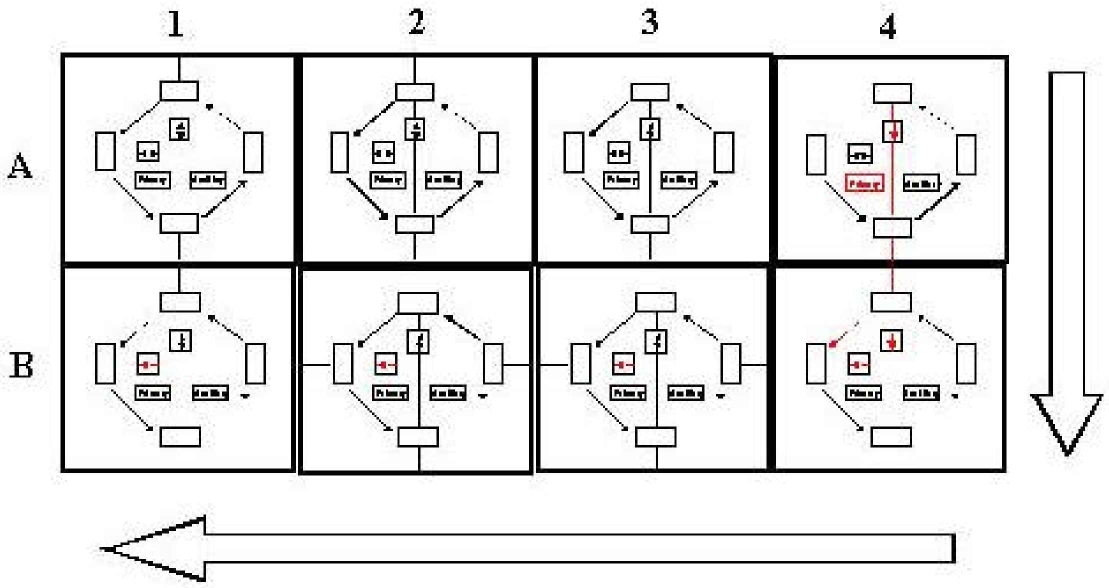


Figure 2.18 Example Skip Bus Connection

2.2.3 Data Port

The data ports provide the means of interfacing with Stallion. The data width of each of the six data ports is 18-bits. Each data port is bi-directional and half-duplex, allowing for a high degree of flexibility when mapping an algorithm to Stallion. For instance when implementing two parallel finite impulse response (FIR) filters on Stallion, each filter needs two input streams and one output streams, thus four inputs and two outputs for Stallion. However, an instantiation of a rate 1/3 convolutional encoder requires one input and three outputs. This is an important consideration the implementation of a waveform requires Stallion to change configurations several times [5]. The process of rapidly and completely changing the configuration of Stallion is known as *hardware paging*.

2.2.3.1 Data Port Programming

The programming options for a data port are shown below in Table 2.10. The data port can be programmed to be either input (read) or output (write) by appropriately setting the RW State bit. Each data port can be tied to any one of three stall lines through the stall mux. Additionally, synchronization of inputs between the data ports can be programmed by setting the synchronization bits. Details of synchronization are given in [14].

RW State	Read	0	Write	1

Stall Mux	None	00	Line 2	10
	Line 1	01	Line 3	11
Synchronization Bits	Five bits that indicate whether its input should wait on a synchronization signal from a particular data port.			

Table 2.10 Data Port Programming Options

In Stallion, the address for a data port is assigned the address 39 to 44 for data port 1-6. The data port's programming information is arranged into the packet shown below in Table 2.11.

0	1	2 3	4 8	9 15
1	RW State	Stall Mux	Synchronization Bits	Address

Table 2.11 Data Port Programming Packet

2.2.4 Crossbar

To facilitate communications between the meshes and between the data ports and the meshes, Stallion instantiates a fully interconnected crossbar. There are a total of twenty-two inputs to the crossbar: eight from each of the two meshes and one from each of the six data ports. As illustrated below in Figure 2.19, the crossbar provides the capability to program a connection from an output node to any input node. Note that the multiple output nodes may be connected to a single input node.

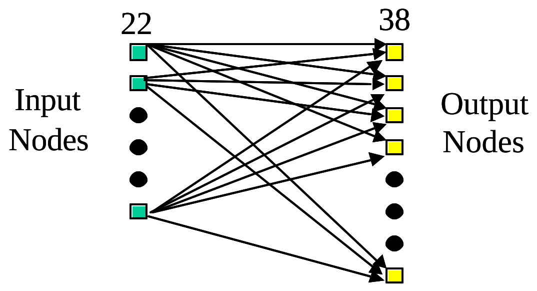


Figure 2.19 Generalized Crossbar Connections

2.2.4.1 Crossbar Programming

Conceptually, the crossbar is programmed by programming the input crossbar nodes. The basic programming options are a specification of the stall line that should be used, and the output crossbar nodes to which the input node should be connected. To facilitate connections to multiple output crossbar nodes, the crossbar programming packet, shown in Table 2.12, includes an extra field to indicate whether or not the next packet will be extending the list of output crossbar nodes that are connected to the input crossbar node.

0	1	2 3	4 8	9 15
1	Extension	Stall Mux	undefined	Output Node Address

Table 2.12 Crossbar Programming Packet

2.3 Object Oriented Programming

Object oriented programming (OOP) is a powerful concept that can be used to model real world devices and their interactions and is used extensively in many languages including C++, Java, and Ada. As part of the documentation of the Stallion simulator, the following gives a brief description of the basic concepts and tools of OOP used in developing the simulator. Note, this is not an exhaustive covering of OOP, as important concepts such as polymorphism are not addressed as they are not used in the simulation. However, this section does cover all of OOP that is required to understand the simulation or would be awkward to completely ignore.

2.3.1 Objects

The basic component of OOP is the *object*. An object is a meaningful abstraction of some thing. As shown below in The abstraction consists of internal characteristics and interfaces.[1] Characteristics may be internal constants/variables, or relationships (functions) that define how inputs to the object affect the output or characteristics of the object. The characteristics of objects are sometimes modeled as state machines. Interfaces provide methods for inputting and retrieving information from the object. In an object the internal characteristics are generally hidden from the outside world through a process known as *encapsulation.*

[1] This differs slightly from the Object Management Group's preferred primary object components of attributes and operations which the feel are better suited for the description of a class than a conceptual object.

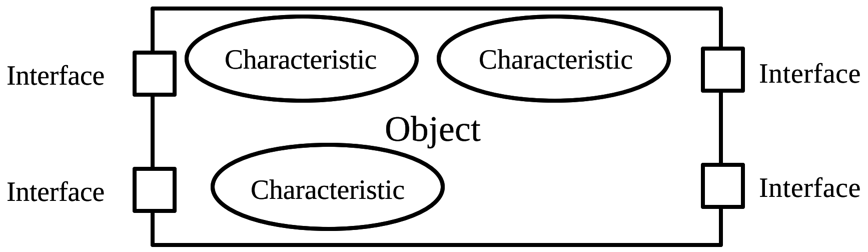


Figure 2.20 Conceptual Model of an Object

To understand the object concept, consider the finite impulse response (FIR) filter shown below in Figure 2.21. A FIR filter has two interfaces: an input and an output. It has the following characteristics: an impulse response, a set of registers, the way the output is calculated, and the method by which the registers are updated.

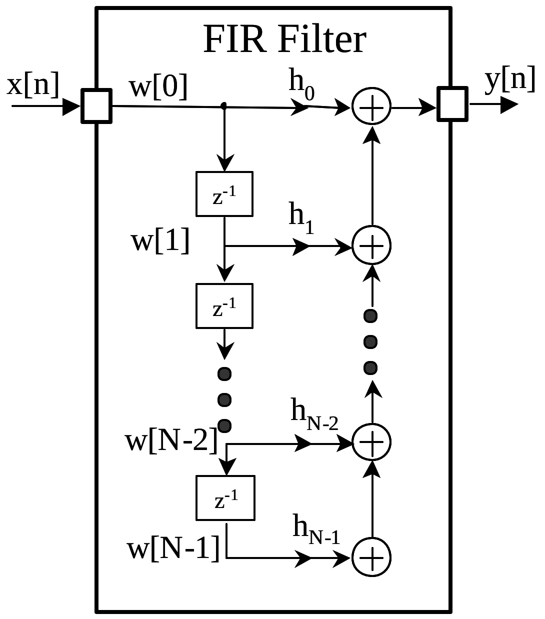


Figure 2.21 FIR Filter

2.3.2 Classes

Objects are usually instantiated with a *class*. A class is a collection of data and functions that collectively describe the characteristics and interfaces of an object. The segregation of a class's component into either a characteristic or an interface is determined by the component's *visibility*. Visibility specifies the access and scope of a component or object. There are two levels of visibility that are fundamental to the object concept: private and public. A data element or a function is given private visibility when it is intended as an internal characteristic; it is given public visibility when it is intended as an interface. As an example, the class diagram in Figure 2.22 shows which components of the previously described FIR filter object would be private and which would be public. The words in guillemots («») are *sterotypes* which are used to group data and functions in a logical manner in a class diagram.

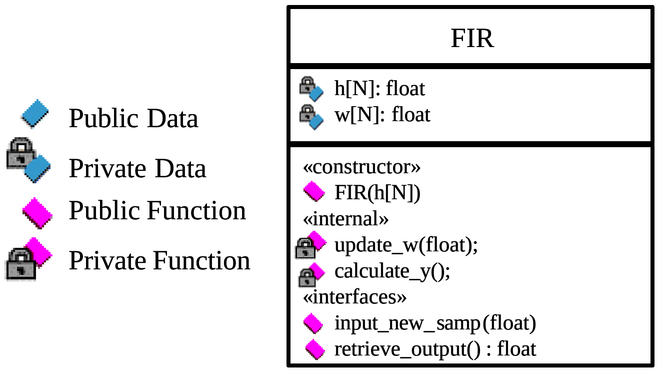


Figure 2.22 FIR Filter Class Diagram

2.3.3 Associations

Often, objects must interact with other objects and some relationship between objects dictates the interaction. When this occurs the objects are said to be *associated* with one another. An association may be well-defined or loosely defined and may be defined between objects of the same kind or between completely different types of objects. An association can be applied to any numerical correspondence (M to N) between objects.

In UML, an association is depicted by a line that connects two objects. Depending on the association, the line can be solid or dashed, have any number of different beginning and ending symbols. Specialized relationships are explicitly written with the object's role in the relationship next to that object. Numerical correspondence can be indicated by stacked objects or by numbers at the edges of association lines. Ellipses are used to indicate inclusive sets, and asterisks denote arbitrary numbers. When a number is not explicitly written, "1" is implicitly assumed. Figure 2.23 shows an explicit association between a person object and a company in UML. Note that a company necessarily has at least one employee, and a person may work for any number of companies (disregarding practical time constraints) or may be unemployed.

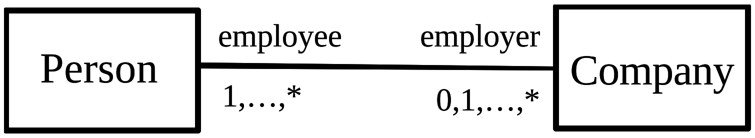


Figure 2.23 Explicit Employer - Employee Association

2.3.4 Dependency

One of the most fundamental associations is the *dependency* association. When a dependency association exists, some operation of an object is reliant on the operation of another object. This is also referred to as a directional relationship. For instance in a pipelined process, each stage of the process is directly dependent on the immediately preceding stage and indirectly dependent on all prior stages. In UML a dependency is indicated by a dashed arrow that points to the object that is depended on. Sometimes the exact dependency is indicted by a stereotype. An example is shown below in Figure 2.24.

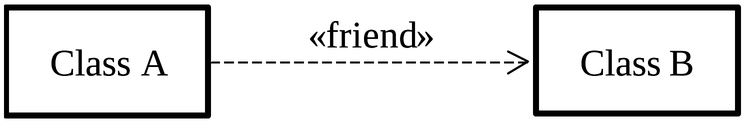


Figure 2.24 UML Dependency Representation

2.3.5 Inheritance

Inheritance is a type of association wherein an “is a” relationship exists. When a chain of inheritance relations exists, the entire relationship structure can be visualized as a hierarchy or taxonomy. When an object inherits from another object, some or all of the attributes of the “parent” object are also present in the “child” object. Inheritance also introduces another type of visibility known as *protected* wherein the protected component is visible internally to the object and to inherited objects. Although an important association, inheritance is not used in the Stallion simulation and thus the details on the types of inheritance and how inheritance is depicted in UML is not included here.

2.3.6 Aggregation

Another kind of association is *aggregation* where objects contain other objects. In general, aggregation describes a “has a” or a “is a part of” relationship between objects.

There are three fundamental types of aggregate relationships: nesting, composite, and aggregation. Note that due to inheritance relationships, a single component can experience many different kinds of aggregation relationships. In nesting the component objects only have scope within the aggregating object, *i.e.*, only objects of this kind have this component. An example of a nesting relationship is backbones and the Vertebrate animal class. The nesting relationship is shown in UML by an anchor at the end of an association line with the anchor towards the aggregating object. An example is shown below in Figure 2.25.

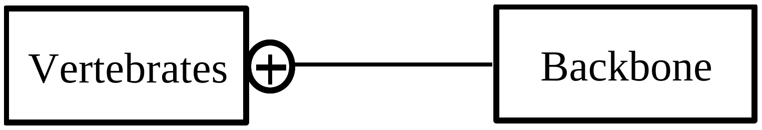


Figure 2.25 UML Nesting Relationship

In a composite relationship, the component object is “owned” by the aggregating object and the component object’s life cycle is completely dependent on the aggregating object. However, other kinds of different objects may also have the same component. An example of a composite relationship is a heart which both humans and birds have; note that a heart cannot exist outside of its aggregating object. The composite relationship is shown in UML by a filled diamond at the end of an association line with the diamond on the end of the aggregating object. An example is shown below in Figure 2.26.

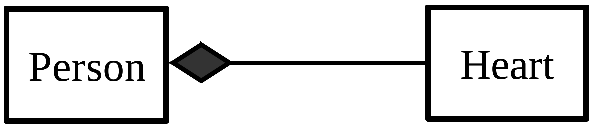


Figure 2.26 UML Composite Relationship

In an aggregate relationship, the component object has its own life cycle independent of the aggregating object, but nonetheless “belongs to” the aggregating object. An example of an aggregate relationship is a person who is a member of a club. The aggregate relationship is shown in UML by an open diamond at the end of an association line with the diamond next to the aggregating object. An example is shown below in Figure 2.27.

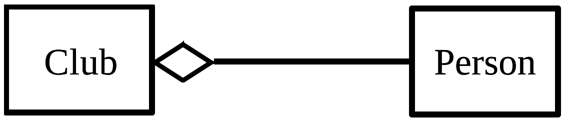


Figure 2.27 UML Aggregation Relationship

Chapter 3

3 Stallion Configuration Development Tools

Chapter 3 describes the tools created as part of this thesis for developing configurations for Stallion as part of the Layered Radio Architecture. Specifically, this thesis involved the development of a programmable Stallion simulator, a graphical programming interface for Stallion, and a programmable controller chip object that mimics the essential operations of the configuration layer.

3.1 Programmable Stallion Simulator

As part of this study, it was determined that a programmable simulation of Stallion was required. This simulation was intended to serve as a tool for:

- validating configurations
- debugging configurations
- predicting power consumption
- measuring cycle counts

Creating a simulation of Stallion is complicated by the following factors:

- managing the multitude of programmable devices
- constructing the dynamic links between elements
- satisfying the varying timing constraints on each link
- maintaining dynamic chains of dependencies of undefined length

In order to be such a flexible and complicated simulation, a powerful modeling approach is required. The remainder of this Section documents the object-oriented approach used to construct the simulation of Stallion. To aid in the documentation, Unified Modeling Language (UML) diagrams are used throughout this Section. Although sufficient information is given in Section 2.3 to read the diagrams, complete documentation on UML can be found in [18].

3.1.1 Stallion Simulation Conceptual Overview

The Stallion simulation was implemented using OOP concepts in the C++ language. OOP is well suited for the simulation of CCMs as there are numerous components with dynamic interconnections, which is not easily implemented using procedural programming techniques. VHDL would also provide similar capabilities for modeling, but without implementing the simulation on a FPGA, the simulation would be excessively slow and interfacing the simulation with support programs that might be developed later would be more difficult.

Each conceptual component of Stallion was implemented as an object from a data port to a crossbar node to an ALU to the carry in flag. These objects were arranged in a number of complex aggregations that extended grouped together the objects in a manner that exactly reflected the components in Stallion. In this simulation, the most commonly used aggregation relationship in Stallion is the nesting relationship as each component in Stallion tends to only interact within its aggregating component.

In addition to the aggregation of objects, numerous objects within each object were dependent on other objects. To facilitate the implementation of these dependencies, each object with a dependency was given a pointer to the object it depends upon. For example each ALU object has a pointer to a shifter object, a right register object, and a carry in object. It was the responsibility of the aggregating object to manage the dependencies of all of its components. To implement the logical (not occurring on clock cycle boundaries) connections within Stallion, each computational object computes its output each time an object that is dependent on it needs to generate an output. Because of the specifications of Stallion, it is possible to create infinite logical dependencies both within a functional unit and across functional units. Since it was decided that a "good" programmer would not create an infinite loop as it would produce an unreliable results, no technique for identifying and escaping from infinite loops was employed. To implement the clocked transmissions within Stallion, registers were created that only updated at the end of a simulation cycle that would not induce the execution of dependency chains. Thus it can be seen that two types of objects are required to simulate Stallion: logical objects whose output is instantaneously computed and clocked components whose output only updates at clock transitions. Aggregate objects may contain both logical and clocked components. Whenever an aggregate object contains at least one clocked component, regardless of the number of logical components, the aggregate object must also be a clocked component.

Each of the primary objects (functional units, data ports, multipliers, and crossbar connections) are programmed at run time from an independent file generated by the Stallion programming interface whose programming words exactly correspond with the bit sequences that would be used to program Stallion. Note that wormhole programming is not supported in the simulation as it was not felt to be vital to the study as hardware paging was used exclusively in all designs as dictated by the design of the configuration layer. The management of assigning the programming files is handled by a main routine that through passed arguments determines the run time of each simulation as well as the data files used for configuration validation.

To support debugging, every object has a number of linked lists that are used to store the inputs, outputs, and state information of every object. At the end of the simulation, this would be dumped to a number of files. Alternate versions for use when repeated simulations were run (such as in the simulation of an entire waveform implemented on the simulator) were later built that used fixed arrays and made the storing of debug info optional.

To support power consumption predictions, all "edge" objects (local busses, skip busses, crossbars and data ports) keep a running total of the number of bit flips that occur during a simulation and use bulk capacitance estimates from the VLSI layout [15] and voltage and clock rate specifications to estimate power. To calculate power, outputs are registered and the number of bit flips that occur are counted by comparing the previous output with the current output. Then dissipated power can be calculated as

$$P_{diss} = \frac{1}{2} bCfv^2 ,$$

where b is the number of bit flips (toggling lines) in a cycle, C is the estimated capacitance, f is the clock frequency of Stallion and v is the voltage. In the simulator C is 39fF, f is 50 MHz, and v is 3.3 volts to match the specifications described in [15].

Later, to support the requirements of the WCDMA study, the Stallion object was further extended so that it could work with up to twelve data ports as specified during initialization.

3.1.2 Stallion Simulation Methodologies

This Section refines the description of the simulation methodologies employed in the Stallion simulation. Specifically, this section covers the techniques used for processing data and clocking and for generating and storing debugging information.

3.1.2.1 Data Processing and Clocking

The objects in the Stallion simulation are grouped into two types of objects: logical objects and clocked objects. Both of these kinds of objects have an output() function that returns that objects processed output. When the object is a logical object, the output() function will cause the object to request a new input from each of its input objects by calling those objects' output() functions. Once the values from these input objects have been returned, the original object immediately performs its calculation and returns its value to the its calling object. However, when the object is a clocked object, the output() function returns the value stored in an output register. The clocked object also contains a process() function that acts like the output function of a logical object's output function by calculating a result after first fetching fresh outputs from its input objects. However, instead of returning its output, the clocked object stores its output in a temporary register. The clock object also contains a third function, update_output(), that transfers the value from the temporary register to the output register. Any object may have either clocked or logical objects as its inputs. However, notice that the object does not need to know what kind of object it has as its input as all objects return an output.

An object may contain an arbitrary number of aggregate objects that can be either logical objects or clocked objects. Objects external to the aggregating object that require the output of a specific aggregate object (such as can be the case for passing flags between functional units) are given a pointer to that object. If the aggregate object contains any clocked objects, then the aggregate object is also considered a clocked object and will support a process() function. When the process() function is called for that object, the aggregating object will then call the process() function for all of its clocked objects. Similarly, when the update_output() function is called, the aggregating object will call the update_output() function for all of its aggregate objects. A conceptual class diagram of the logical objects, clocked objects, and aggregating objects is shown below in Figure 3.1. Note that it is the operation of the output() function that is unique for each logical computational core and the process() function that is unique for each clocked computational core. For instance a shifter object would have a different output function than a FC (carry in) object, and a bus_output object would have a different process() function than an aux_output object in a simulation of a Functional Unit.

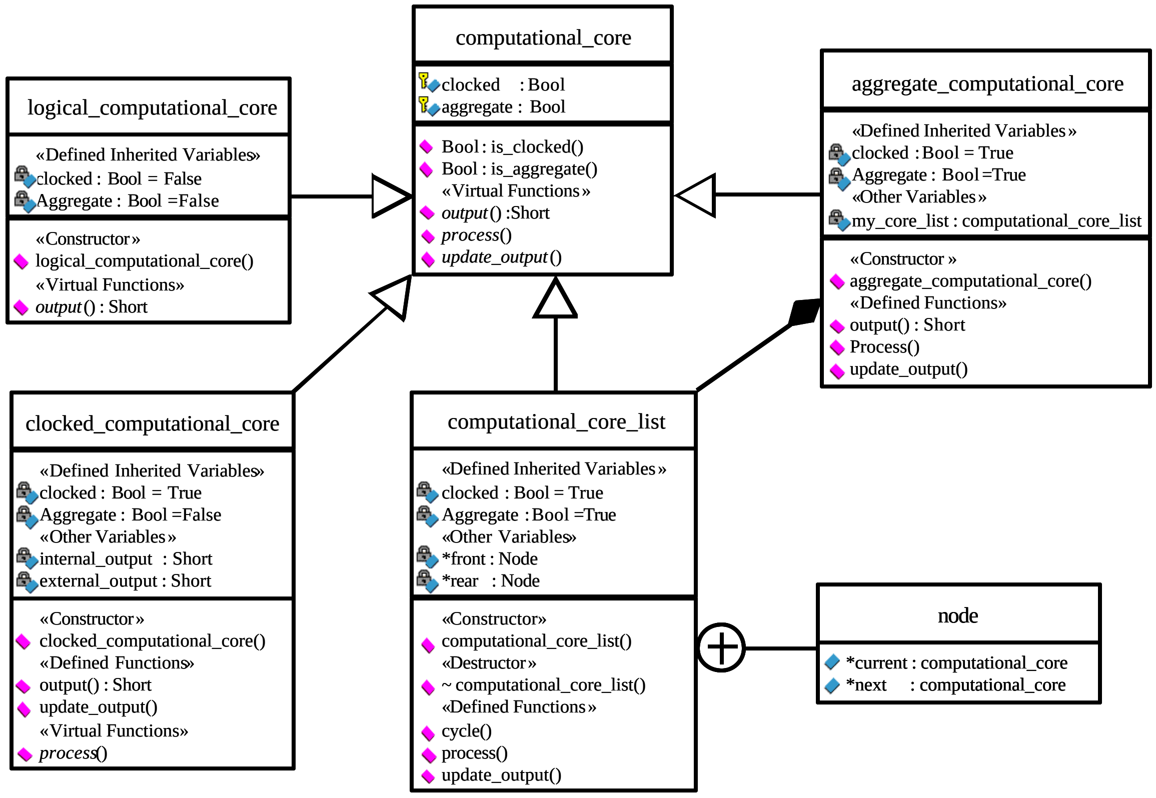


Figure 3.1 Conceptual Class Diagram for Simulation Objects

This methodology naturally lends itself to the following clocking approach. A clock cycle for the simulation is performed by executing the following steps at the top level aggregating object.

1. Call each of its clocked objects' process() function.
2. Call each of its clocked objects' update_output() function.

Note that in this approach, logical objects whose output is not needed may never be called and thus may never change their value. However, it is expected that all of the final outputs of the simulation will be clocked (in the case of the Stallion simulation, this corresponds to the data port objects). Thus all logical objects that are in some way involved in the operation of the simulation will be called, and all logical objects that are not involved, will be left idle. This has a beneficial side effect of reducing the time required to execute the simulation.

By using this methodology, an arbitrary number of logical objects can be linked together to create outputs that change "instantaneously." Note that in a real device, there would be practical latency considerations that would limit the number of components that could be used in a chain. However, when simulating an existing device, the latency of processing chains should have been considered and excessively long paths should be registered. Thus any processing chain should terminate in a clocked object. An example processing chain is illustrated in Figure 3.2. In this diagram, the functional unit objects are aggregate objects, the bus output objects are clocked, and the remaining objects are logical. When functional unit LB4 is instructed to process, it will instruct its bus output object to process which will call the output() function of the conditional mode object. This will result in a string of operations and calls to output() functions until the output() functions are called for the bus outputs of functional units LA4 and LB5 (Flag 1 has been programmed to constantly output a '1' and thus has no dependencies). As the bus output objects of LA4 and LB5 are clocked, they will simply return the output they stored on the previous clock cycle. Processing then proceeds back up the chain until the process function call of the bus output of LB4 completes. After all of the clocked objects have had their process function called, their update_output() functions will be called to make their just calculated output available on the next output.

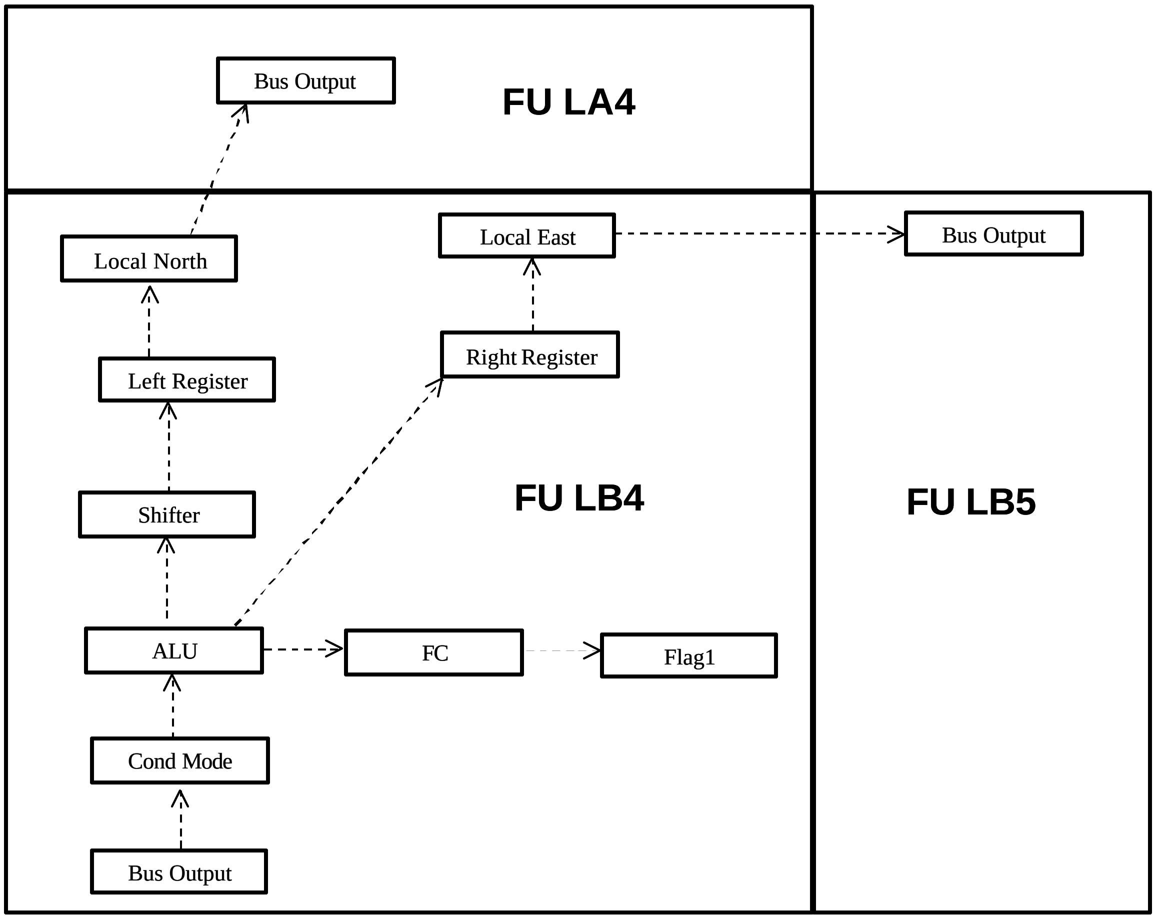


Figure 3.2 Example Data Processing Chain

This simulation methodology is currently being used in the development of a dynamic simulation of arbitrary CCMs for identification of the ideal CCM for handsets. However, in the Stallion simulation, while this approach was conceptually followed, the approach was not crystallized until after the initial coding of the simulation had completed. Thus the structure depicted in Figure 3.1 is not strictly followed, and neither the inheritance scheme nor the nomenclature is closely followed. In the formal documentation of the Stallion simulation that follows in 3.1.3, all deviations from this nomenclature are clearly noted.

3.1.2.2 Generation and Storage of Debugging Information

To speed up the development of programs on Stallion, the Stallion simulation makes available precise, detailed information on the value and timing of every operation performed by each component and subcomponent within Stallion. To do this, each component and subcomponent stores its output in a unique singly linked list on a cycle-by-cycle basis. Thus an additional step occurs during a simulation clock cycle to instruct the components to update their linked lists with their current outputs.

At the end of a simulation, all of a component's subcomponents' linked lists are grouped together and dumped to the file specified during simulation configuration. This file is broken into logical sections dictated by the component's relationships to other subcomponents. For instance, the ALU carry in flag, ALU carry out flag, overflow flag, and ALU results are presented in the same section. Each section is clearly labeled and separated by a title and lines of equal signs ('='). Each piece of output data is given a meaningful description and the value of that data written from left to right by clock cycle. A partial listing of an example debug file for a functional unit is shown below in Figure 3.3. This approach allows for a quick examination of exactly what was done by all of the simultaneous operations occurring in Stallion.

```
====================================================
=                          Left Register           =
====================================================
Content of left Register
0 0 0 0 0 65417 65439 65243 26 25 65395 65445 160 111
Valid Bit
0 0 0 0 1 1 1 1 1 1 1 1 1 1

====================================================
=                          Right Register          =
====================================================
Content of right Register
0 0 0 0 0 65417 65439 65243 26 25 65395 65445 160 111
Valid Bit
0 0 0 0 1 1 1 1 1 1 1 1 1 1

====================================================
=                            Shifter               =
====================================================
Output
0 0 0 0 0 65417 65439 65243 26 25 65395 65445 160 111
```

```
FS Out
0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 3.3 Partial Listing of a Simulator Debug File

As the continuing allocation of memory in a linked list can be time consuming, the simulation was designed so that debugging information could be suppressed and the link lists would never be accessed. This results in a significant improvement in speed and is most valuable when running long simulations using previously validated configurations, such as when predicting power consumption or measuring performance for a waveform implemented on Stallion. Another alternative to speeding up the runtime would be to use fixed memory allocations, perhaps in arrays. However, this has the potential to greatly limit the flexibility of the design and would typically allocate excessive memory for the relatively short simulations used while developing a configuration. Future simulation designs might strike a compromise by dynamically allocating memory at initialization based off of desired number of execution cycles. However, this approach could run into problems if the simulator incorporated stream based programming as the total number of cycles may not be known during initialization.

3.1.3 Stallion Simulation Documentation

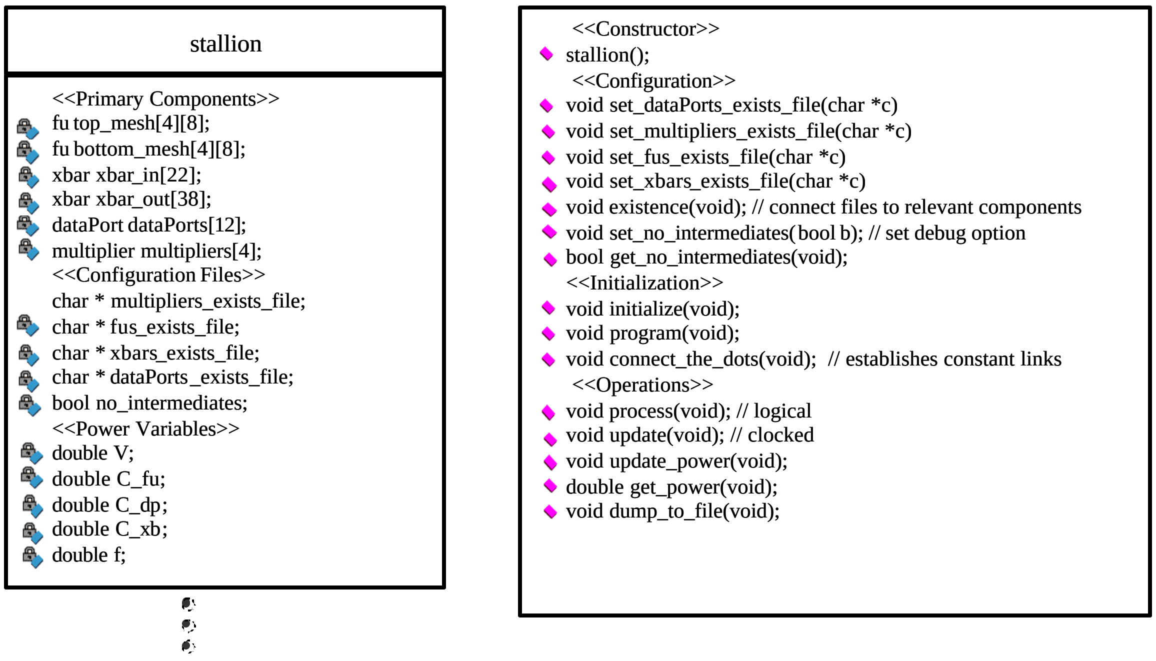
The following gives a description of the interfaces and characteristics of all of the objects within the Stallion simulation and also shows the associations between different objects within the simulation.

3.1.3.1 Stallion

Stallion is a clocked component as all of its aggregate components are also clocked. The class diagram for the Stallion object is shown below in Figure 3.4. There are five basic objects in the Stallion simulator, which correspond to the primary components of Stallion: Functional Unit, Multiplier, Data Port, Input Crossbar Node, and Output Crossbar Node. The Stallion object is responsible for the following actions:

- providing the references to the basic objects so connections between the objects can be formed (connect_the_dots())

- managing the programming of the components (initialize(), program())
- managing the initiation of logical calculations and clocked transitions (process(), update())
- managing the generation of debugging information (dump_to_file())
- managing the power calculations (update_power(), get_power())



```
stallion

<<Primary Components>>
fu top_mesh[4][8];
fu bottom_mesh[4][8];
xbar xbar_in[22];
xbar xbar_out[38];
dataPort dataPorts[12];
multiplier multipliers[4];
<<Configuration Files>>
char * multipliers_exists_file;
char * fus_exists_file;
char * xbars_exists_file;
char * dataPorts_exists_file;
bool no_intermediates;
<<Power Variables>>
double V;
double C_fu;
double C_dp;
double C_xb;
double f;
```

```
<<Constructor>>
stallion();
<<Configuration>>
void set_dataPorts_exists_file(char *c)
void set_multipliers_exists_file(char *c)
void set_fus_exists_file(char *c)
void set_xbars_exists_file(char *c)
void existence(void); // connect files to relevant components
void set_no_intermediates(bool b); // set debug option
bool get_no_intermediates(void);
<<Initialization>>
void initialize(void);
void program(void);
void connect_the_dots(void);  // establishes constant links
<<Operations>>
void process(void); // logical
void update(void); // clocked
void update_power(void);
double get_power(void);
void dump_to_file(void);
```

Figure 3.4 Stallion Class Diagram

The basic components and multiplicity of components in the Stallion object are arranged as shown below in Figure 3.5. Note that although there are technically 64 functional units, four of these are never programmed nor connected, nor instructed to execute (top_mesh[0][0], top_mesh[0][7], bottom _mesh[0][0], bottom_mesh[0][7],) as these positions within a mesh are actually occupied by multipliers. This simplified the process for the initialization of connections and management of the meshes.

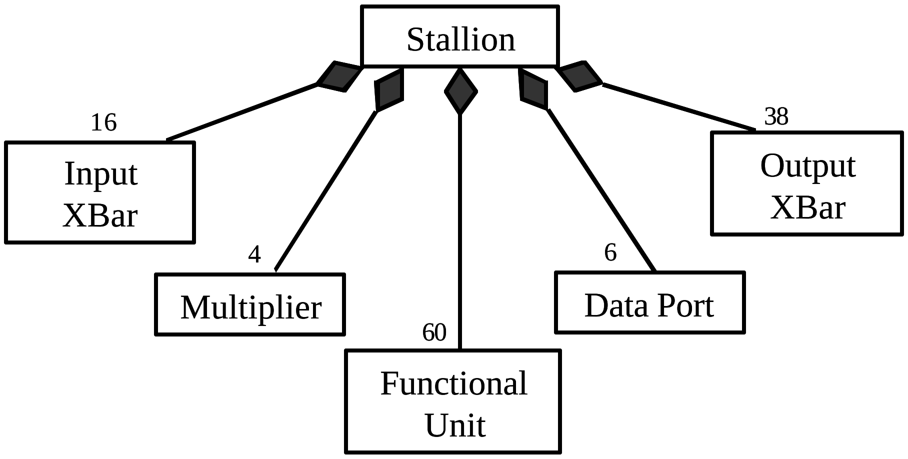


Figure 3.5 Stallion Component Diagram

The dependencies between the Stallion object's components are shown below in Figure 3.6. Notice that the output of a functional unit can depend on a multiplier, an output crossbar, or up to four different functional units. A functional unit may or may not have a dependent input crossbar. Note that the exact dependencies for a specific functional unit is a function of its location within a mesh. For instance a functional unit at position A7 would have a dependency on two output crossbar nodes, a multiplier and two functional units while a functional unit at position B7 would be dependent on four functional units. These details are taken care of in the connect_the_dots() function. However, every functional unit had components that contained pointers to all of the possible dependencies. In theory, any number of output crossbar nodes can be dependent on a single input crossbar node. However, in the actual implementation, the programming of the input crossbar node was created so that it only supports data flow with a single output crossbar node. Potential steps to address this issue are described in Section 3.1.3.5.

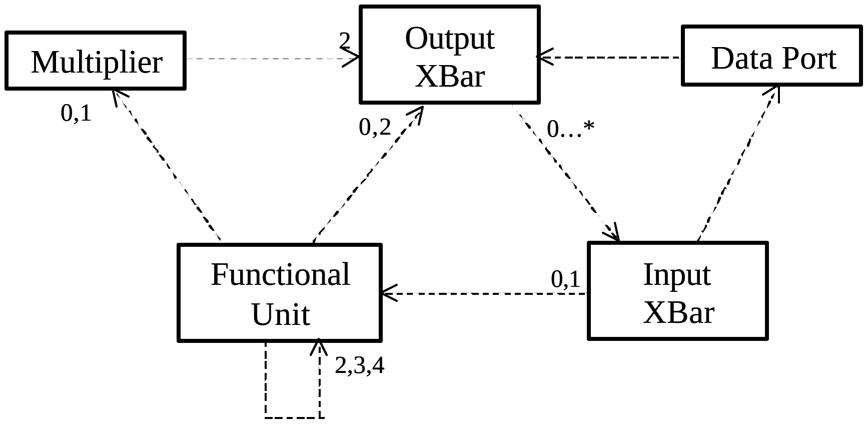
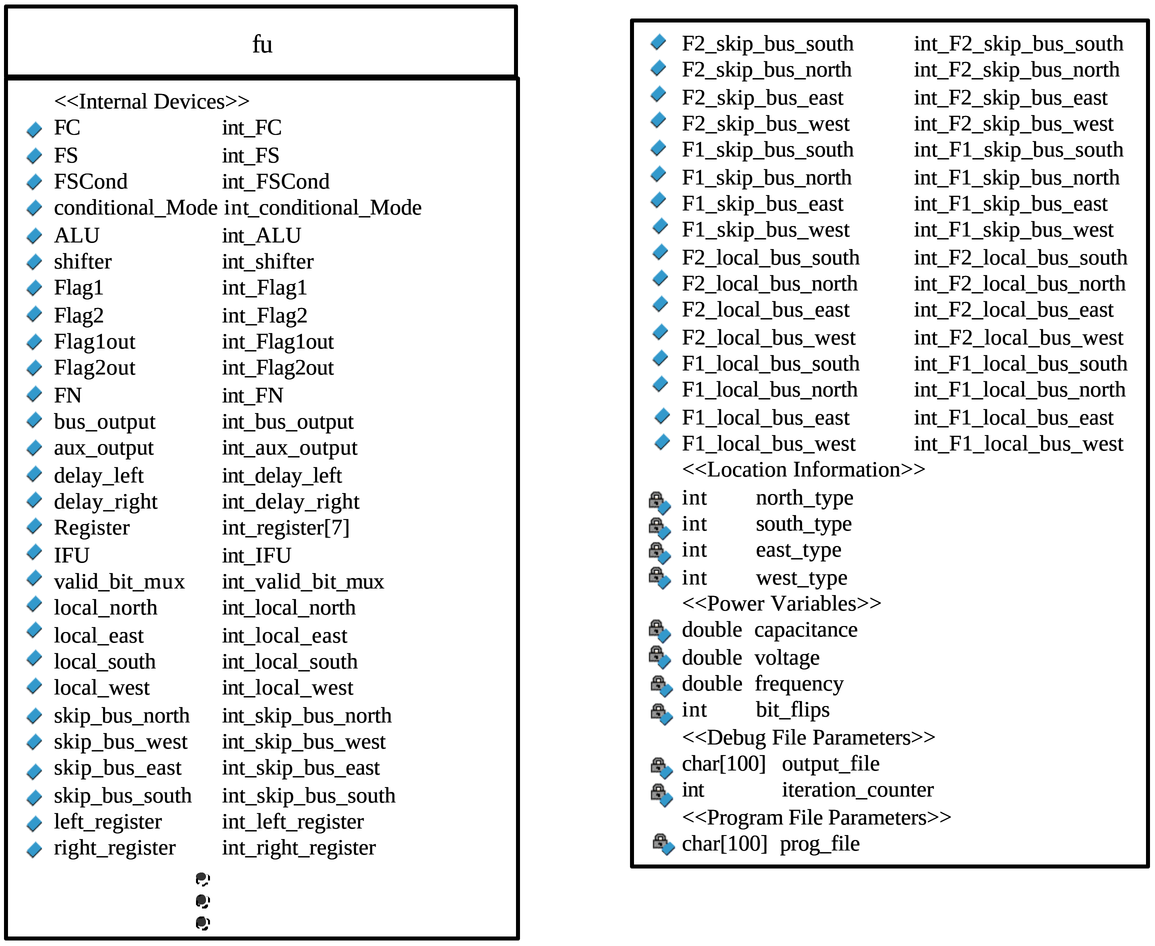


Figure 3.6 Stallion Primary Component Dependencies

3.1.3.2 Functional Unit

The functional unit is a clocked object as it contains clocked objects. A partial class diagram for the functional unit object that only shows the components and data elements (attributes) is shown below in Figure 3.7, and the methods for the functional unit object (operations) are shown in Figure 3.8. Each programmable component within a functional unit is instantiated as an object and performs the operations and has the connections associated with that component. For instance, the ALU object implements an arithmetic logic unit that takes an input from the shifter and right input register, is dependent on a carry in value, and generates a carry out flag. Note that each of the functional unit components have public visibility. This simplifies the management of the dependencies between components of different functional units. There are also data elements that are used to hold information that is specific to a functional unit, such as the type of component that is next to each functional unit in the mesh in each direction (north_type, south_type, east_type, west_type) as there are four kinds of components to which the functional unit may be connected: another functional unit, a multiplier, an input crossbar node, and an output crossbar node. This information is given to the functional unit by the parent Stallion object in the connect_the_dots() function. Note that there are also bus and auxiliary registers (bus_output, aux_output) that are not programmable, but are instantiated to support clocked transitions.



fu	
<<Internal Devices>>	
FC	int_FC
FS	int_FS
FSCond	int_FSCond
conditional_Mode	int_conditional_Mode
ALU	int_ALU
shifter	int_shifter
Flag1	int_Flag1
Flag2	int_Flag2
Flag1out	int_Flag1out
Flag2out	int_Flag2out
FN	int_FN
bus_output	int_bus_output
aux_output	int_aux_output
delay_left	int_delay_left
delay_right	int_delay_right
Register	int_register[7]
IFU	int_IFU
valid_bit_mux	int_valid_bit_mux
local_north	int_local_north
local_east	int_local_east
local_south	int_local_south
local_west	int_local_west
skip_bus_north	int_skip_bus_north
skip_bus_west	int_skip_bus_west
skip_bus_east	int_skip_bus_east
skip_bus_south	int_skip_bus_south
left_register	int_left_register
right_register	int_right_register
	⋮

F2_skip_bus_south	int_F2_skip_bus_south
F2_skip_bus_north	int_F2_skip_bus_north
F2_skip_bus_east	int_F2_skip_bus_east
F2_skip_bus_west	int_F2_skip_bus_west
F1_skip_bus_south	int_F1_skip_bus_south
F1_skip_bus_north	int_F1_skip_bus_north
F1_skip_bus_east	int_F1_skip_bus_east
F1_skip_bus_west	int_F1_skip_bus_west
F2_local_bus_south	int_F2_local_bus_south
F2_local_bus_north	int_F2_local_bus_north
F2_local_bus_east	int_F2_local_bus_east
F2_local_bus_west	int_F2_local_bus_west
F1_local_bus_south	int_F1_local_bus_south
F1_local_bus_north	int_F1_local_bus_north
F1_local_bus_east	int_F1_local_bus_east
F1_local_bus_west	int_F1_local_bus_west
<<Location Information>>	
int	north_type
int	south_type
int	east_type
int	west_type
<<Power Variables>>	
double	capacitance
double	voltage
double	frequency
int	bit_flips
<<Debug File Parameters>>	
char[100]	output_file
int	iteration_counter
<<Program File Parameters>>	
char[100]	prog_file

Figure 3.7 Functional Unit Attributes Compartment

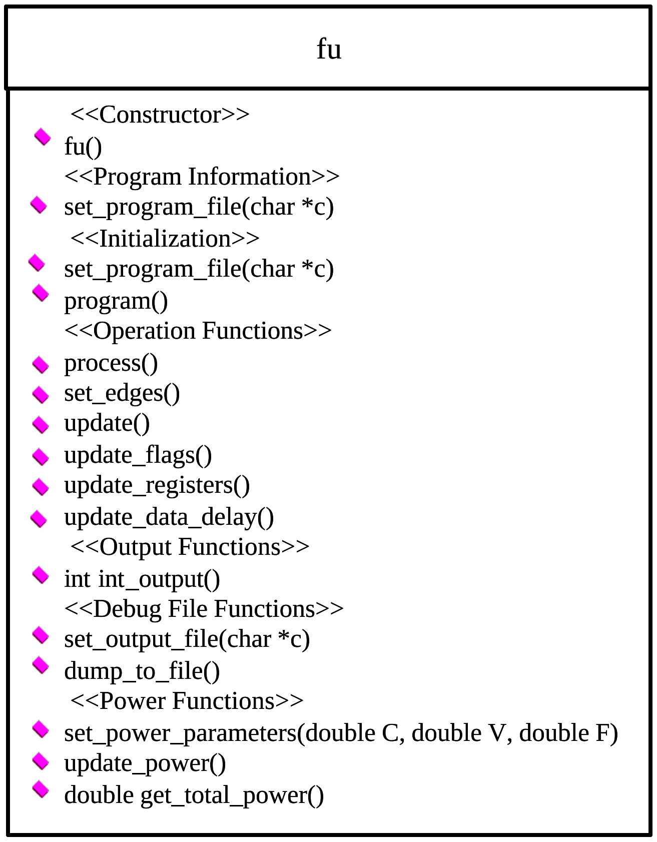


Figure 3.8 Functional Unit Operations Compartment

Of the objects listed in Figure 3.7, the following are clocked components:

- int_bus_output
- int_aux_output
- int_delay_left (delayed bus output)
- int_delay_right (delayed auxiliary output)
- Flag1out
- Flag2out

Note that that Flag1out and Flag2out are only clocked when their delay is enabled. The rest of the time, the flag outputs are not clocked. Also note that according to Stallion specifications, the left and right registers are both supposed to be latched. However, registering the bus and auxiliary outputs has the same effect. Additionally, clocking at the bus output eliminates the need for repeating calculations on the previous output. Also it should be noted that the inputs to the ALU and FC are also supposed to be latched, but the exact timing of when this latching would occur in relation to the operation of the shifter is unclear. While an important practical consideration to ensure proper timing in the actual chip, this is not considered in the simulation as the actual implementation of Stallion was supposed to take care of the timing considerations.

The internal dependencies of components within a functional unit that directly impact the functional unit's computations are illustrated below in Figure 3.9. Each object that is dependent on another internal object has a pointer to its dependency object. For all components in the functional unit other than the bus_output and aux_output, when an output is requested, the object recalculates its output. As part of recalculating its output, it requests any of its dependencies to recalculate its output before determining its output. Thus for a calculation to occurs, a series of calculation requests following back along the dependency path first occurs; once a clocked register is reached which has no dependencies, calculations follow back down the dependency path. A particular object will, however, only follow the dependencies indicated by its programming. For instance, an ALU programmed to add together its inputs will only use its shifter and right register dependencies, but not its flag dependencies.

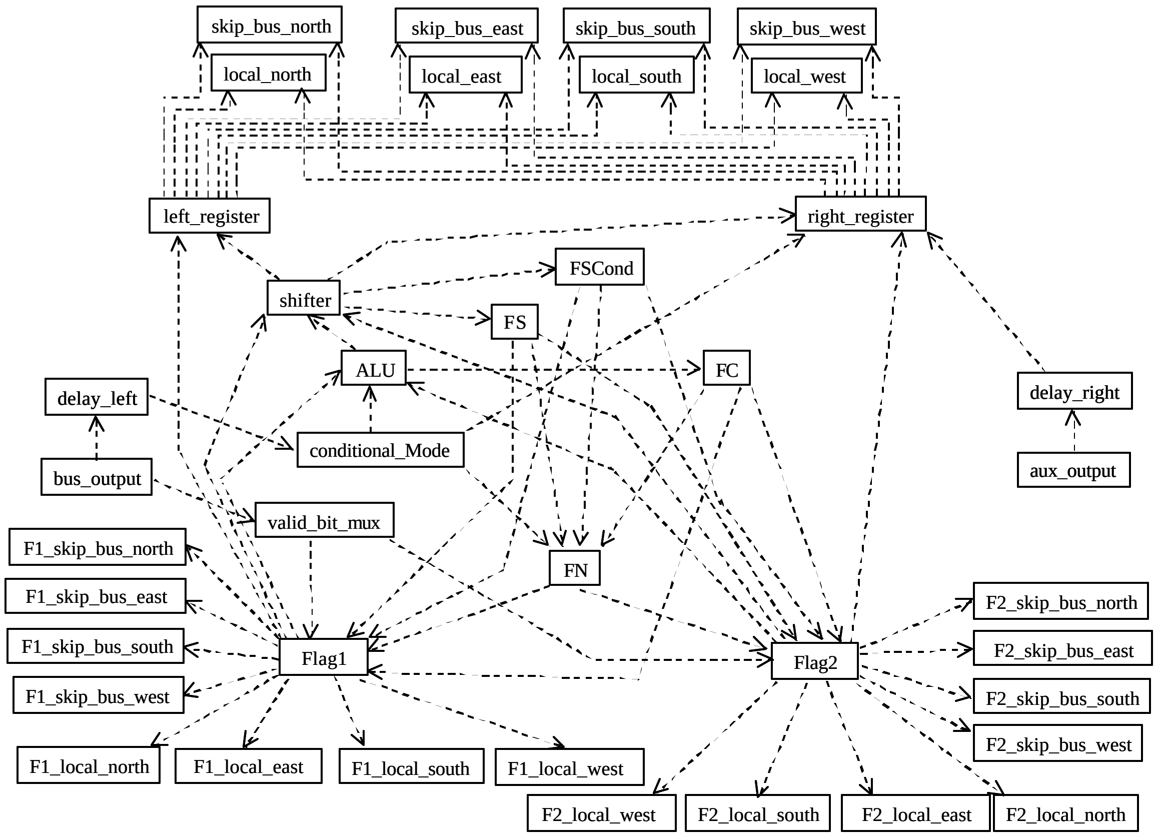


Figure 3.9 FU Computational Component Dependencies

The internal dependencies for data output for a functional unit are shown below in Figure 3.10. Note that the current output of the bus_output and aux_output objects are not dependent on any other object which are registers. The output of all local busses are dependent on the bus_output. To support bi-directional communications, each local_bus has two outputs, one for internal use and one for external use. When the external output is requested (from a local bus in another functional unit), the local bus will request the data held in the bus_output object. The dependencies used by skip bus objects are programmed, but each skip bus object has a single output function. To build up skip bus links across functional units, each skip bus link also has a pointer to skip bus object in another functional unit. The pointers to local busses and skip busses are established as part of Stallion parent object's connect_the_dots() function.

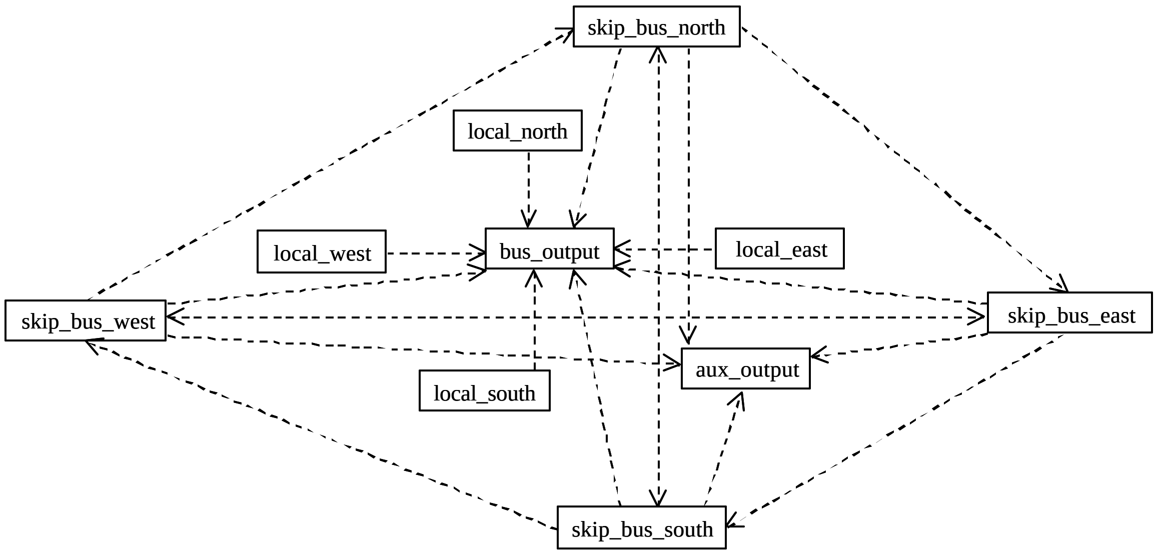


Figure 3.10 Data Output Dependencies

The internal dependencies for flag outputs for a functional unit are shown below in Figure 3.10. Note that unlike the data dependencies, the Flag1_out and Flag2_out are only clocked when the flag delays are enabled as per [14]. Thus the current output of the Flag1_out and Flag2_out objects may be dependent on other objects. The output of all local busses are dependent on Flag1_out and Flag2_out. To support bi-directional communications, each local_bus has two outputs, one for internal use and one for external use. When the external output is requested (from a local bus in another functional unit), the local bus will request the data held in the bus_output object. The dependencies used by skip bus objects are programmed, but each skip bus object has a single output function. To build up skip bus links across functional units, each skip bus link also has a pointer to skip bus object in another functional unit. The pointers to local busses and skip busses are established as part of Stallion parent object's connect_the_dots() function.

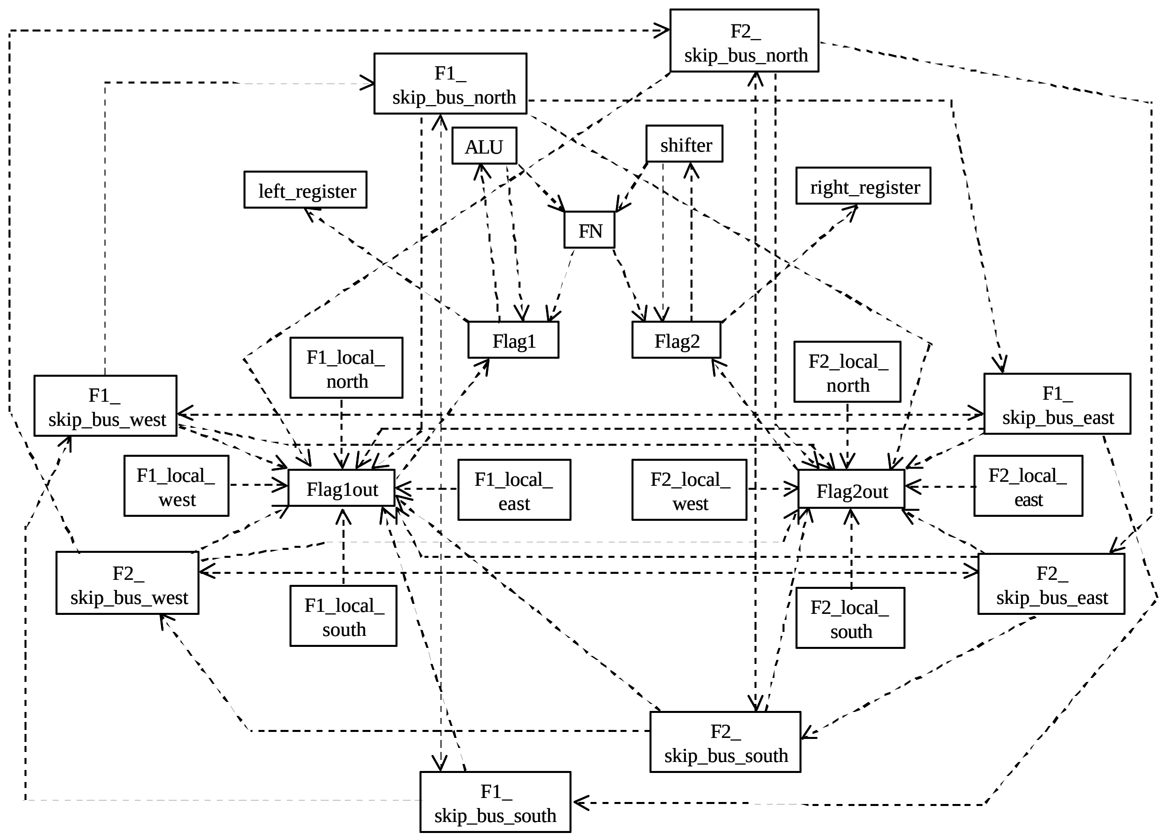


Figure 3.11 Flag Internal Output Dependencies

The functions in the Functional Unit object are shown below in Figure 3.12. The constructor function initializes the internal connections within a functional unit. The external connections are initialized as part of the Stallion parent object's connect_the_dot() function. The process function is responsible for calling the process functions of the clocked components in the functional unit. The update functions are used to update the various clocked registers. The power functions instruct the various components to update their power calculations. The program function parses the programming file and passes it to the functional unit's components.

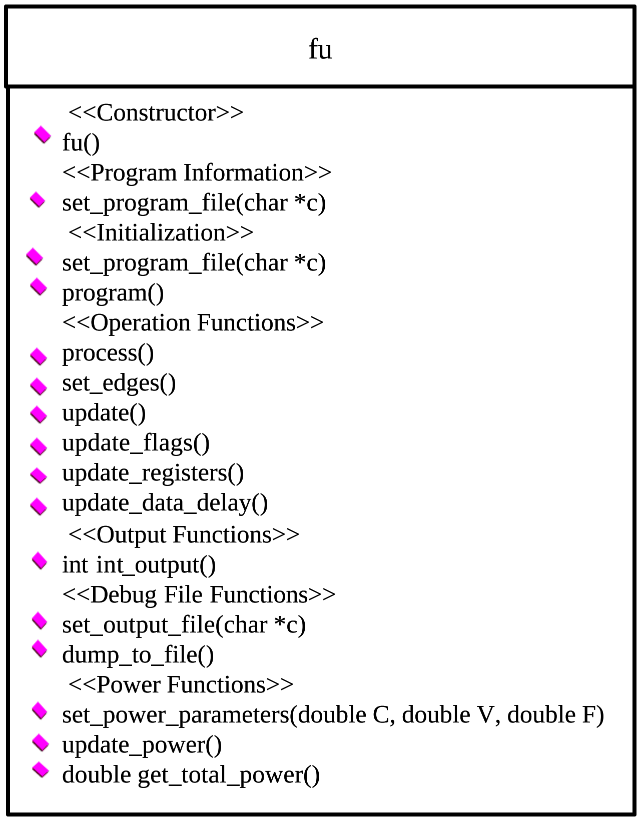
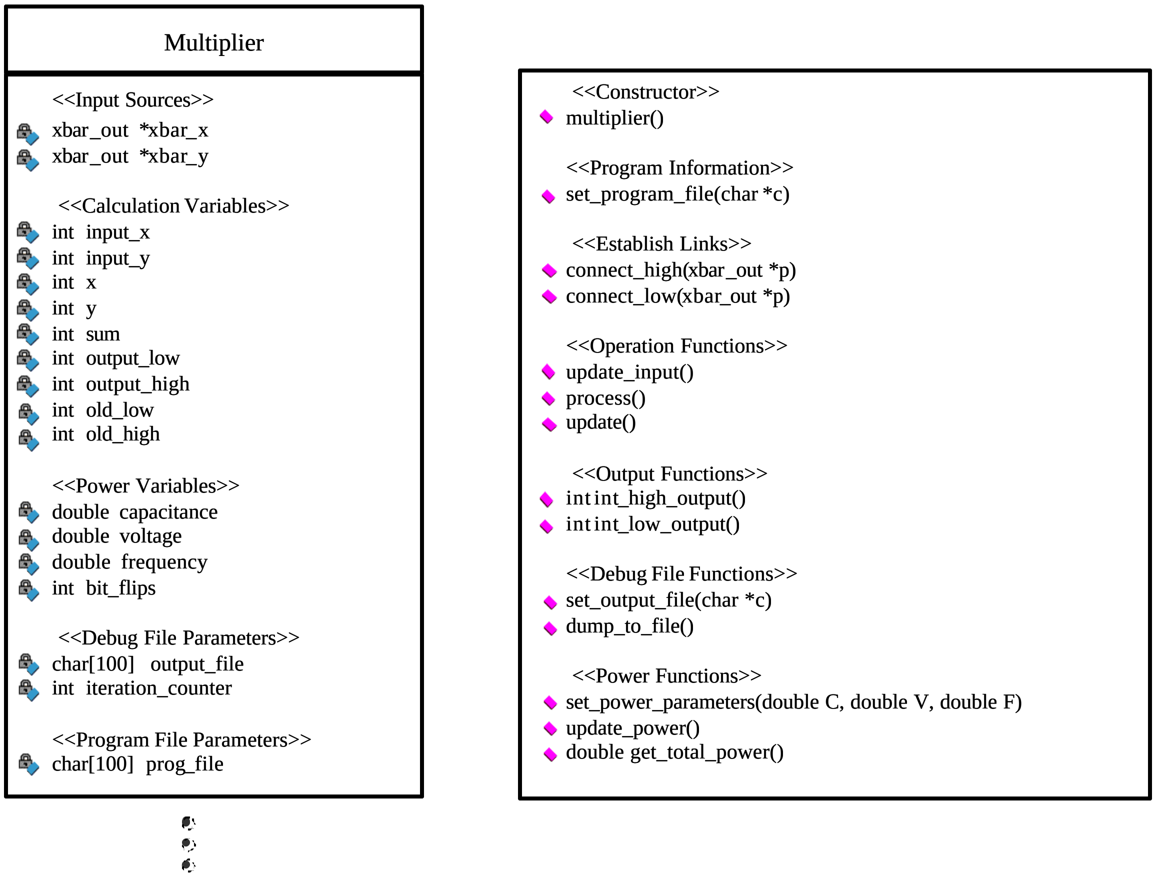


Figure 3.12 Functional Unit Functions

3.1.3.3 Multiplier

The multiplier object is a clocked object. The class diagram is shown below in Figure 3.13. The multiplier has pointers to two output crossbar nodes used as the inputs to the multiplier which are initialized by the Stallion parent object through the connect_high and connect_low functions. It calculates the unsigned product of its inputs using a shift and add implementation when instructed by the process() function. The update() is used to clock the transition between registers to support the two cycle unsigned multiplication. The int_high_output() and int_low_output() functions are used to retrieve the high (MSBs) and low (LSBs) outputs from the multiplier.


```
Multiplier

<<Input Sources>>
xbar_out  *xbar_x
xbar_out  *xbar_y

<<Calculation Variables>>
int  input_x
int  input_y
int  x
int  y
int  sum
int  output_low
int  output_high
int  old_low
int  old_high

<<Power Variables>>
double  capacitance
double  voltage
double  frequency
int  bit_flips

<<Debug File Parameters>>
char[100]   output_file
int  iteration_counter

<<Program File Parameters>>
char[100]  prog_file
```

```
<<Constructor>>
multiplier()

<<Program Information>>
set_program_file(char *c)

<<Establish Links>>
connect_high(xbar_out *p)
connect_low(xbar_out *p)

<<Operation Functions>>
update_input()
process()
update()

<<Output Functions>>
int int_high_output()
int int_low_output()

<<Debug File Functions>>
set_output_file(char *c)
dump_to_file()

<<Power Functions>>
set_power_parameters(double C, double V, double F)
update_power()
double get_total_power()
```

Figure 3.13 Multiplier Class Diagram

3.1.3.4 Data Port

The data port is implemented as a clocked object. The class diagram for the data port object is shown below in Figure 3.14. Each data port can be programmed to be an input source or an output source where data is either read from or written to a data file. Each data port has a pointer to an output crossbar node, though this is only used when the data port is programmed to operate in write mode. In terms of power calculations, it operates in a manner similar to the other Stallion components. Note that the data port object uses update_input() and output() functions rather than process() and output() although they serve similar functions.

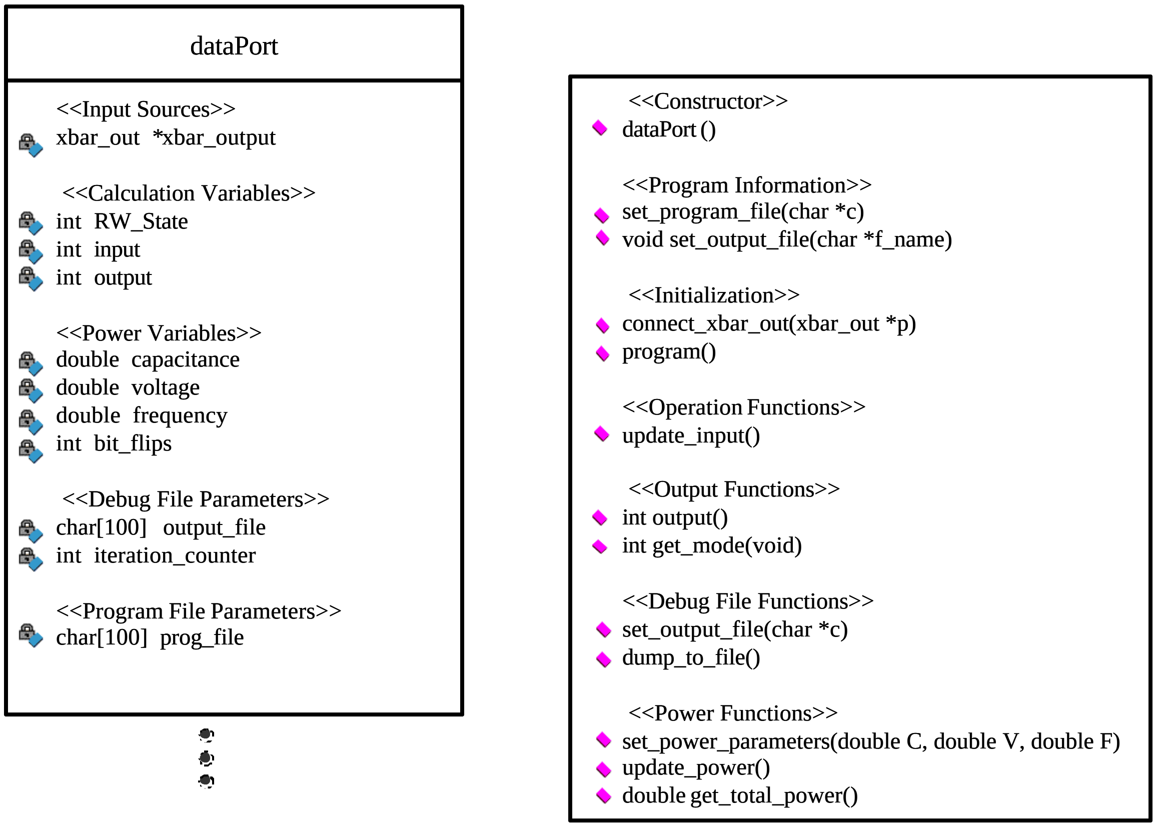


Figure 3.14 Data Port Class Diagram

3.1.3.5 Input Crossbar Node

The input crossbar node is implemented as a clocked object. The class diagram for an input crossbar node is shown below in Figure 3.15. The input crossbar node has a pointer to a functional unit and a pointer to a data port. These connections are made by the Stallion parent object in its connect_the_dots() function with the aid of the connect_fu() and connect_dp() functions. It registers its input in its output variable which is returned by the int_output() function and updated when the update_input() function is called. While the input node does generate debugging information, it does not generate power statistics which is the responsibility of the attached output crossbar nodes. Note that like the data port, its process() and update() functions are called by different names though they perform similar tasks. Note that in the simulation, the crossbar only supports point to point communications and does not support multicasting. This is a known bug, and if a solution would ever be needed could be readily solved by altering the set_program_file() function to accommodate the use of multiple output crossbar nodes.

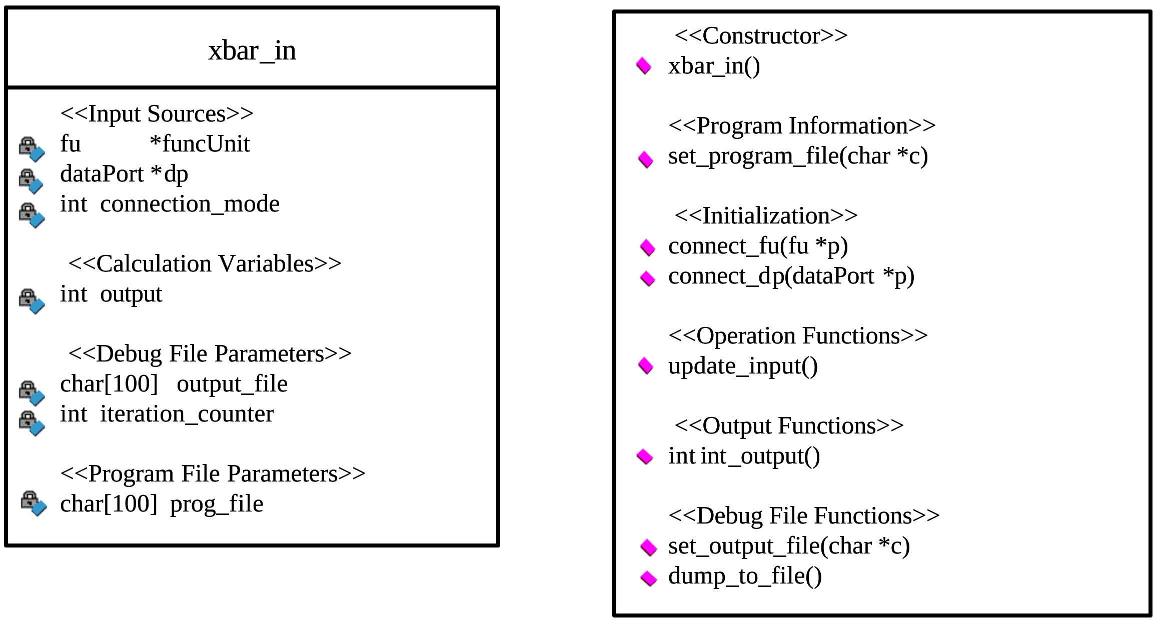


Figure 3.15 Input Crossbar Node Class Diagram

3.1.3.6 Output Crossbar Node

The output crossbar node is implemented as a logical object. Since the input node is clocked, the movement of data across the crossbar is also clocked. The class diagram for the output crossbar node object is shown below in Figure 3.16. In addition to more closely modeling the Stallion chip, its inclusion hides the input crossbar node from the functional units so that crossbar programming information need not be shared with the functional units. Each output crossbar object has a pointer to an input crossbar node that is established as part of the input crossbar node's programming in conjunction with the Stallion parent object. When an output is requested from an output crossbar, the output crossbar returns the output from its pointed to input crossbar node. The output and old_output variables are only used to perform power calculations.

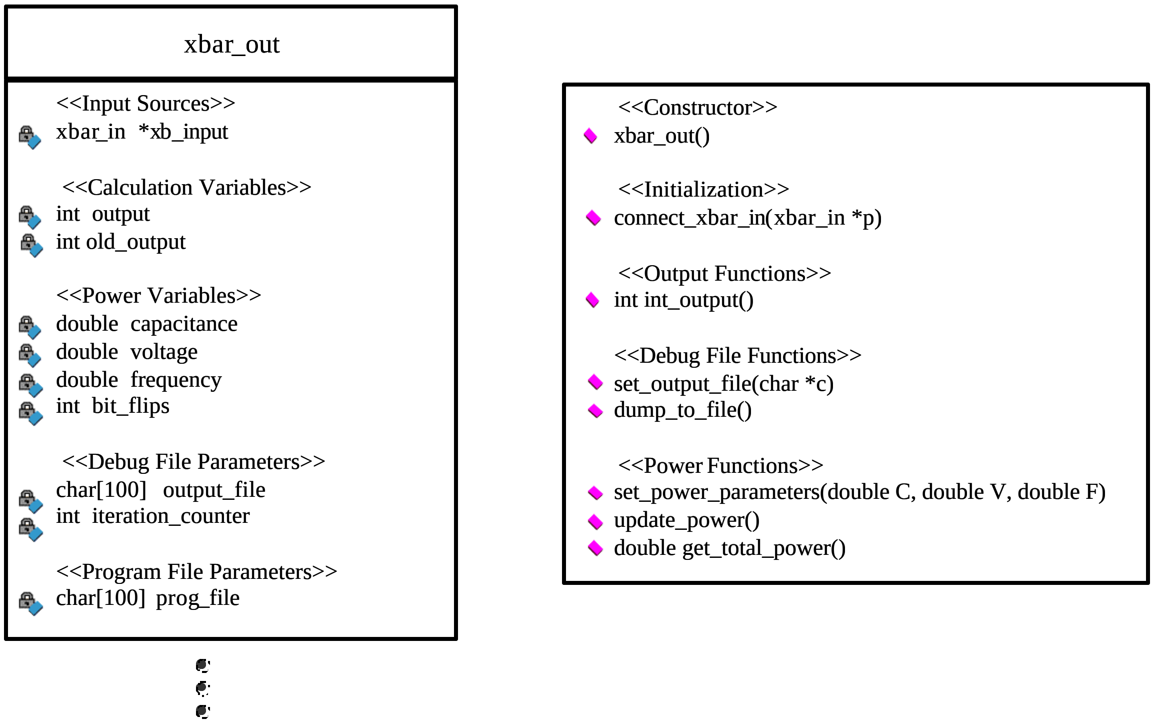


Figure 3.16 Output Crossbar Node Class Diagram

3.1.4 Configuring the Stallion Simulator

The Stallion Simulator is configured by passing it the command line arguments shown in Table 3.1. The number of iterations indicates the number of cycles that the simulator should execute. Each configuration file lists which components require programming and the names of each of the programming files and debugging files for each Stallion component that should be programmed. For instance, the functional unit configuration file can list up to 60 files for programming. If any value is passed to the sixth argument, the simulator will estimate the power consumed during its operation. If any value is passed to the seventh argument, then debugging information will not be collected during this simulation.

Table 3.1 Stallion Simulator Command Line Arguments

arg1	number of iterations
arg2	functional unit configuration file
arg3	multiplier configuration file
arg4	data port configuration file

arg5	crossbar configuration file
arg6 [optional]	perform power calculations
*arg*7 [optional]	disable debug mode

As previously stated, each of the Stallion level components are programmed through an independent file. The programming information stored in each of these files is expected to be formatted in the same manner as was initially required for the FPGA emulation of Stallion that was designed by the Configurable Computing Lab. Specifically, the format for each word of the emulation was specified as shown below in Figure 3.17.

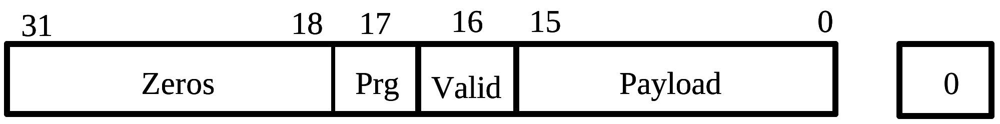


Figure 3.17 Stallion Emulation Packet Format

For components that require multiple programming packets (such as a functional unit), the simulator expects the packets to be arranged in the sequential order they would be inserted in a stream. The programming file for functional unit LA4 in the Acquisition configuration of the Single User Adaptive Receiver implementation is shown below in Figure 3.18. The programming packets for each of the components is identical to that described in Chapter 2.

Legend	FU Programming File
IFU Register	00038030 0
Configuration Register 1	00030000 0
Configuration Register 2	00030600 0
Configuration Register 3	0003350c 0
Configuration Register 4	00032000 0
Configuration Register 5	00030800 0
Configuration Register 6	00032000 0
Configuration Register 7	00030668 0

Figure 3.18 Example Programming File

3.1.5 Data Input / Output from the Stallion Simulator

To input data into and out from the simulator, an approach similar to that employed in debugging is again utilized. As the data ports have only a single subcomponent, the data register, the debugging file listed in the data port configuration file acts as either the input file or the output file for the data register depending on the read / write state specified as part of the data port's programming. When the data port is programmed to read, during each cycle the data port will read a single value from its input file to its data register. Likewise, if programmed to write, a single value will be written to the output file from the data register. The format of this file is described in 3.2.9.

3.2 Stallion Graphical Programming Interface

It was decided as part of this work that a programming interface for Stallion was required as the direct generation of the bit sequences required to program Stallion was laborious and prone to error. As Stallion is not well suited for a traditional compiler, and the amount of time required to build a good VHDL interpretive language compiler would be excessive, it was decided that the programming interface should be graphical, allowing direct control of each of the programmable components on Stallion. As the author was most familiar with MATLAB's Graphical User Interface (GUI) tools, the programming interface was developed in MATLAB. It was also desired that this programming interface be capable of performing the following tasks:

- storing and loading configurations
- providing direct control over the operation of each component and subcomponent in Stallion
- generating the programming bits needed to program Stallion within the same interface
- driving a simulation of the programmed configuration from the interface so that debugging could be more easily accommodated.

The following describes the GUI used to manage configurations, program Stallion, and to interface with the Stallion simulation.

3.2.1 Stallion Programming Interface

The primary programming interface is shown below in Figure 3.19. The interface consists of a number of interactive buttons that can be used to program different components of Stallion, change programming options, translate the graphical programming information into packets, simulate a "compiled" configuration, save a configuration, and load a configuration. For the Stallion programming interface and all component programming interfaces, the save and load buttons preserve directory structure during a configuration to further simplify the building of a configuration. The compile button operates as a macro for calling the compile option for each of the components that have been generated. The simulate button is another macro button that is used to format information for the simulator and run the simulation.

The component buttons are arrayed in a somewhat intuitive manner wherein the traditional 3-D depiction of Stallion (shown in Figure 2.19), has been projected onto a two dimensional surface. The data ports are represented by a square and placed at the far left and the far right of interactive diagram to emphasize their use as the input and output of Stallion. Beside each data port button is a button for its corresponding input crossbar node and its output crossbar node. The input crossbar node is represented by a wide button and the output crossbar node by a narrow button. The functional units are represented by a square, usually blank button, and arranged into high and low meshes. To meet intuitive expectations, the high mesh is at the top and the low mesh at the bottom of the programming interface. To provide a unique identity to each element in the mesh, the functional units are labeled by mesh (High or Low) a row (A-D) and column (1-8). Note this differs from the addressing scheme used internally to Stallion. However, this is only the concern of the parser and the process of address conversion is hidden from the developer. Also in each mesh are the multipliers. These are indicated by a 'X' written in the button. At the top of each mesh are 16 output crossbar nodes which are represented as buttons with half the width of the functional unit button. Above each output crossbar node for the meshes a 'L' or a 'S' is written to indicate whether the output node is connected to a local bus or a skip bus, respectively. When an output crossbar node has been assigned to an input node, it is labeled with an 'A'. At the bottom of each mesh are 8 buttons that are used to program the input crossbar nodes attached to each mesh. When a component has been programmed and the configuration saved or "Generated," a "G" is placed in the component. When a component has also been compiled, this designation is changed to "GC." Each one of these graphical components that launch context specific programming windows upon clicking on the component. These are described in detail as follows.

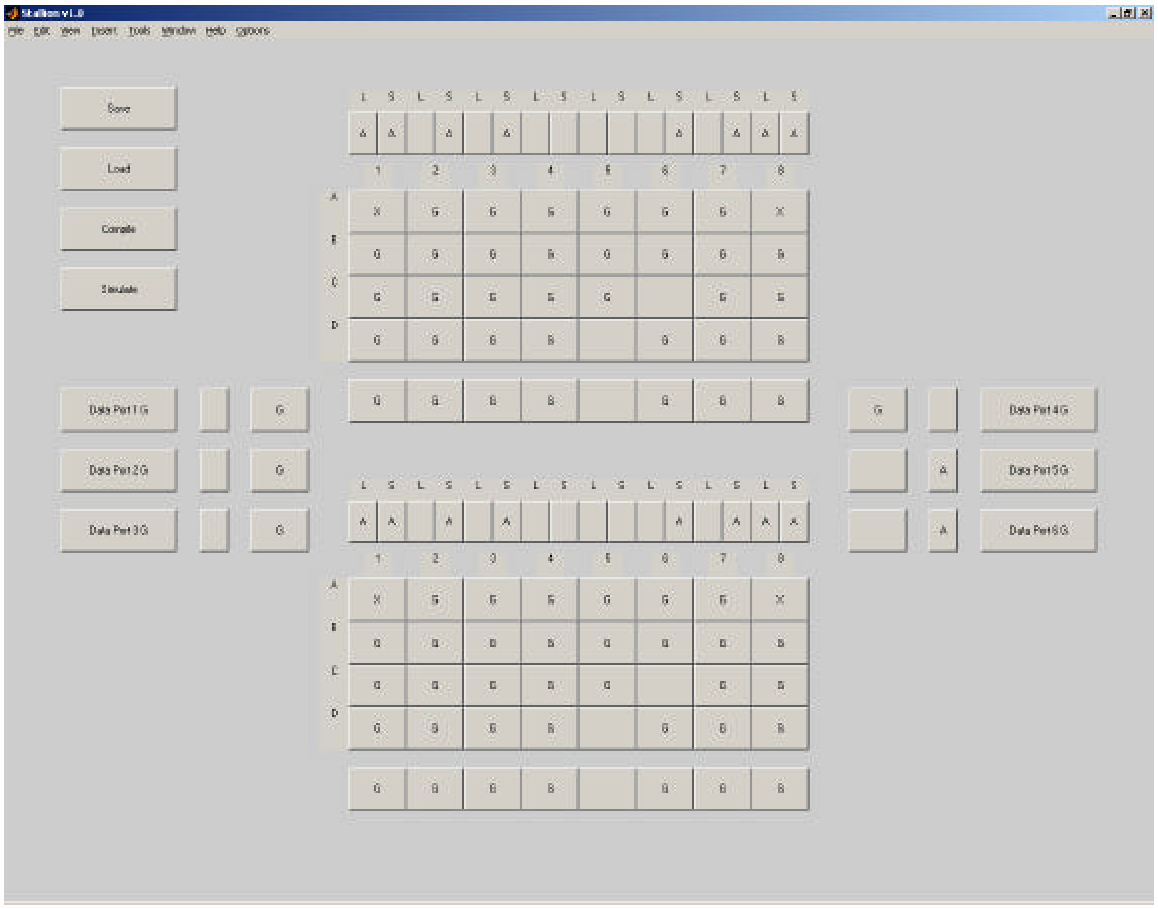


Figure 3.19 Primary Stallion Programming Interface

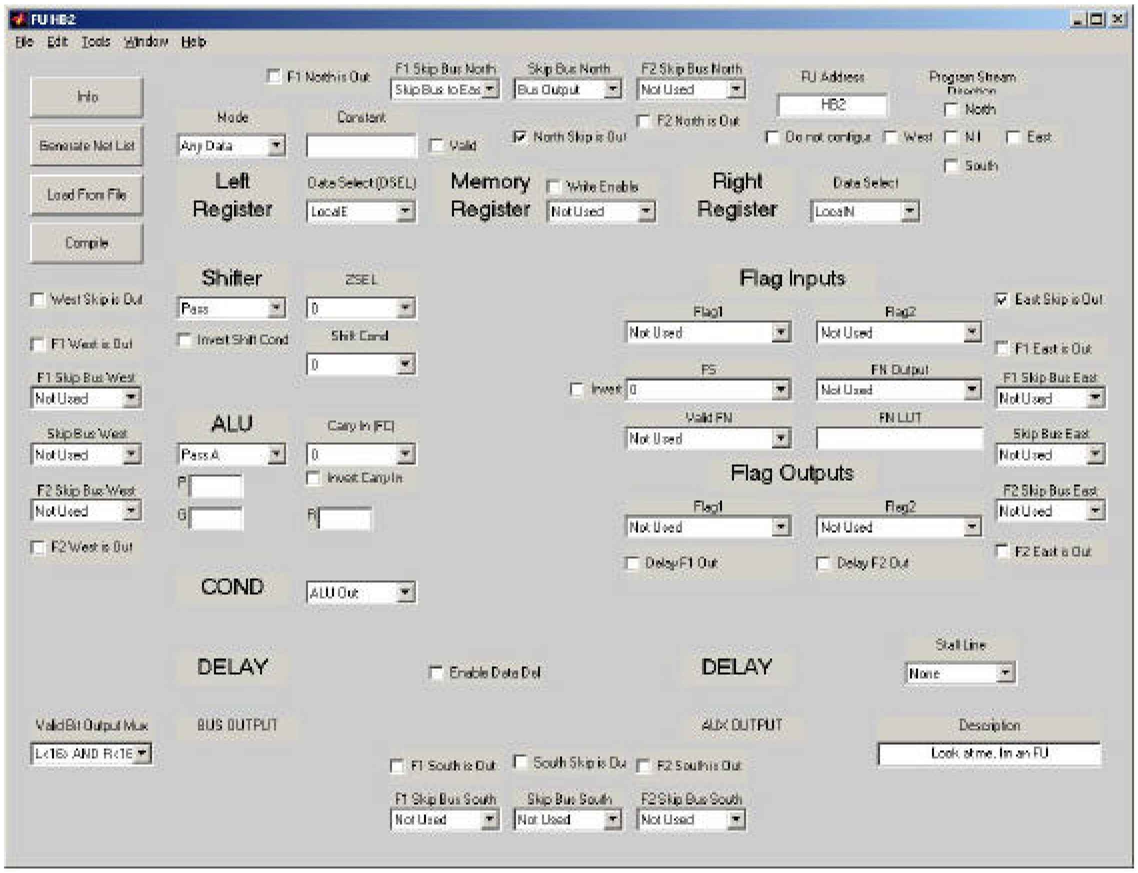
3.2.2 Functional Unit Programming Interface

Figure 3.20 shows the functional unit programming interface. This is launched whenever one of the boxes in either mesh is left clicked on. If previous data has been stored (which would be indicated by a G), the functional unit programming interface automatically loads the previously saved data. The components within the functional unit are arranged so that processing flows from top to bottom. Components that perform an operation logically on the edge of the functional unit, such as skip busses are arrayed at the edge of the diagram and grouped such that traditional cartographical relationships are preserved (north is up). Most of the programmable elements within the functional unit interface have a dedicated pull down menu of the available program options for that component.

For constants and LUTs, an edit box is provided so that greater flexibility is achieved. In each edit box used for programming, it is expected that the desired number will be written in decimal notation. Although hexadecimal may be the more traditional representation for those involved in digital design, communications engineers, for whom this interface was intended, are more accustomed to decimal notation.

The ALU contains both a pull down menu and edit boxes. The pull down menu allows the ALU to be programmed for any of the operations that the simulated ALU understands. For operations not supported by the simulation, the edit boxes are used to set the P,G, and R registers of the ALU. There is also space provided for commenting on the specific operation that the functional unit is being programmed to perform, especially valuable for complex simulations. These comments in no way affect the programming information generated.

Whenever it was possible to identify associated programming decisions, programming elements were given "intelligence" to adjust their setting in response to changes to programming decisions made within the FU. For instance when increment is chosen for the ALU, the carry in flag is set to 1, and when any of the skip bus direction check boxes is toggled, the opposite check box is correspondingly toggled. Except for the toggling of directional check boxes (which could possibly lead to programming ambiguities if this were not done), any of these toggling decisions can be overridden. Each functional unit is also labeled according to the mesh convention, *e.g.* LA4, to help the programmer keep track of which functional unit is being programmed when multiple windows are open.

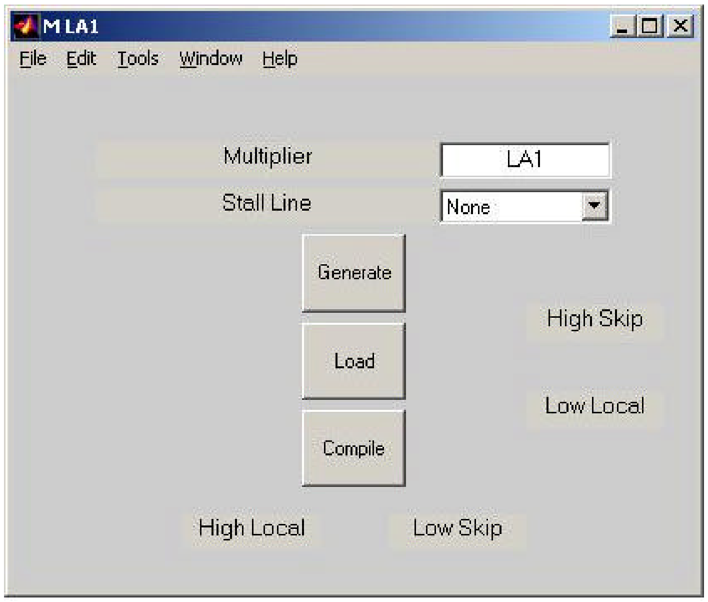


F1 North is Out: ☐
F1 Skip Bus North: Skip Bus to Eas
Skip Bus North: Bus Output
F2 Skip Bus North: Not Used
FU Address: HB2
Program Stream Direction: ☐ North ☐ West ☐ N I ☐ East ☐ South
Do not configu: ☐
Mode: Any Data
Constant:
Valid: ☐
North Skip is Out: ✓
F2 North is Out: ☐
Data Select (DSEL): LocalE
Write Enable: ☐
Not Used
Data Select: LocalN
ZSEL: 0
Shifter: Pass
Invert Shift Cond: ☐
Shift Cond: 0
West Skip is Out: ☐
F1 West is Out: ☐
F1 Skip Bus West: Not Used
Skip Bus West: Not Used
F2 Skip Bus West: Not Used
F2 West is Out: ☐
ALU: Pass A
P:
G:
Carry In (FCI): 0
Invert Carry In: ☐
R:
COND: ALU Out
Flag Inputs Flag1: Not Used
Flag Inputs Flag2: Not Used
Invert: ☐
FS: 0
FN Output: Not Used
Valid FN: Not Used
FN LUT:
East Skip is Out: ✓
F1 East is Out: ☐
F1 Skip Bus East: Not Used
Skip Bus East: Not Used
F2 Skip Bus East: Not Used
F2 East is Out: ☐
Flag Outputs Flag1: Not Used
Flag Outputs Flag2: Not Used
Delay F1 Out: ☐
Delay F2 Out: ☐
Enable Data Del: ☐
Stall Line: None
ValidBit Output Mux: L<16> AND R<16
Description: Look at me. Im an FU
F1 South is Out: ☐
South Skip is Ou: ☐
F2 South is Out: ☐
F1 Skip Bus South: Not Used
Skip Bus South: Not Used
F2 Skip Bus South: Not Used

Figure 3.20 Functional Unit Programming Interface

3.2.3 Multiplier Context Windows

A context window opens when a multiplier button is clicked. This window is shown below in Figure 3.21. While also providing a method for selecting a stall line (stall lines are a feature not supported in the simulation), the primary purpose of the Multiplier context window is to serve as a context window for reminding the programmer which outputs correspond to the upper 16 bits and which correspond to the lower 16 bits. Multipliers are also labeled according to the functional unit mesh convention.

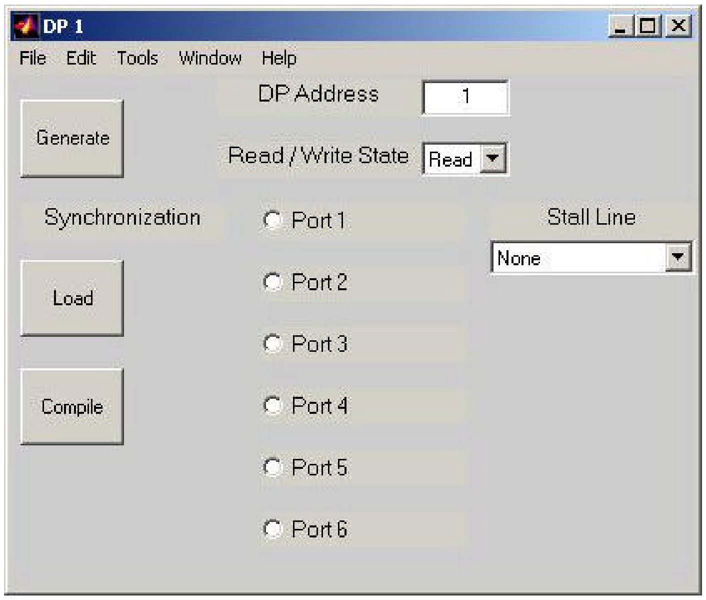


Multiplier: LA1
Stall Line: None

Figure 3.21 Multiplier Programming Interface

3.2.4 Data Port Programming Interface

A data port programming window is opened when a data port button is clicked. This window is shown below in Figure 3.22. This window is used to configure whether the data port is an input port (read) or an output port (write). Although not a part of the simulation, it is also possible to specify with which data ports to synchronize which may be important for implementations on Stallion.

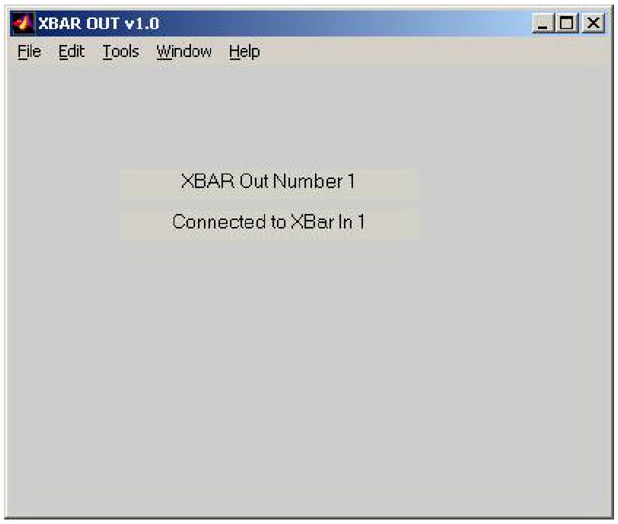


DP Address: 1
Read / Write State: Read
Stall Line: None

Figure 3.22 Data Port Programming Interface

3.2.5 Output Crossbar Node Interface

Shown below in Figure 3.23, each output crossbar shows the input crossbar node to which it is connected. This programming interface only supports point-to-point connections and will "disconnect" an output node from its input node when a new output node is connected to the input node by removing the 'A' from the contextual launch button and the reference within the interface.



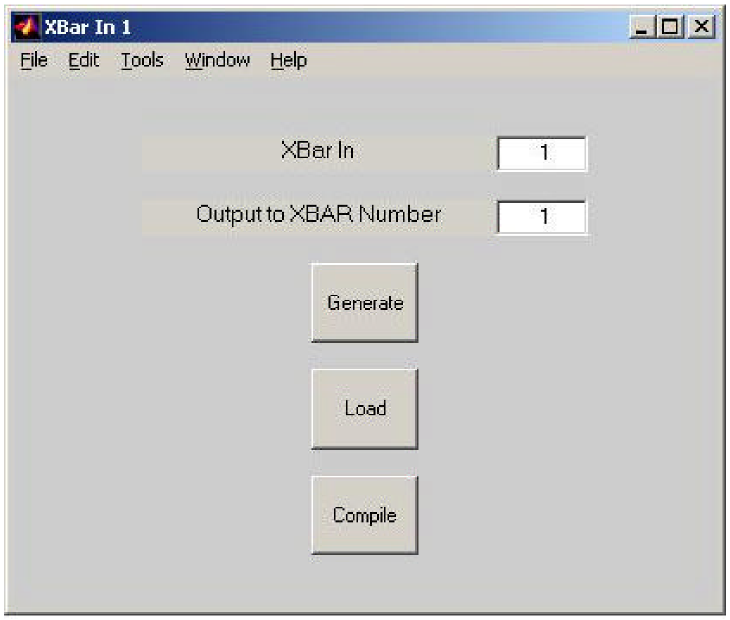
XBAR Out Number 1

Connected to XBar In 1

Figure 3.23 Output Crossbar Node Interface

3.2.6 Input Crossbar Node Programming Interface

Shown below in Figure 3.24, each input crossbar node interface is used to generate the various desired connections within the crossbar. Each input crossbar node specifies the output crossbar node to which it should be connected.



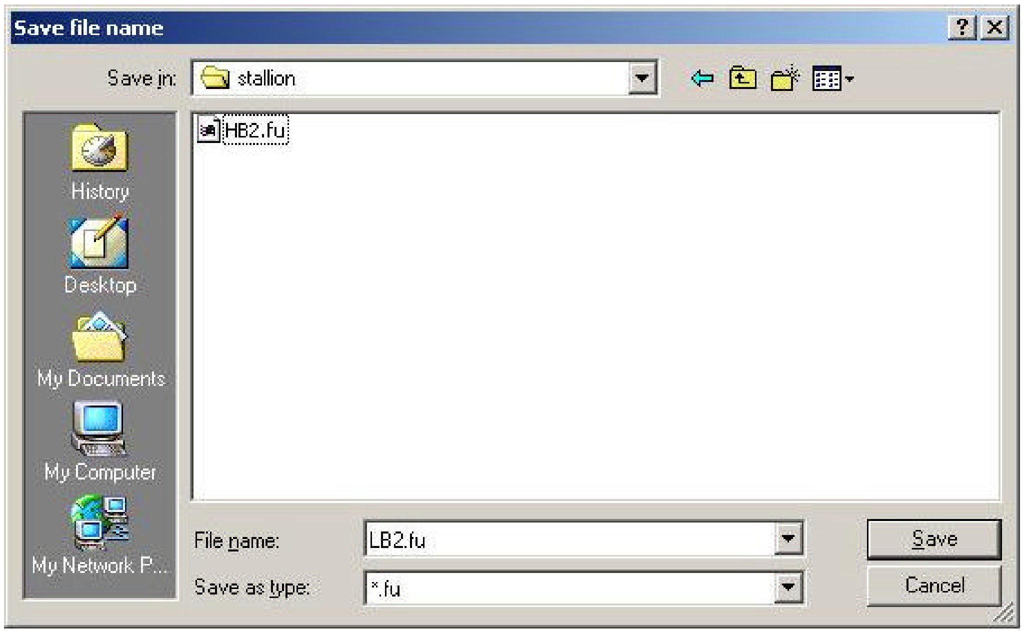
XBar In 1

Output to XBAR Number 1

Figure 3.24 Input Crossbar Node Programming Interface

3.2.7 Stallion Programming Interface Data Storage

When the Generate button is clicked in any of the component programming interface windows, *e.g.* functional unit, data port, input crossbar, a context window similar to the one shown in Figure 3.25 is launched. Based on stored directory information, the interface will suggest a storage location identical to the most recently used location, since the beginning of the current programming session, for storage of data for that type of component. The interface will also suggest a contextual name and file extension.



HB2.fu

File name: LB2.fu

Save as type: *.fu

Figure 3.25 Stallion Component Data Storage Interface for a Functional Unit

After a file location has been specified, the interface will produce a text file of labels and attributes specific to that type of component. For subcomponents, this consists of all of the programming information specifiable for that component as well as any comments that may have been added in that interface. This can be used to quickly reestablish the status of context menus, edit boxes and checkboxes. For the Stallion interface, the text file consists of a listing of directories, files, and status variables (which components have been saved, which crossbar connections have been assigned) that will be required to recreate the programming status of a particular configuration at a later date. An excerpt from example functional unit data storage file is shown below in Figure 3.26. Note that in the figure, each of the values correspond to a particular menu choice, or check box status associated with that programming tag.

```
FUAddress 58
SingleWord 0
IFUNorth 0
IFUWest 0
IFUEast 1
IFUSouth 0
Mode 0
LeftDSEL 1
RightDSEL 0
Shifter 0
ZSEL 1
F2DelayFlag 0
F1DelayFlag 0
ShiftCond 0
InvertShiftCond 0
ALUOp 9
```

Figure 3.26 Functional Unit Data Storage File Excerpt

When pressed, the Load button will start the load callback routine of that particular component. After prompting for the desired file, the information stored in the data storage file will be used to reinitialize the various components of the programming interface. This same routine is also called when the programming interface for a previously generated component (denoted by a 'G') is launched. However, the file naming information used for this interface will be automatically supplied, and the interface will appear exactly as it was last saved at launch.

3.2.8 Data File Translation (Compile)

Translation of these text-based data files into the programming format described in Section 2.2 is performed when the "Compile" button is depressed. Again this launches a storage window similar to the one shown in Figure 3.25 wherein the location of recently stored files serve as a starting location for the resultant file and context information serves as a suggestion for file names. After determining a file name, the compile callback routine automatically calls a C-language translator that parses the data file and produces a file that contains programming information in the format described in Section 2.2 based on the context of the calling function. For instance the functional unit programming interface instructs the translator to convert the text file to a file suitable for the programming of a functional unit on Stallion. An example output for a functional unit is shown in Figure 3.27.

```
00038033 0
00030000 0
00030600 0
0003000f 0
00032000 0
00030800 0
00032000 0
00030cc0 0
```

Figure 3.27 Example Translated File

When the Compile button is selected in the Stallion programming interface, all currently generated components in the configuration will be compiled in a process similar to the one just described. However, this process will not prompt for a filename for each component. Rather, it will use the file name chosen for any previously compiled versions, if present, or it will use the default name. The option for compiling all of the components removes the tedium associated with the development of a large configuration. The ability to compile a single component reduces development time when only minor changes need to be made to a configuration that is being debugged.

3.2.9 Configuration Simulation (Simulate)

The Stallion programming interface also provides a method for simulating a configuration through the Simulate button. When clicked, the interface shown below in Figure 3.28 appears prompting for the desired number of iterations in the simulation. Using this input information and the stored directory information, the interface automatically calls the Stallion Simulation described in 3.1.1 with the appropriate arguments.

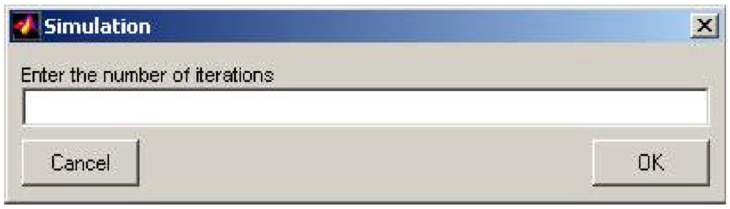


Figure 3.28 Simulation Iteration Interface

Note that in order to inject a data stream into a data port, a data port ".out" file must be written and stored in the configuration's data port directory with the desired input data written in decimal separated by white space delimiters. This file should be given the same name as was used to compile the particular data port, but with a ".out" extension instead of the ".e" extension. An example is shown below in Figure 3.29.

```
0 65536 27 8 5 32768 5 3 0 0
```

Figure 3.29 Example Data Port .out File

The debugging information for each of the components in the simulated configuration is stored in the compiled files directories and may be viewed using any text editor.

3.3 Programmable Controller Chip Object

In order to fully explore the applicability of custom computing machines to software radio, to perform a more "realistic" comparison of the custom computing machines to traditional processing solutions, and to evaluate the feasibility of the concepts employed in the Configuration Layer of the Layered Radio Architecture, a programmable controller object was created as part of this work. Specifically, this controller chip object was intended to perform the following:

- support the implementation of waveforms through the paging of configurations into the Stallion simulation
- act as a common point for collecting performance statistics across configurations
- demonstrate the feasibility of the following concepts employed in the configuration layer by supporting
    - the use of nonembedded variables for storing the intermediate arrays of data used between configurations

- the use of embedded variables for modifying the operation of configurations.

The following sections describe how this was accomplished.

3.3.1 Controller Chip Object Overview

The construction of the controller chip as a collection of objects allows for a logical collection of a component's data and operations into a single entity to simplify management and code development – a traditional use of OOP. Specifically, the controller chip object consists of the components shown in Figure 3.30. The data storage objects, supplemented by the big_array objects, are used to implement the operations of NEVs. The configuration objects are used to manage the loading of new configurations into Stallion and to manage the operation of embedded variables.

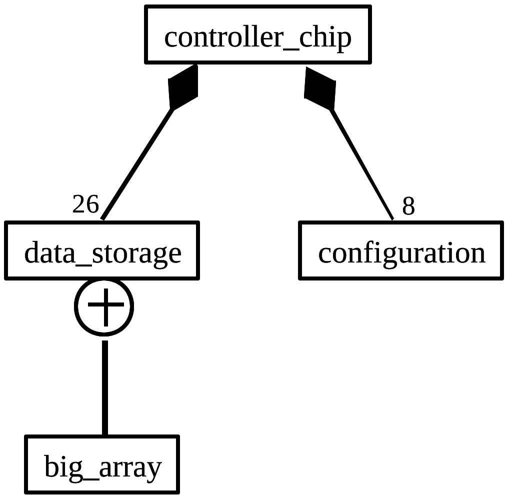


Figure 3.30 Controller Chip Component Diagram

The class diagram for the controller_chip object is shown below in Figure 3.31. The data_source_I and data_source_Q are input file streams are used to model the continuous input of data from an I-Q ADC. These files are expected to contain a sequence of white space delimited doubles. This was to support data generated from MATLAB fading simulations where doubles are the native type used in MATLAB. In order to translate these doubles into a format useable by Stallion simulation, which primarily uses 16 bit integers, these input are scaled by the scaling factor and cast into integers. The responsibility for executing this operation is split between the controller_chip which is responsible for multiplying inputs by the proper scaling factor, and the data storage objects which ensure that the inputs occupy the proper 16-bit field width. The controller_chip contains 26 general purpose NEVs, a NEV dedicated for temporary variables, and two NEVs dedicated for further buffering the input data necessary for synchronization operations. The controller chip contains a number of functions including load_configuration() which is used to make a configuration the active configuration, run() which is used to start a simulation of the active configuration (named current), and methods for interfacing NEVs to files.

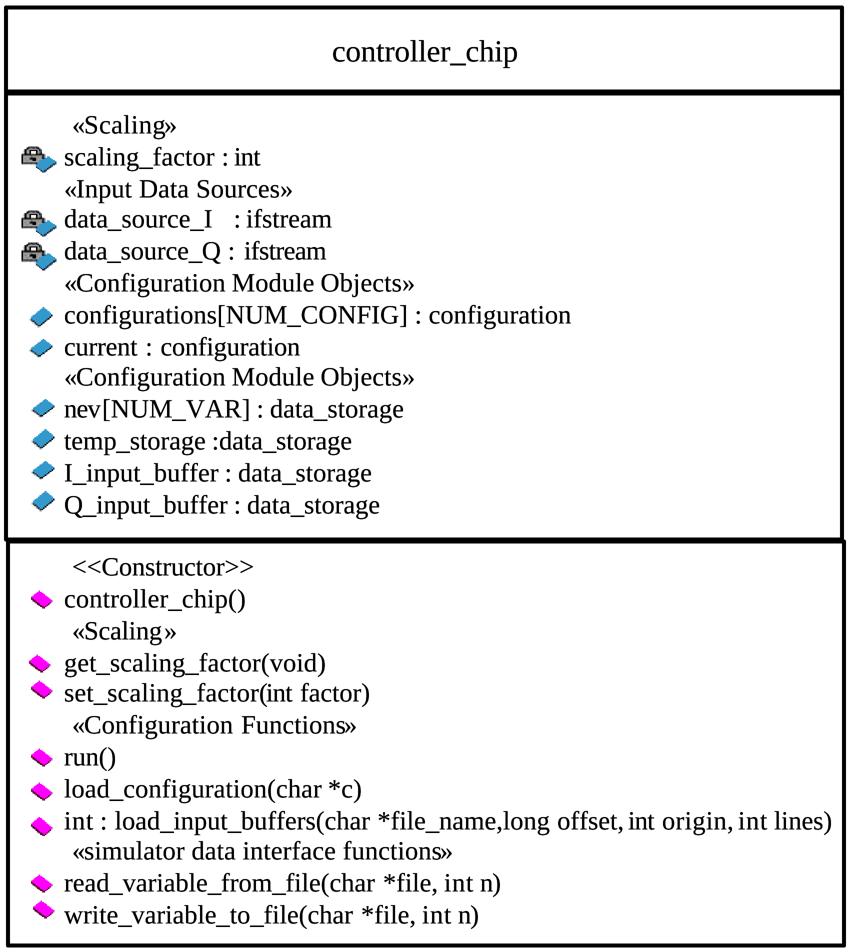

controller_chip
«Scaling»
scaling_factor : int
«Input Data Sources»
data_source_I : ifstream
data_source_Q : ifstream
«Configuration Module Objects»
configurations[NUM_CONFIG] : configuration
current : configuration
«Configuration Module Objects»
nev[NUM_VAR] : data_storage
temp_storage :data_storage
I_input_buffer : data_storage
Q_input_buffer : data_storage
<<Constructor>>
controller_chip()
«Scaling»
get_scaling_factor(void)
set_scaling_factor(int factor)
«Configuration Functions»
run()
load_configuration(char *c)
int : load_input_buffers(char *file_name,long offset, int origin, int lines)
«simulator data interface functions»
read_variable_from_file(char *file, int n)
write_variable_to_file(char *file, int n)

Figure 3.31 Controller Chip Class Diagram

3.3.2 Configuration Object

The Configuration Object is responsible for the following tasks:

- managing the information required to load a configuration onto the Stallion simulator
- handling the updating of embedded variables within a configuration

A class diagram for the configuration object is shown in Figure 3.32. The configuration object contains all the components necessary for loading a configuration onto the Stallion simulation including the names of a configurations' information files for functional units, data ports, crossbars, and multipliers and the number of iterations that a configuration should run. Additionally, the configuration contains a component and methods for allowing a configuration to be named so that configurations can be managed in a more easily read manner with commands in the controller_chip object such as load_configuration(configuration name). There are methods for individually setting these parameters as well as methods for copying an existing configuration's information or for loading this information from a file. The configuration object also contains a method for altering the value of an embedded variable through the use of the change_value_of_constant() function.

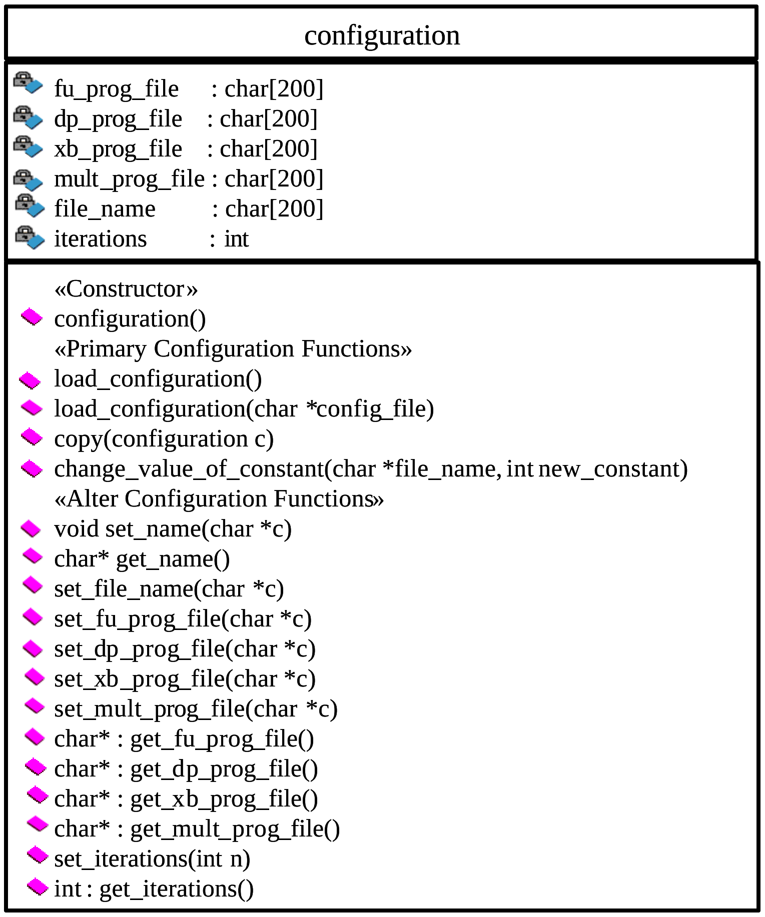

configuration
fu_prog_file : char[200]
dp_prog_file : char[200]
xb_prog_file : char[200]
mult_prog_file : char[200]
file_name : char[200]
iterations : int
«Constructor»
configuration()
«Primary Configuration Functions»
load_configuration()
load_configuration(char *config_file)
copy(configuration c)
change_value_of_constant(char *file_name, int new_constant)
«Alter Configuration Functions»
void set_name(char *c)
char* get_name()
set_file_name(char *c)
set_fu_prog_file(char *c)
set_dp_prog_file(char *c)
set_xb_prog_file(char *c)
set_mult_prog_file(char *c)
char* : get_fu_prog_file()
char* : get_dp_prog_file()
char* : get_xb_prog_file()
char* : get_mult_prog_file()
set_iterations(int n)
int : get_iterations()

Figure 3.32 Configuration Class Diagram

3.3.3 Data Storage Object

The Data Storage Object is responsible for the following tasks:

- managing the movement of data to and from the Stallion simulation
- supporting the discarding of specified packets
- supporting the insertion of valid bits in the proper locations
- ensuring data is in a format useable by the Stallion simulation (16-bit integer)

A class diagram for the configuration object is shown in Figure 3.33.

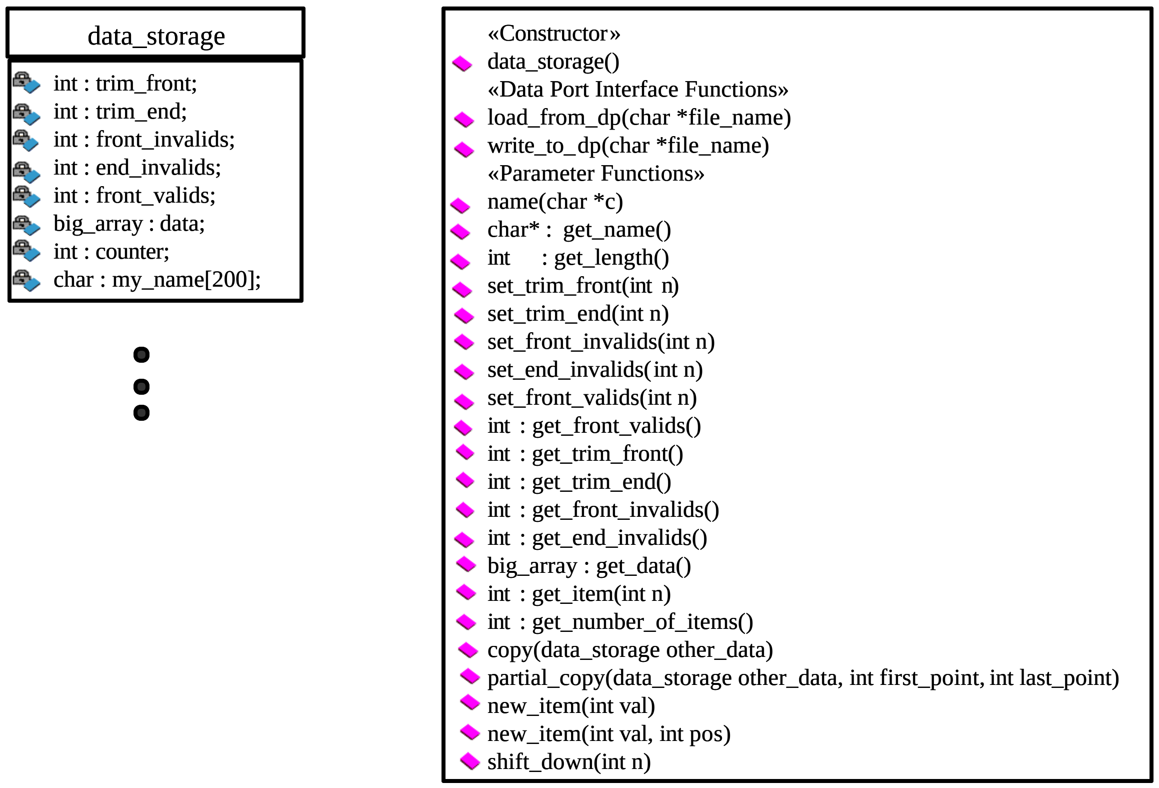


Figure 3.33 Data_storage Class Diagram

The data storage object implicitly operates using the format shown in Figure 3.34 for moving the data between the dataports and a NEV. As there will necessarily be some latency before the first actual result returns from Stallion, the trim_front() function is used to discard the first *n* number of packets. The various valid and invalid functions used to append a specified number of valid zeros or invalid zeros at the beginning or end of a data stream. This is particularly useful for resetting certain counters that may be instantiated in a configuration implemented on Stallion.

Trim Front	Front Invalids	Valids	End Invalids	Trim End

Figure 3.34 Data_storage Data Format

Each data_storage class also has a big_array object that it uses to manage the actual data to be stored in the data storage class, keep track of the actual number of elements used, and to allow for items to be added to the NEV without having to independently track the total number of items currently in the NEV.

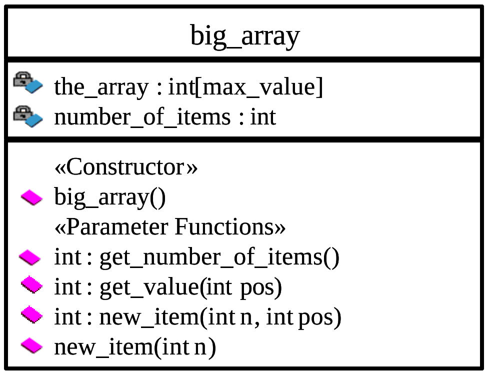


Figure 3.35 Big_array Class Diagram

Chapter 4

4 Implementation of a Single User Receiver

As part of satisfying the first primary goal of this work, the demonstration of the feasibility of a CCM within the Layered Radio Architecture, a single user adaptive receiver was implemented. Additionally, performance statistics (cycle counts) were also collected about this implementation in order to demonstrate any gains in cycle counts as well as the fidelity of the implementation. This chapter describes the implementation developed using the tools described in the previous chapter and the implementation's performance metrics.

4.1 Single User Receiver Algorithmic Description

The single user adaptive receiver was initially developed as part of the thesis work of Michael Hosemann [19]. The receiver was designed to work with the components to be used in the GloMo Soft Radio shown below in Figure 4.1. The GloMo soft radio consists of a Slaac FPGA development board that consists of three Xilinx Virtex 1000 FPGAs and can take its input from either a host PC for simulation or from the Rockwell Miniature Radio Codec (MRC). It was envisioned that one FPGA would act as the SRI layer and the configuration layer, one FPGA would be an emulation of Stallion, and the third FPGA would be a turbo decoder (as part of the GloMo project). It was ultimately the requirement that the Configuration Layer and SRI Layer be implemented on a Virtex FPGA that led to its design. The MRC was known to perform noncoherent demodulation which results in a residual frequency in the downconverted signal.

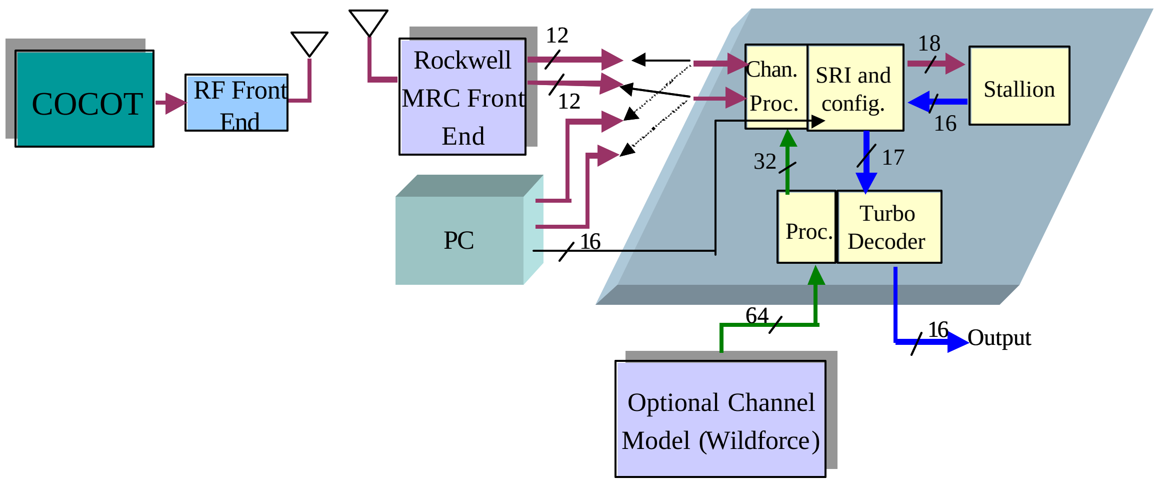


Figure 4.1 GloMo Soft Radio Block Diagram

The transmitted signal was a differentially encoded Direct Sequence Spread Spectrum (DSSS) signal and with a m-sequence spreading code of length 15. The single user receiver is a CDMA-like receiver that is designed to equalize the channel and remove the effects of residual frequency. A block diagram of the single user adaptive receiver is shown below in Figure 4.2. The receiver oversamples the input signal by a factor of four to help achieve and maintain chip synchronization. The primary component in the single user receiver is the complex filter which is used for despreading, equalization, and synchronization. The output symbol of the complex filter, $y(n)$ is given by

$$y[n] = \vec{w}^{H}[n]\vec{r}[n]$$

where w is a vector of filter weights, $\vec{r}[n]$ is the vector of received samples associated with symbol n and the superscripted H indicates the Hermitian operation. This is then differentially decoded to form the output symbol, $z[n]$, by evaluating the following equation

$$z[n] = y[n] \times y^{*}[n-1]$$

where the superscripted * indicates the complex conjugate operation. The filter weights are updated for symbol $n + 1$ according to

$$w(n+1) = w(n) + me^{*}y^{*}(n-1)r(n)$$

where $w(n)$ is the vector of weights for the current symbol, *m* is the step size, *e* is the error in the current symbol, During decision directed operation, the error is calculated as the difference between the desired differentially symbol and $z[n]$. During blind mode the error is calculated as

$$e_R[n] = \mathrm{sgn}(z[n]) - z[n]$$

$$e_I[n] = -z_I[n].$$

As it is expected that there will be a timing mismatch between the transmitter and the receiver, the timing of the signal will drift in time and thus the energy of the adaptive filter weights will drift as well. If the filter weights drift too far to one side, the synchronization with a symbol will be lost and catastrophic errors will result. To limit the effects of this process, the adaptive filter is twice the length of the oversampled spreading code and the weight vector is periodically correlated with the spreading code in order to determine where the energy of the weight vector lies. When the position is determined to be close to the edge of the vector, an indication of an impending catastrophic failure, the weights are accordingly shifted and the received samples delayed or advanced as required.

The operations associated with the Single User Adaptive Receiver are summarized in Figure 4.2.

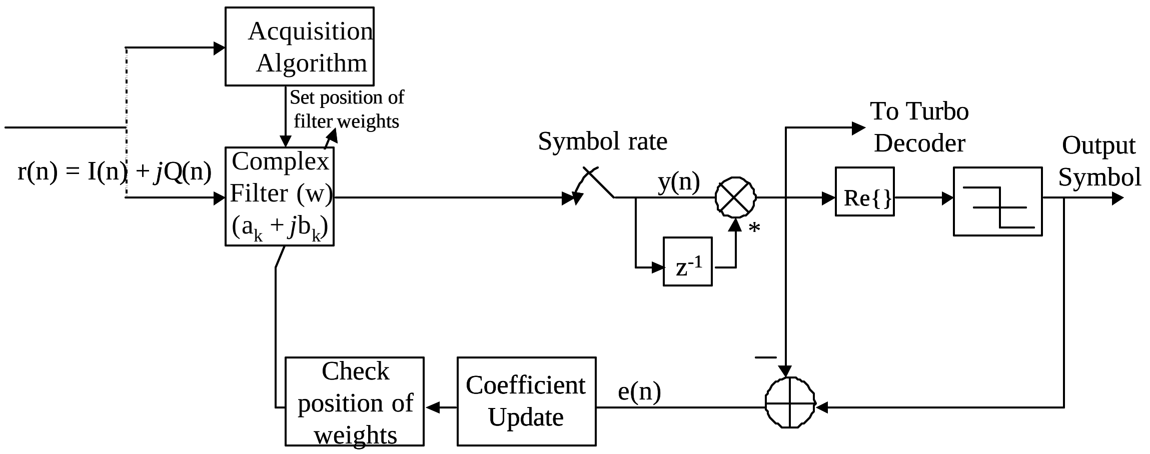


Figure 4.2 Single User Adaptive Receiver Block Diagram

The acquisition algorithm is shown below in Figure 4.3. In acquisition, the signal is passed through a filter whose taps are set with the spreading coefficients. The input slides through the matched filter for N samples and the magnitude of the correlation measured. To improve the estimate, these correlations are averaged over L symbols. This is performed by adding together the current result with the previously calculated symbol estimate. To limit the probability of a false detection, the squared magnitude that corresponds to the position of maximum correlation is compared with a preset threshold. If the threshold is exceeded, the receiver goes into tracking mode, and the index is used to determine the initial position of the weights in the filter.

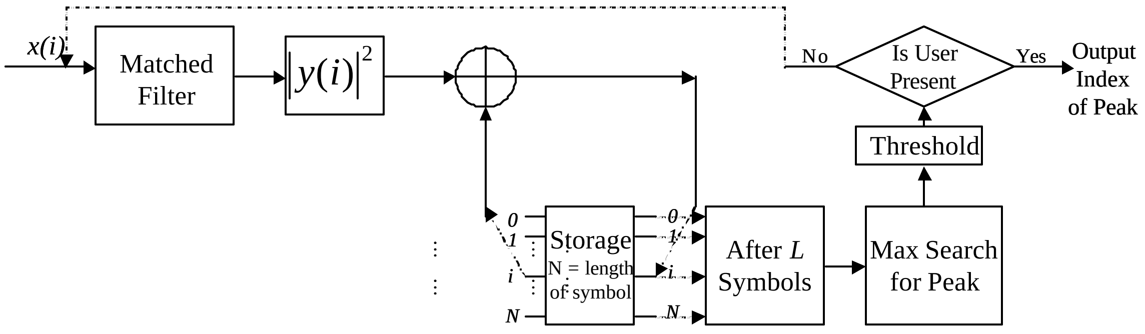


Figure 4.3 Acquisition Algorithm Block Diagram

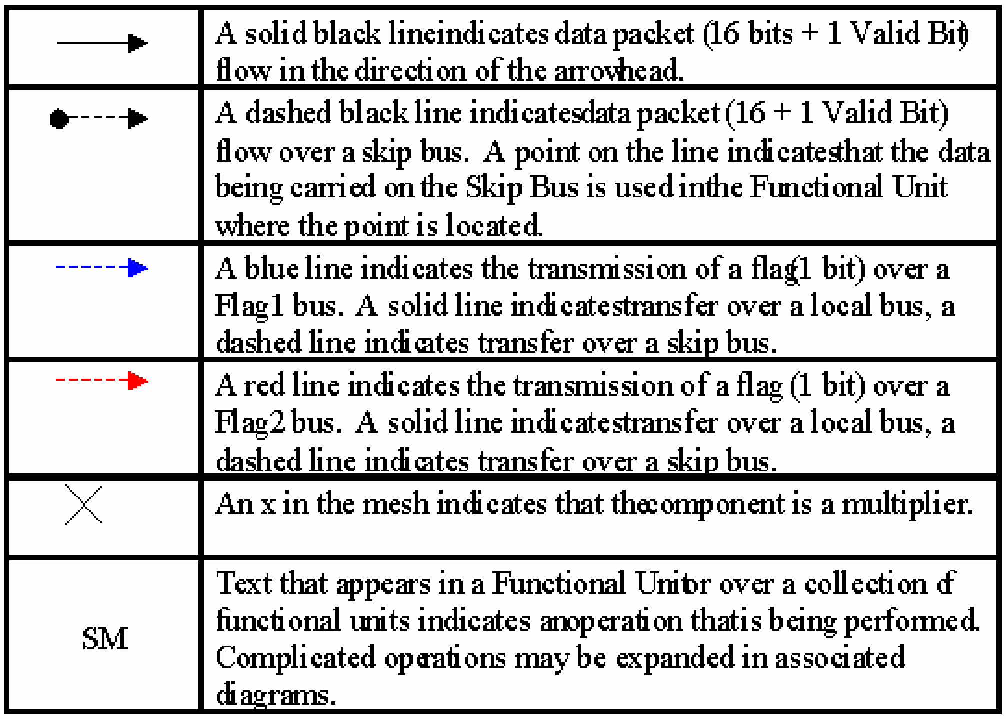
4.2 Physical Layer Mappings

To support an implementation on the Layered Radio Architecture, the single-user receiver was segmented into the following configurations:

- Filter
- Acquire and Synchronize
- Demodulate I and Q
- Calculation of Error and Weight Offsets
- Weight Updating
- Flexible μ
- Weight Position Determination

The Filter configuration is used to perform the filtering operation of the complex filter. The Acquire and Synchronize configuration is used to perform the averaging during acquisition as well as the peak search. Due to the scarcity of multipliers, the updating of weights is performed over three configurations, each of which performs part of the multiplications required for this process. Demodulate I and Q is used to perform differential decoding. The Calculation of Error and Weight Offsets configuration is used to calculate the error and to calculate the offsets to be added to the previous symbol's weights. Flexible μ provides the capability of The Weight Position Determination is used to identify the position of the weights and generate a flag for potentially shifting the weights forwards or backwards.

The mappings to the physical layer were initiated as part of the research conducted by Michael Hosemann [19]. As part of this work, the mappings were adjusted to accommodate the unsigned multiplication operation of the multipliers and to perform the necessary scaling. The following Sections detail the mappings the final mappings used to implement the physical layer of the Single User Receiver.



→	A solid black lineindicates data packet (16 bits + 1 Valid Bit) flow in the direction of the arrowhead.
●- - -→	A dashed black line indicatesdata packet (16 + 1 Valid Bit) flow over a skip bus. A point on the line indicatesthat the data being carried on the Skip Bus is used inthe Functional Unit where the point is located.
- - - - -→	A blue line indicates the transmission of a flag(1 bit) over a Flag1 bus. A solid line indicatestransfer over a local bus, a dashed line indicates transfer over a skip bus.
- - - - -→	A red line indicates the transmission of a flag (1 bit) over a Flag2 bus. A solid line indicatestransfer over a local bus, a dashed line indicates transfer over a skip bus.
╳	An x in the mesh indicates that thecomponent is a multiplier.
SM	Text that appears in a Functional Unitor over a collection of functional units indicates anoperation thatis being performed. Complicated operations may be expanded in associated diagrams.

Figure 4.4 Physical Layer Diagram Conventions

4.2.1 Filter Configuration

The Filter Configuration, shown in Figure 4.5, is used to perform the complex filtering operation. $y(n) = w^H(n) r(n)$ Because of the multiplier's lack of support of signed multiplication, significant resources are dedicated to supporting this operation. This is complicated by the separation of Stallion into two meshes which requires additional resources to be used for routing information around Stallion.

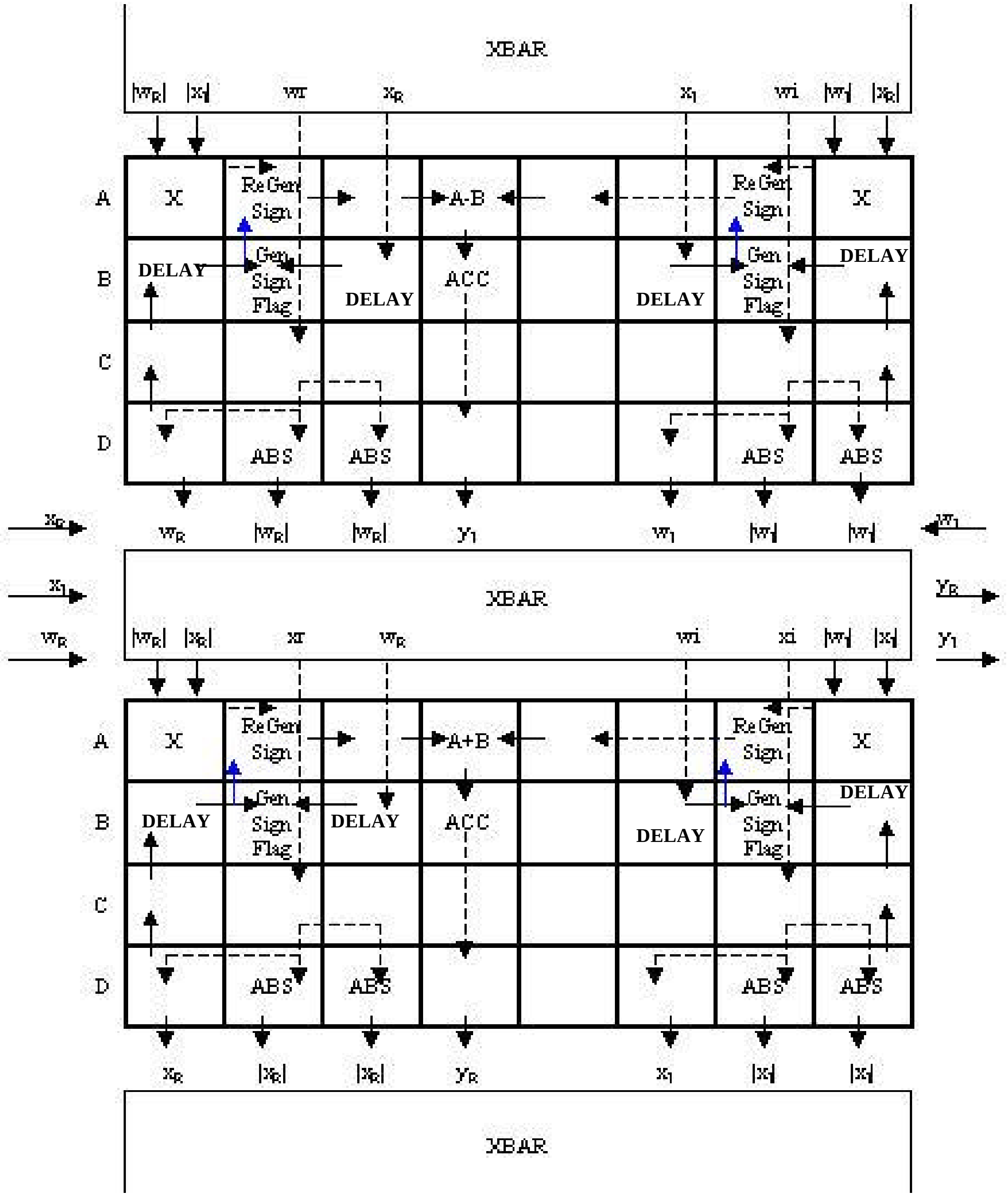


Figure 4.5 Filter Configuration Physical Layer Mapping

To perform a multiplication with sign extension, the following steps are taken:

- each of the multiplicands are replicated into four components
- the absolute values of two of these components are taken and distributed to different multipliers
- the remaining two components are distributed and delayed (for timing) to points near those same multipliers for use in sign restoration post multiplication
- each of the absolute values are then multiplied together with its associated multiplicand.
- on the cycle in which the multiplication results are delivered from the multiplier, the unaltered versions of the multiplicands are used to generate the sign bit which is used to restore the sign of the multiplier's output. This sign restoration is performed by creating a negated and an unaltered version of the result and then using the conditional mode to select the appropriate result.

The mapping of this process can be readily seen in the physical layer mapping shown in Figure 4.5. Note that propagation considerations must be considered in order to achieve the requisite timing for matching up the sign bit to the results coming out of the multiplier in this configuration.

4.2.2 Acquire and Synchronize

The Acquire and Synchronize configuration is used to implement the acquisition block of the Single User Receiver. This includes the position correlation as well as the peak search over the correlated outputs. However, in order to perform both of these operations within the same configuration, the Configuration layer is required to be able to store and route the appropriate results to the Physical layer. Note that there is no need to restore the sign post multiplication as the squaring process involved in evaluating the magnitude removes the possibility of negative values. Note that a modified version of the magnitude is used in this configuration as no square root is evaluated in order to achieve this result. This in no way biases the statistic, but it does alter the distribution.

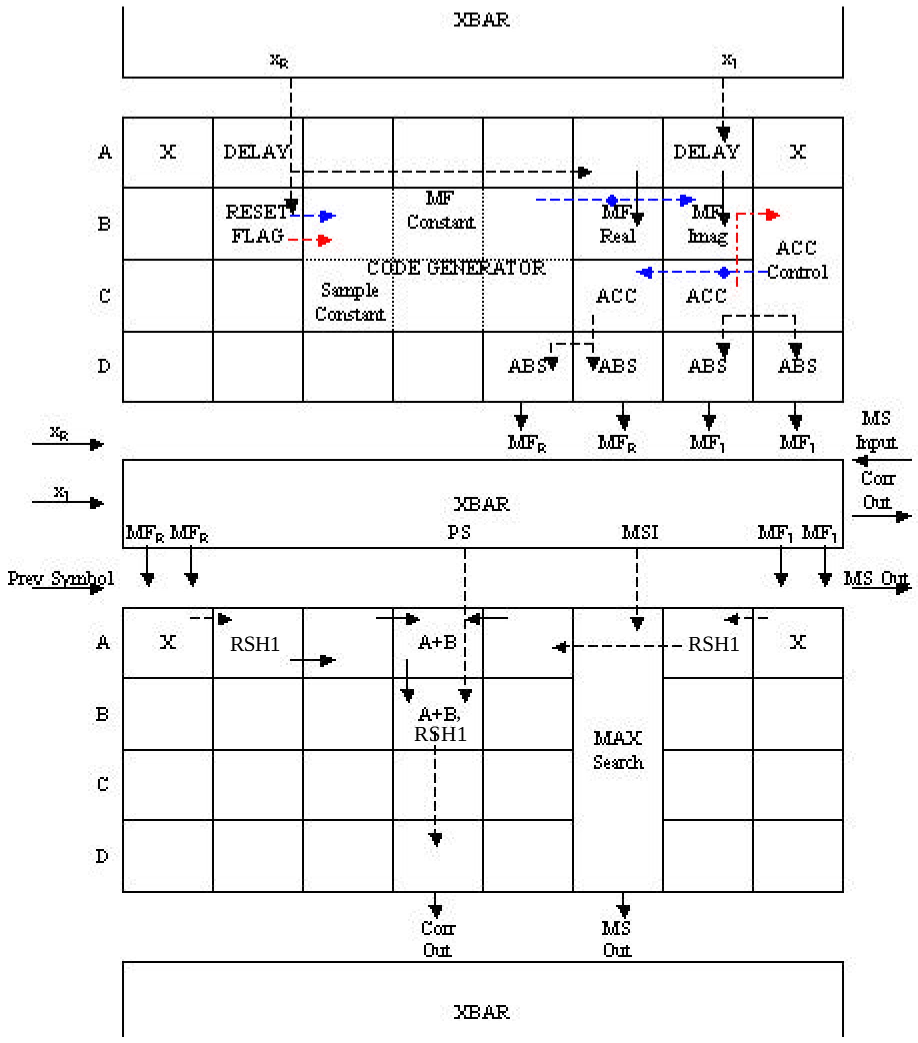


Figure 4.6 Acquire and Synchronize Physical Layer Mapping

The code generation block is shown in Figure 4.7. It takes as inputs two reset lines which are tied together and generated by the valid bit in an adjacent FU. The two leftmost FUs in the configuration are used to generate a flag to indicate when a new chip needs to be generated. This is performed with a simple resettable counter in the top left FU and a comparator in the bottom most FU. Note that due to propagation effects, the stored constant needs to be one less than the number of samples per chip. The spreading code is generated by the middle two functional units. At initialization, the constant, which holds the spreading code, is loaded. However, resource limitations dictate that the inversion of this flag be performed elsewhere and requires two additional functional units for the inversion and transmission of the flag. Both the oversampling factor and the spreading code can be treated as embedded variables by the configuration layer allowing for low level flexibility in the configuration.

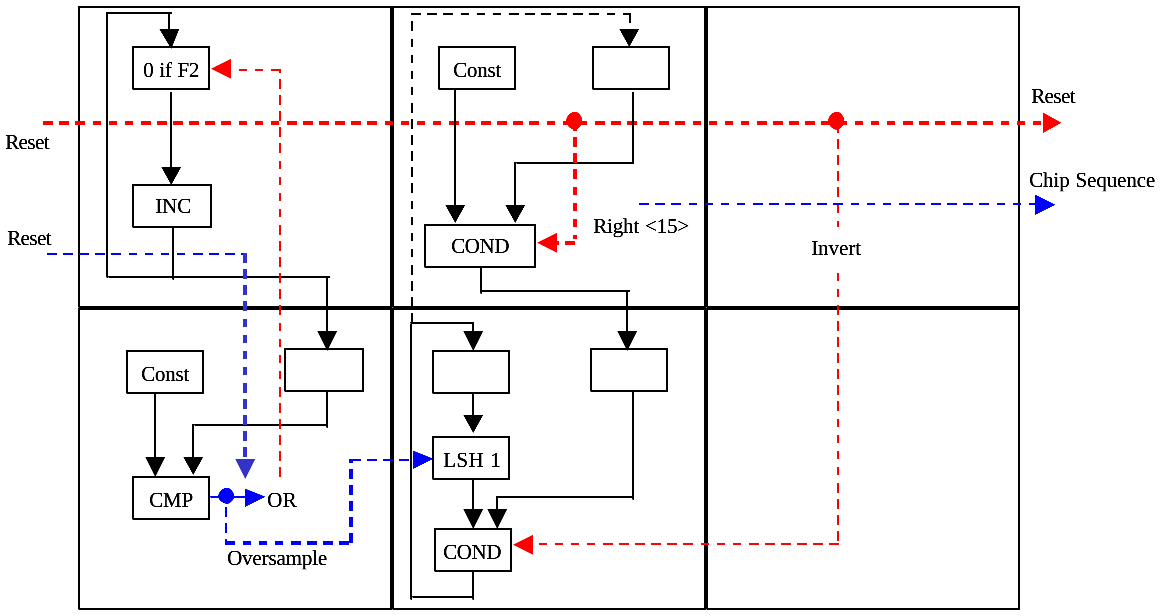


Figure 4.7 Code Generation Block

Figure 4.8 gives a detailed depiction of the peak search block used within the Acquisition configuration for identifying the sample index at which maximum correlation with the spreading code has been achieved. The Configuration layer is responsible for feeding in the sequence of correlation values (peak values) generated by the remainder of the Acquisition configuration. It is also responsible for inserting an invalid zero at the beginning of this vector for the purposes of instructing the circuit when it should reset. The top two FUs in this block are used to store the maximum correlation observed from the incoming vector and to compare incoming samples to this maximum value. When a new sample exceeds the maximum value, it is set as the maximum level and a flag is generated and transmitted to the lower two FUs. The lower two FUs serve as an index counter and a settable register. After initial reset, the index counter continuously increments and transmits this index to the lower FU. The lower FU acts as an index register that is set to the currently received index when signaled from the peak search circuit (top two FUs). With these two circuits working together, the index corresponding to maximum correlation can be identified from a vector of correlation results.

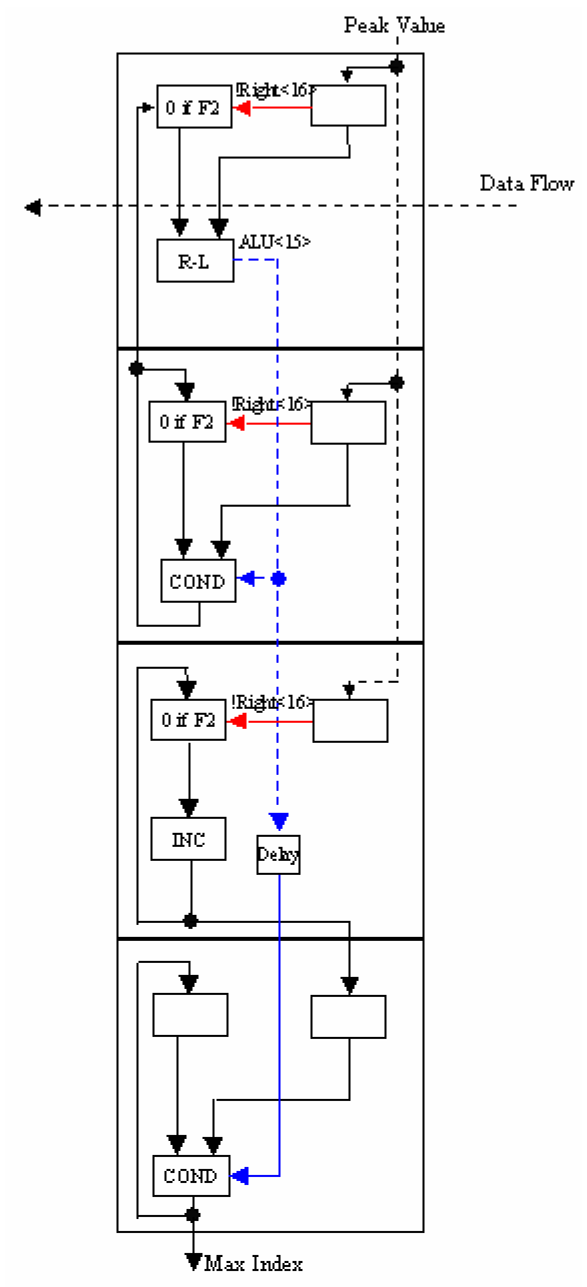


Figure 4.8 Peak Index Search Block

The accumulate control is implemented with a resettable counter in a manner similar to that of the oversample generator used in code generation block.

4.2.3 Demodulate I, Q

The Demodulate I, Q configuration performs the operation $z(n) = y[n] \times y^{*}[n-1]$. This is performed using a complex multiplication as in the Filter configuration. However, this configuration differs from the filter configuration in that it is not a vector operation, no accumulation is performed and no scaling is performed. Note that in order to reduce the number of processing cycles required to implement this configuration, $y(k)$ and $y^{*}(k-1)$ are included in this configuration using embedded variables in the locations shown in Figure 4.9.

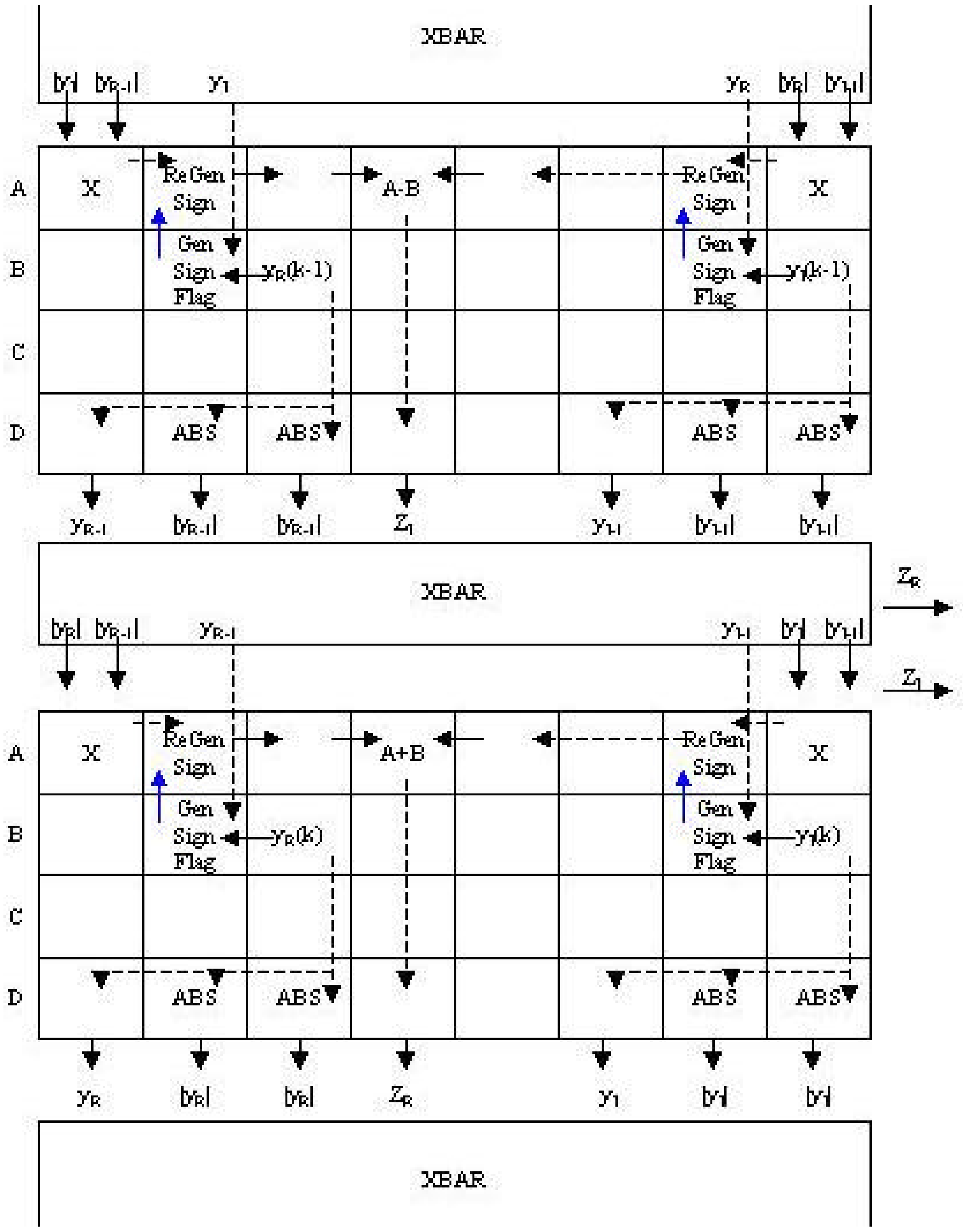


Figure 4.9 Complex Demodulation Physical Layer Mapping

4.2.4 Calculation of Error and Weight Offsets

The Calculation of Error and Weight Offsets Configuration, shown in Figure 4.10, is used to calculate the error, e_I and e_Q, and the offsets that should be used to update the weights, u_I and u_Q, error . Note that the "1" used below is not an actual 1, but rather a 8192 which is a 1 in Q13 notation. To reduce the latency in generating a result by two cycles, relevant values are stored within the configuration requiring the use of embedded variables. Again the primary operation involved in this configuration is the implementation of a sign extension multiplication.

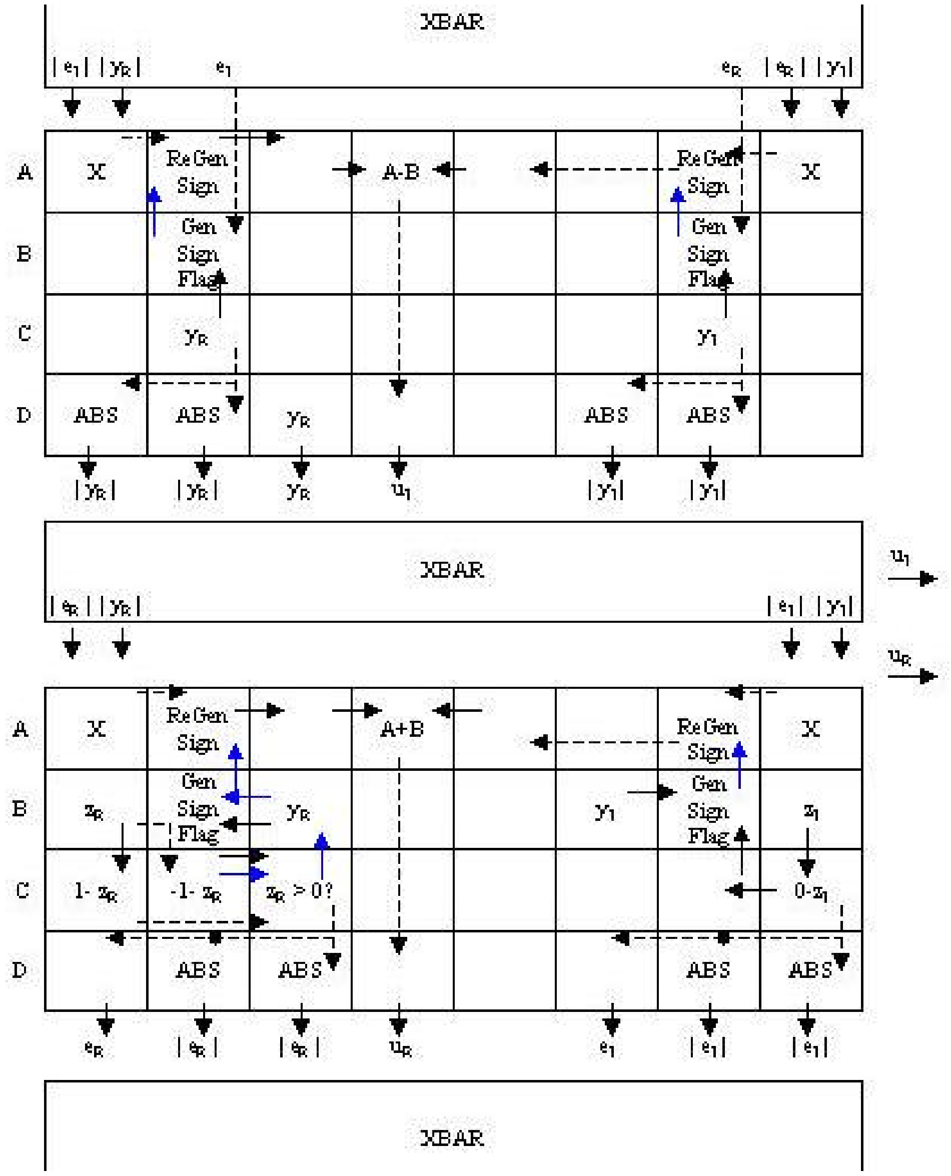


Figure 4.10 Error and Weight Offset Calculation Configuration

4.2.5 Weight Updating

The Weight Updating configuration, shown below in Figure 4.11, performs the next stage in the weight updating process. In this stage the vector of weights used for the previous symbol are multiplied by the updating factor, *u*, calculated as part of the Error and Weight Offset configuration. Note that *u* is expected to be treated as an embedded variable. Again note that the majority of the resources in this configuration are used to perform the sign extension of the multiplication.

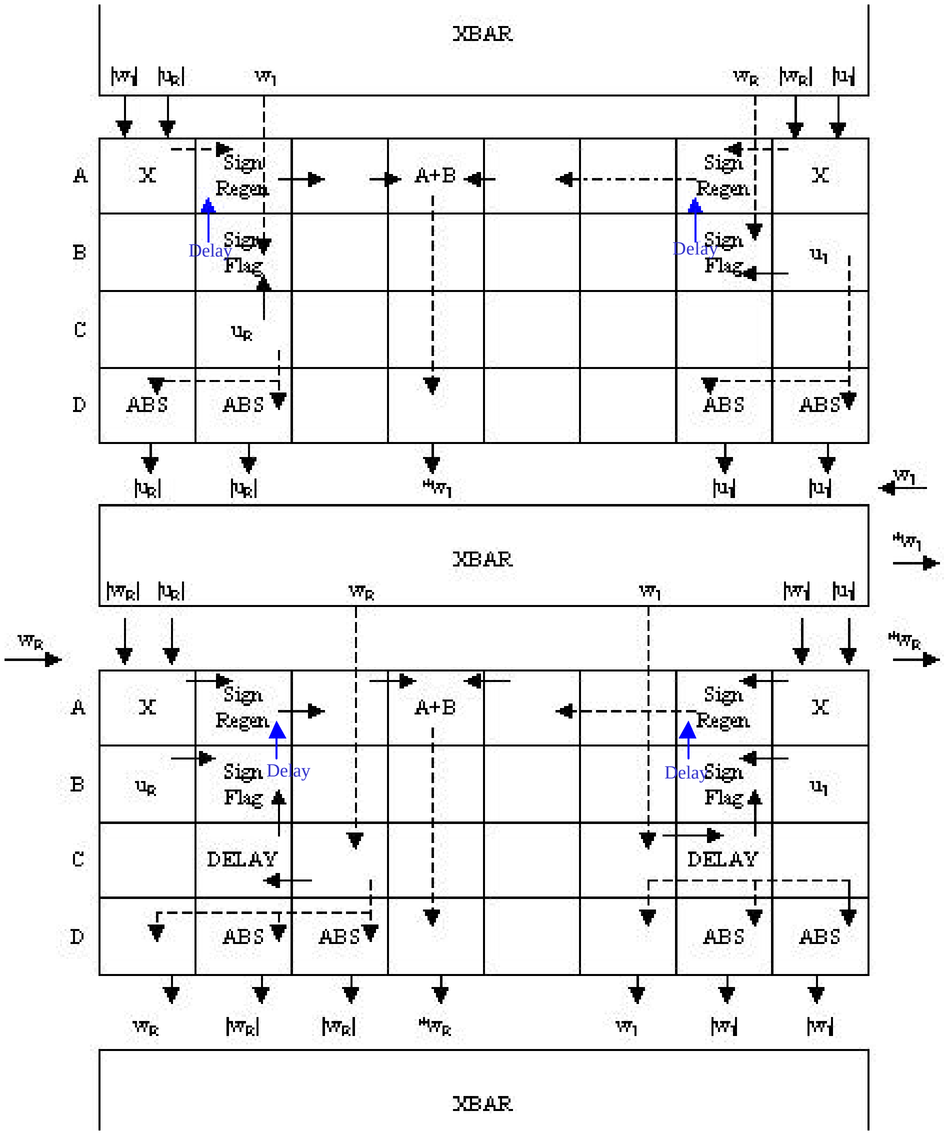


Figure 4.11 Weight Updating Physical Layer Mapping

4.2.6 Flexible m

In the Flexible μ configuration, shown in Figure 4.12, the partial products produced from the Weight Updating configuration are multiplied by the step size *m* and added to the previous symbols weights. Note that it would be possible to alter the Weight Updating configuration to perform this operation if the desired step size could be formed from powers of two (such as 0.5 or 0.75) where the operation would be performed by a series of shifts and additions.

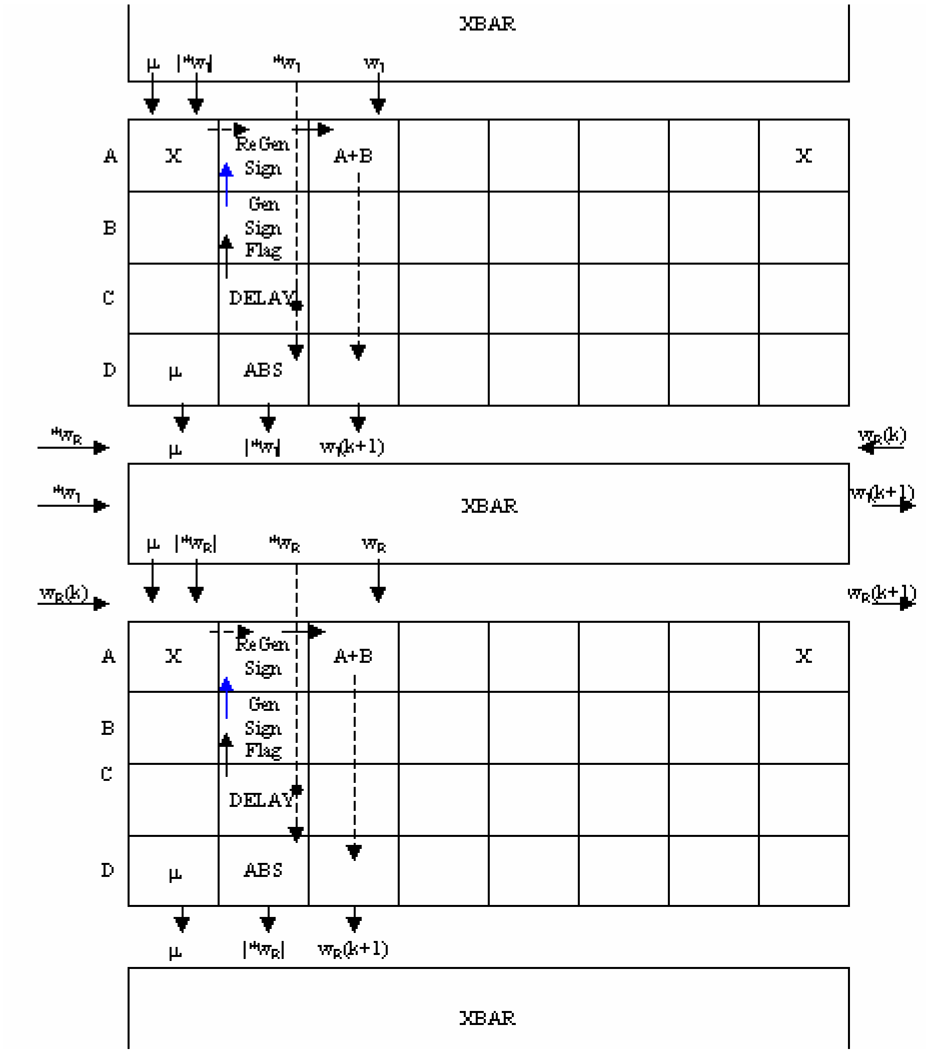


Figure 4.12 Flexible μ Configuration Mapping

4.2.7 Weight Position Determination

The weight position determination is performed using the same physical layer implementation as used for the acquisition configuration. It is expected to be called at a rate *r* which is a fraction of the symbol rate and thus may not be called every symbol. This parameter is implicitly set by the SRI layer when it sets up the Configuration Layer's state transition table. Note that it must search over 61 sample positions instead of the 60 samples in the acquisition configuration. Also note that depending on channel conditions, it may not be necessary to check the weight position every symbol. Reducing the rate at which this configuration is called would greatly reduce the total number of cycles required for the single user receiver implementation.

4.3 Configuration Layer Details

This section details how the Configuration Layer was programmed to support the implementation of the Single User Adaptive Receiver. This Section details the embedded variables, nonembedded variables, data memory, configuration, and stream header information needed to properly load the configuration onto Stallion for use in the Single User Adaptive Receiver used to set the parameters of the controller chip object and that could be used to program a configuration layer implementation. These details are broken down by controller module, configuration module and data module.

4.3.1 Memory Allocations and Controller Module State Machine

This Section describes the basic operations of the configuration layer. This includes the allocation of configuration memory and data memory and also describes the Controller Module State Machine used to determine the general operation of the Layered Radio Architecture.

For the Single User Adaptive Receiver, the Configuration Memory was allocated as shown in Table 4.1. The left column indicates the memory location and the right column gives the Configuration Mneumonic used to differentiate the Physical Layer mappings in Section 4.2. Note that the Acquisition, Peak Search, Weight Correlation and Weight Peak Search utilize the same physical layer mapping, but since each must be treated differently by the Configuration layer, a separate configuration is made for each.

Note that while this incurs a penalty in terms of wasted memory, this does not result in increased cycle usage.

Table 4.1 Configuration Memory Allocation

Configuration Number	Configuration Mneumonic
0	Acquisition
1	Peak Search
2	Filter
3	Demod I,Q
4	Error Calculation
5	Weight Update
6	Flexible μ
7	Weight Correlation
8	Weight Peak Search

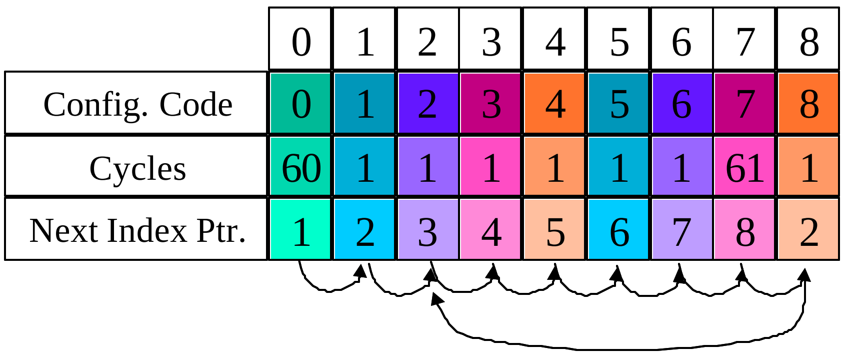
For the Single User Adaptive Receiver, I and Q memory allocations remain the same. The NEV portion of the data memory is allocated as shown in Table 4.2. Note that NEV 0 is allocated as a temporary swap space. The use of NEV 5 and NEV 6 is dependent on whether the radio is operating under acquisition or tracking. NEV 7 and NEV 8 are used to store both the current nondifferentially decoded symbol and the current differentially decoded symbol.

Table 4.2 Data Memory Allocation in Configuration Layer

NEV Number	Variable Mnemonic
0	temp
1	I Acq Input
2	Q Acq Input
3	I Weights
4	Q Weights
5	Acq Sums / I Partial Weights
6	Acq Sums / Q Partial Weights
7	$Y_I(n-1)$, $Y_I(n)$
8	$Y_Q(n-1)$, $Y_Q(n)$
9	Z_I

10	Z_Q
11	u_R
12	u_I

The Control Module State Machine is shown in Figure 4.13. The configuration code, listed in the top row refers to the configuration number in configuration memory. shown in Table 4.1. Note that Acquisition runs 60 times in order to search over each of the possible sample positions. If averaging is employed, the SRI Layer would have to specify a larger number as part of programming the Configuration Layer. Also note that during tracking, after the weight peak search is performed, operation resumes with the Filter configuration.



	0	1	2	3	4	5	6	7	8
Config. Code	0	1	2	3	4	5	6	7	8
Cycles	60	1	1	1	1	1	1	61	1
Next Index Ptr.	1	2	3	4	5	6	7	8	2

Figure 4.13 Control Module State Machine

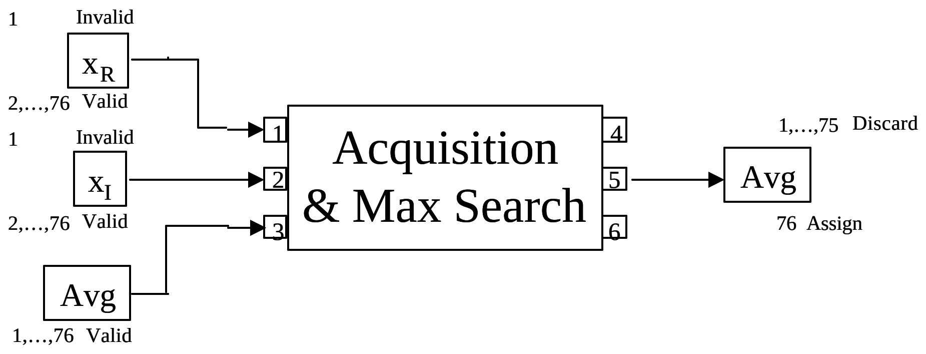
4.3.2 Configuration Information

This Section describes the information needed to load and execute each of the configurations required for this implementation through a series of diagrams. In these diagrams, the box in the center lists the configuration mnemonic and designates the physical layer mapping. On this box, there are numbered squares that represent data ports through which streams flow into and out of the physical layer implementation as indicated by directional arrows. These arrows are connected to boxes that indicate the stream's source or sink which is listed by mnemonic. For each input stream, information is listed that says which packets should have a valid bit appended to it. Note whether a packet is valid or invalid is only relevant for processing packets as all programming packets must be valid. For each output stream, information is given to list which output packets should be discarded and which should be is listed above the variable name.

Those cycles for which the output should be assigned to the output variable are listed below the output stream.

Additional information associated with the configuration is listed in textboxes at the bottom of the diagram. One text box lists the programming and processing cycles as well as if the configuration needs to be reprogrammed when used consecutively. Another textbox gives a breakdown of the embedded variables associated with each configuration with its associated FU. If an embedded variable is listed as being indirect, then its source is also listed.

Figure 4.14 shows the information stored to run the Acquisition Configuration. Note that it has provisions for updating the number of samples per chip as well as the spreading code. Also note that this configuration does not need to be reloaded when used in consecutive configurations thus improving the efficiency of the implementation. The averaging symbols are stored in the NEV used to store temporary variables.

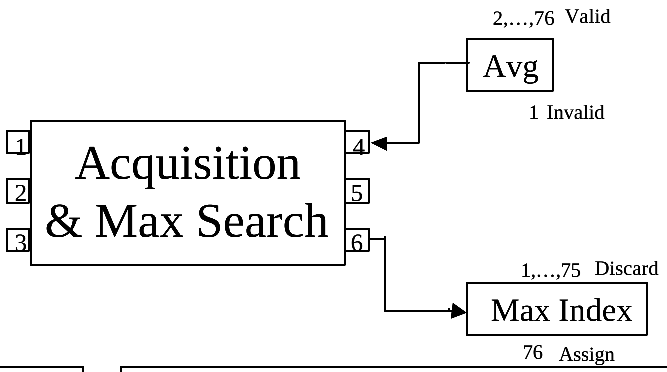


Cycles	
Programming	106
Processing	76
Reprogram	False

Embedded Variables		
Sample Constant	HC3	Direct
MF Constant	HB4	Direct

Figure 4.14 Acquisition Configuration

Figure 4.15 shows the Peak Search configuration. Note that it actually has 0 programming cycles; thus when this configuration is called, the physical layer will be identical to the previously loaded configuration. Also note that this configuration expects to operate with the same Acquisition & Max Search configuration used in the Acquisition configuration. However, different inputs and outputs are used necessitating a new configuration.



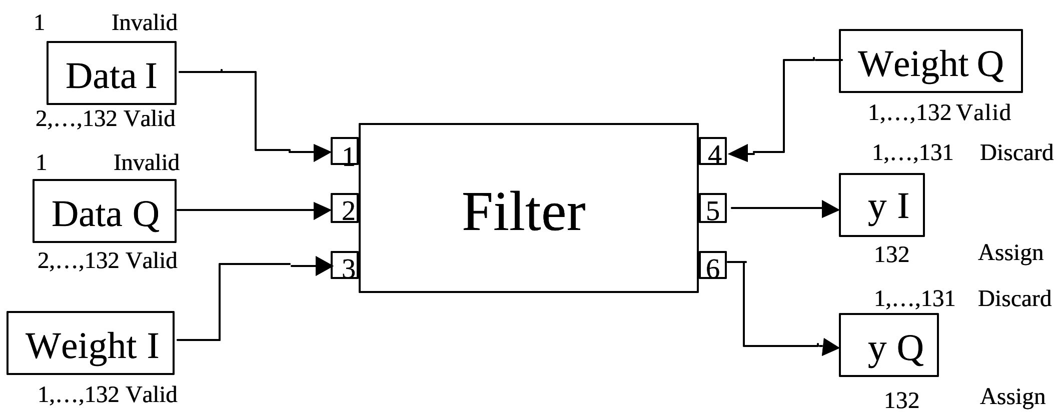
Cycles	
Programming	0
Processing	76
Reprogram	False

Embedded Variables

None

Figure 4.15 Peak Search Configuration

Figure 4.16 shows the information needed to run the Filter configuration. Note that no embedded variables are required to implement this configuration. It is also merely a coincidence that the number of cycles for programming and processing are the same for the filter configuration.



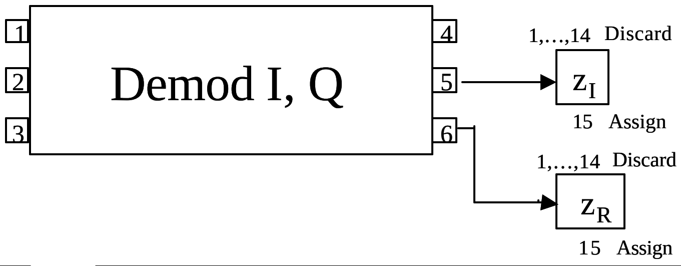
Cycles	
Programming	132
Processing	132
Reprogram	False

Embedded Variables

None

Figure 4.16 Filter Configuration

Figure 4.17 depicts the information needed to run the Demod I, Q configuration. Note that this configuration makes extensive use of indirect embedded variables for its inputs in an effort to reduce the number of processing cycles associated with this configuration. Note that this incurs no additional penalty in terms of programming cycles as the word length of a FU is fixed.

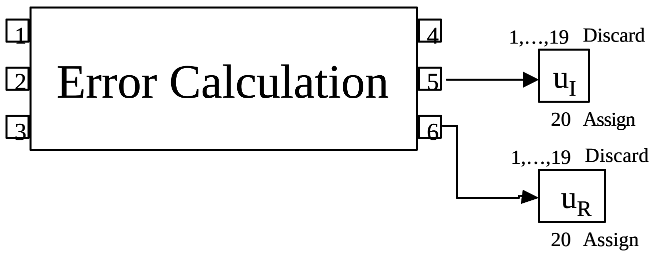


Cycles	
Programming	104
Processing	15
Reprogram	False

Embedded Variables		
$y_R(k-1)$	HB3	NEV_7[0]
$y_I(k-1)$	HB8	NEV_8[0]
$y_R(k)$	LB3	NEV_7[1]
$y_I(k)$	LB8	NEV_8[1]

Figure 4.17 Demod I, Q Configuration

Figure 4.18 shows the information needed to run the Error Calculation configuration. Again note that this configuration makes extensive use of embedded variables as inputs to reduce the number of processing cycles.

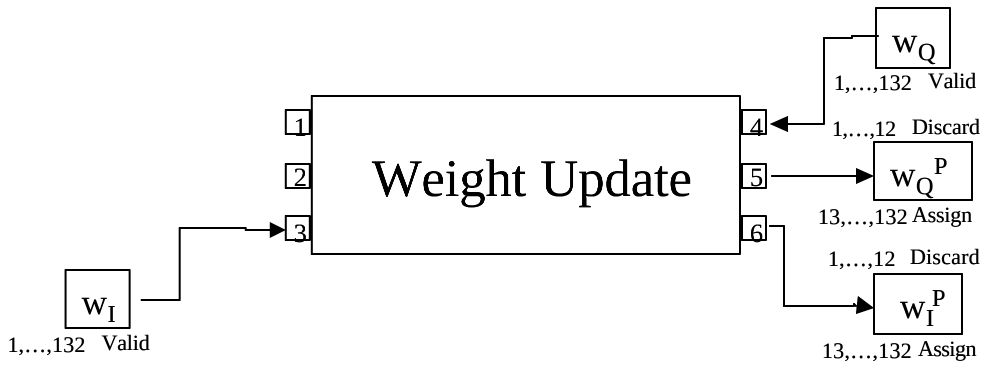


Cycles	
Programming	104
Processing	20
Reprogram	False

Embedded Variables		
$y_R(k)$	HC2	NEV_7[1]
$y_I(k)$	HC7	NEV_8[1]
$z_R(k)$	LB1	NEV_9[0]
$y_R(k)$	LB3	NEV_7[1]
$y_I(k)$	LB6	NEV_8[1]
$z_I(k)$	LB8	NEV_10[0]

Figure 4.18 Error Calculation Configuration

Figure 4.19 shows the information needed to load and run the Weight Update Configuration. This configuration makes use of streams to handle vectors of data while using embedded variables to handle scalar values.

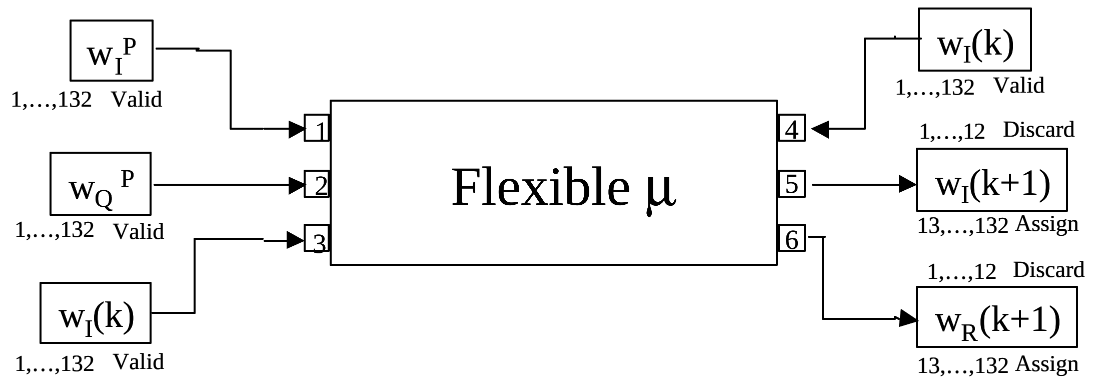


Cycles	
Programming	132
Processing	132
Reprogram	False

Embedded Variables		
u_R	HC2	NEV_11[0]
u_I	HB8	NEV_12[0]
u_R	LB1	NEV_11[0]
u_I	LB8	NEV_12[0]

Figure 4.19 Weight Update Configuration

Figure 4.20 shows the information needed to run the Flexible μ Configuration. Note that one embedded variable points to another within the same configuration. This reduces the amount of information the SRI Layer needs in order to adjust the step size.

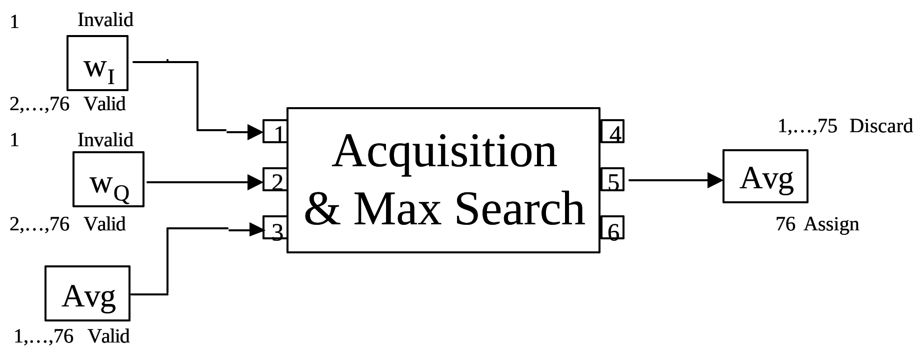


Cycles	
Programming	85
Processing	132
Reprogram	False

Embedded Variables		
μ	HD1	Direct
μ	LD1	EV0

Figure 4.20 Flexible μ Configuration

Figure 4.21 shows the information needed to load and run the Weight Correlation Configuration. Note that the physical layer is identical to the Acquisition configuration although the inputs are different.



Cycles	
Programming	106
Processing	76
Reprogram	False

Embedded Variables		
Sample Constant	HC3	C0 EV0
MF Constant	HB4	C0 EV1

Figure 4.21 Weight Correlation Configuration

Figure 4.22 shows the information needed to run the Weight Peak Search. However, this configuration requires 77 processing cycles instead of 76 since it must search over 61 positions instead of 60. This represents a relative weakness of the Configuration Layer where a new configuration is required in order to make relatively small changes in a configuration. However this sort of tradeoff was necessitated by its implementation on a FPGA which is relatively inflexible.

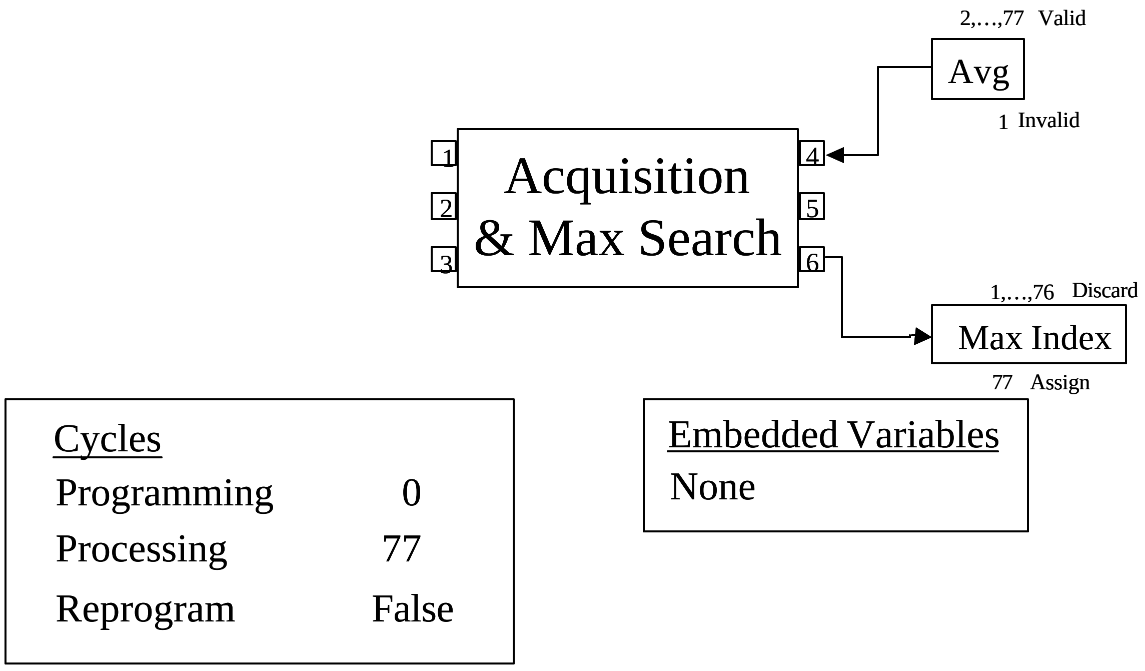


Figure 4.22 Weight Peak Search

4.4 *Implementation Results*

This Section describes the results of this implementation of the Single User Receiver. For this implementation, the primary objectives were the verification that the Layered Radio could be used to implement a flexible receiver structure and to measure the receiver’s utilization and the gathering of performance statistics in order to give an indication of the potential benefits of the Stallion processor within the Layered Radio Architecture framework.

4.4.1 Implementation Validation

In this Section, the fidelity of the implementation is examined in order to demonstrate the accuracy of the implementation. To verify the operation of the Acquisition algorithm as mapped to the Layered Radio Architecture, a sequence of five symbols with an offset of one sample were simulated and input to the combined configuration layer and physical layer simulation described in Chapter 3. The correlation values produced from this simulation are shown in Figure 4.23. Note that the correlations must always be positive due to the squaring operation within the Acquisition configuration. Also note that this same process was performed for additional timing offsets, though not shown in this thesis.

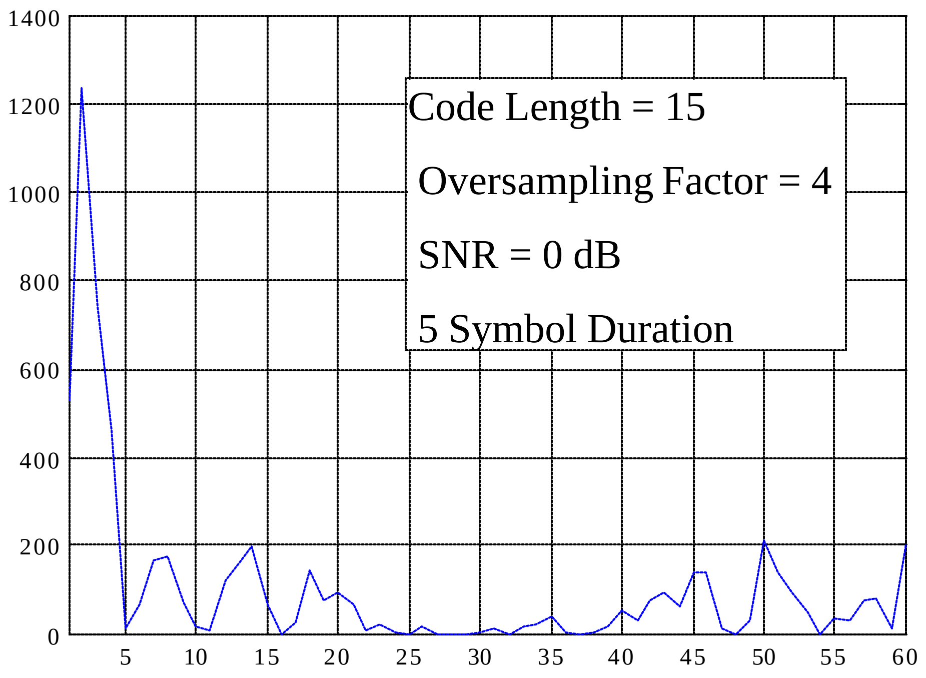


Figure 4.23 Acquisition Statistics

Next to verify the operation of the tracking portion of the receiver, a signal was simulated with multipath and a rotating constellation. A fixed point simulation was developed and implemented in C++. The simulated signal was then fed into this simulation to serve as a baseline for comparing the operation of the receiver as implemented on the Layered Radio Architecture. The average MSE for both of these implementations were collected over 1200 symbols and are shown in Figure 4.24. Note that the MSE of the fixed point simulation is very close to the MSE of the Layered Radio Implementation. The slight differences in value result from different field widths being used in the fixed point simulation and the Layered Radio implementation. However, the close similarities in the shape of the MSE plots serve to show the ability of the Layered Radio Architecture to support this implementation.

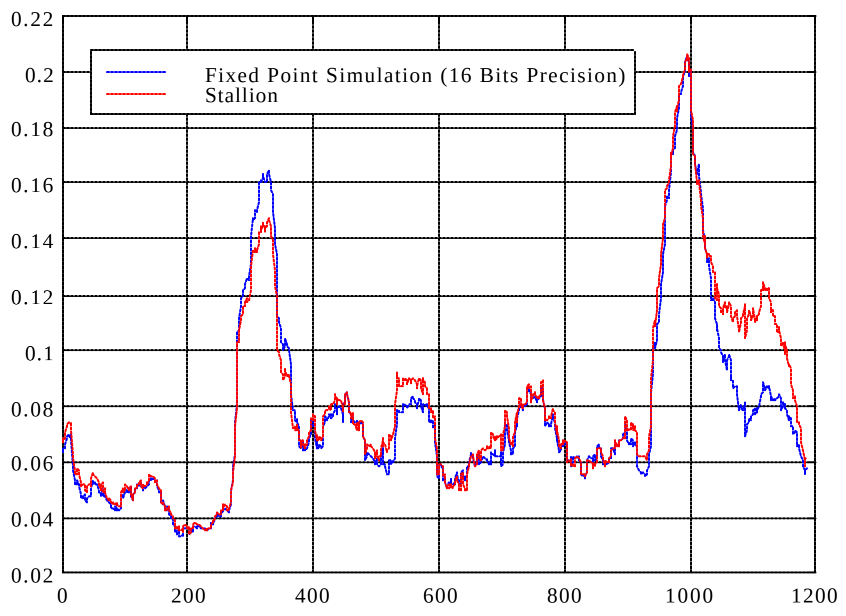


Figure 4.24 Average MSE During Tracking As Compared to a Fixed Point Implementation in C++

As noted in Sections 4.2 and 4.3, this implementation permits the alteration of entire configurations as well as altering configuration specific parameters such as the spreading code. Thus in this instance, the fidelity and flexibility of the Layered Radio Architecture approach is demonstrated.

4.4.2 Utilization Statistics

This Section summarizes the relevant utilization statistics for this receiver implementation as broken down by configuration and across the entire implementation. These statistics are intended to give an indication of the performance of the Stallion CCM when coupled with the hardware paging and stream based processing approaches of the Layered Radio Architecture.

For this implementation the following statistics were gathered / generated:

- Functional Units – the number of functional units that a configuration uses. In general the higher the number of functional units a configuration can use, the more effectively Stallion is being used, thus giving an indication of the processing power that Stallion is using in each cycle. Note that totals are weighted by the number of percentage of total processing cycles used by each configuration.

- Multipliers – same as for Functional Units, but instead measuring the number of multipliers
- Gross Component Utilization – a raw count of the percentage of total processing resources (functional units and multipliers) that is being used in a configuration. In general, an increase in this number indicates an increase in the level of parallelism that Stallion achieves in the configuration.
- Adjusted Component Utilization – a modified utilization statistic that considers how many components would be required if each component could perform the operations that would typically be done within a single component in other processors. This allows for a fairer comparison between devices and gives a better indication of the efficiency of an implementation. Thus adjusted device utilization discounts those functional units that are not actually performing a calculation and are only used in routing data as well as those functional units involved in operations that in future designs should be performed within a single device. For instance, for a sequence of more than one right shift, only the functional unit performing the first right shift is counted as it is expected that any future designs will support right shifts by factors greater than 1. Also any functional units that are only used for helping to restore the sign after multiplication are also not included in this number.
- Programming Cycles – the number of cycles that must be consumed before processing can proceed. Note that if hardware paging were not being used, then this would need to be broken down by stream in order to give a fairer assessment.
- Processing Cycles – the total number of cycles required to calculate all of the results of the configuration. This includes any leading invalid numbers that must be input in order to set or reset a configuration.
- Processing Efficiency – the ratio of processing cycles to the total number of cycles used in a configuration. For stream based processors, cycles used to program the device represent "lost" cycles for which results are not generated and the device is not being used as efficiently. Processing Efficiency therefore gives

an indication as to how effectively the stream based concept is being applied with higher numbers being more desirable.

- Number of Streams Used – this measure the total number of data ports, whether for input or output, that the configuration is using.
- I/O Efficiency – the ratio of used data ports to total available data ports. This measures how well the device is making use of the macroscopic parallelism of the Stallion chip. If a configuration consumes all available resources without using all the available data ports, the I/O efficiency would also be considered to be one as the maximum rate of processable data would already be achieved.
- Gross Utilization – the product of Gross Component Utilization, Processing Efficiency, and I/O Efficiency. This gives an overall picture of how well a configuration uses Stallion's resources.
- Adjusted Utilization - the product of Adjusted Component Utilization, Processing Efficiency, and I/O Efficiency. This gives an overall picture of how well a configuration uses Stallion's resources in a way that can be compared to other processors.

Using these metrics, statistics were then gathered for the acquisition and tracking operations of this receiver.

Table 4.3 summarizes the statistics for the acquisition portion of the Single User Adaptive Receiver. Note that n denotes the number of symbols averaged over when finding the peak. Also note that Acquisition and Peak Search are actually implemented as part of the same physical implementation on Stallion. The separation in this table is indicative of how the Configuration Layer treats these configurations and also gives a fairer depiction of the performance of these configurations as the configuration layer uses the physical layer in distinctly different manners between these two configurations.

Table 4.3 Acquisition Statistics for Single User Statistics

	Acquisition	Peak Search	Total
Functional Units	35	35	35
Multipliers	2	2	2
Gross Resource Utilization	57.81%	57.81%	57.81%
Adjusted Resource Utilization	25%	6.25%	24.94%
Programming Cycles	106	0	106
Processing Cycles	4440*n*	67	67 + 4440*n*
Processing Efficiency	99.52%	100%	99.53%
Number of Streams	4	2	3.99
I/O Efficiency	66.67%	33%	66.56%
Gross Utilization	38.31%	19.18%	38.25%
Adjusted Utilization	16.59%	2.06%	16.55%

Table 4.4 summarizes the performance statistics measured during the Tracking Operations. Note that *n* denotes the number of symbols used for averaging during acquisition and *r* is the frequency at which Weight Position Search configuration is used. Notice that most of these configurations achieve particularly low utilization primarily due to their poor processing efficiency and poor adjusted resource utilization. Note that

achieving better resource utilization would be difficult to achieve since most configurations already have high stream utilization and high gross resource utilization.

Table 4.4 Tracking Statistics for Single User Receiver

	Filter	Complex Demod	Error Calc.	Weight Update	Flexible μ	Weight Pos. Det.	Total (assumes r = 0.5)
Functional Units	52	43	45	44	18	35	36.69
Multipliers	4	4	4	4	2	2	2.47
Gross Resource Utilization	93.75%	73.44%	76.56%	75%	31.25%	57.81%	61.68%
Adjusted Resource Utilization	12.5%	9.38%	14.06%	9.38%	6.25%	24.94%	20.69%
Programming Cycles	132	104	120	132	85	106r	573 + 106r
Processing Cycles	132	15	20	132	132	4581r	431 + 4581 r
Processing Efficiency	50%	12.61%	14.29%	50%	60.83%	99.53%	81.30%
Number of Streams	6	2	2	4	4	3.99	3.92
I / O Efficiency	100%	33.33%	33.33%	66.67%	66.67%	66.56%	65.41%
Gross Utilization	46.88%	3.09%	3.65%	25.00%	12.67%	38.30%	32.80%
Adjusted Utilization	6.25%	0.39%	0.67%	3.13%	2.53%	16.52%	11.00%

4.5 DSP Implementation Comparison

For a fair evaluation of the performance of the CCM, these numbers should be compared to what would be expected of a typical DSP implementation. Let us consider implementing these same algorithms on TI's TMS320C6201 processor, a near top-of-the-line fixed-point VLIW (Very Long Instruction Word) DSP. The C6201 has two multipliers (.M units) and six ALUs two of which are also used for loading variables (.D units), and two of which are also used for shifting operations (.S units) and two of which may also perform logical operations (.L units). The native bus width of these units is 32 bits, but the C6201 also supports 16-bit operations and like Stallion will perform a pipelineable 16x16 multiplication in two clock cycles. The (.D) units have an initial 5 cycle latency for loading data, but this process can be pipelined. Each .D unit can be used to simultaneously load 2 sixteen bit words if the words are aligned next to each other in data memory. Further the .D can perform address increments while loading the data, and thus will be able to continuously load blocks of data two elements at a time without halting and without the assistance of other units. This also assumes that the data and the coefficients are stored in separate memory blocks so as to not have a memory access conflict that would slow down this process.

As the NEV implementations used in this implementation naturally align these words in such a manner, it is expected that the same will be doable for the DSP implementation. The .L and .S units perform simple arithmetic operations in a clock cycle. Also the .S unit is used to branch to program subroutines, a process that consumes 5 cycles. There are also 32 local registers that can be used to store intermediate results without having to use the general data memory which is more than sufficient for the intermediate results associated with the Layered Radio Architecture [20]. Using these operating characteristics, the same configurations that were used to implement the Layered Radio Architecture were mapped to implementations onto a C6201 DSP. Note that neither the statistics gathered in Section 4.4 nor the statistics that follow consider the overhead associated with setting up each configuration for operation.

4.5.1 Filter Configuration

The filtering operation is a commonly implemented DSP algorithm for which the C6201 was optimized and thus can be performed very efficiently on the C6201. Assuming datapacking and complete loop unrolling, all of the operations associated with two MAC operations could be performed in a single cycle after the pipeline became filled by using the two .D units to load two coefficients and two pieces of data for each .M for a multiplication and each .L for the accumulate operation.

However, before these operations can take place, an initial prolog must be executed to fill up the pipeline. This prolog consists of an initial load requiring five cycles and an initial multiplication requiring two cycles. This gives a total prolog length of seven cycles. In parallel to these operations, the accumulators can be initialized to 0. Additionally, at the end of this operation, an epilog must be used to clean out the pipeline. The epilog consists of one cycle for a final multiplication and one cycle for a final addition. Intermediate results can be pushed into other local registers after completion. Then the total number of cycles required to perform this operation should be equal to

Prolog = 5 (initial load) + 2 (initial multiply) = 7

Kernel = 1 (Load, Multiply & Accumulate) * (120 -2) = 118

Epilog = 1 (Final Multiply, Penultimate Accumulate) + 1 (Final Accumulate) = 2

Total = 7 (Prolog) + 118 (Kernel) + 2 (Epilog) = 127 cycles

These two MAC results would then be added together to form the real component of y for one more cycle for 128 total cycles.

To create the imaginary component of y, this same operation would have to be performed again, but with a subtraction at the end instead of an addition. These two results would then have to be stored. Thus the total number of cycles required for the C6201 to perform the filtering configuration can be estimated as

DSP Filter Cycles = 2x128(Actual Operation) + 5 (Store Results) = 261 cycles.

4.5.2 Acquisition Configuration

The actual correlation with the spreading code could be implemented in a manner similar to that described for the filtering operation. However, the final addition (or subtraction) does not need to be performed. Additionally, this only needs to be performed once, and the intermediate result need not be stored. Plus, this only needs to be performed for 60 coefficients instead of 120. Thus to create the equivalent of the MF_I and MF_Q variables 67 cycles are required.

Then to perform the squaring and adding operation to find the magnitude, three cycles are required – two for the multiplication, and one for the addition. Note that no scaling is needed as the C6201 supports both 16-bit words and 32-bit words and the peak search can be performed over the 32-bit words. Thus this operation can be skipped. At the same time that this is started, the previous symbol's magnitude could be loaded; thus only two extra cycles would be required to wait for the load. Adding together these two values would take another cycle. The subsequent storing of this result would take five cycles. Thus the complete acquisition correlation and averaging for one stage would occur in

Total cycles for one sample position = 67(matched filter of length 60) +
2 (multiply)+ 1 (add) + 2 (end of load) + 1 (add) + 5 (store) = 78 cycles.

This would then have to be repeated 60 times for a symbol. Thus the total cycles to determine the position for one symbol would be given by 78 x 60 = 4680 cycles. For averaging over *n* symbols, the total number of cycles would be equal to 4680*n*.

4.5.3 Peak Search

As this operation is more complicated than the simple filtering operations or multiplications that is characteristic of the other configurations, the assembly code that would perform this operation is listed below. To speed up operation, we will assume that start addresses of the copies of the array to be searched over are stored at the addresses pointed to by B4 and A4. By duplicating the arrays, the number of cycles required to perform this operation can be cut approximately in half.

```
; PROLOG (6 Cycles)
        SUB     .S1 A5, A5, A5   ; zero the index register
||      SUB     .S2 B5, B5, B5    ; zero the max value register
||      LDH     .D2 *B4++, B6    ; begin loading first value (1)
        LDH     .D2 *B4++, B6    ; begin loading second value (1)
||      LDH     .D1 *A4++, A6    ; begin loading first value (2)
```

```
        LDH     .D2 *B4++, B6   ; begin loading third value (1)
||      LDH     .D1 *A4++, A6   ; begin loading second value (2)
        LDH     .D2 *B4++, B6   ; begin loading fourth value (1)
||      LDH     .D1 *A4++, A6   ; begin loading third value (2)
        LDH     .D2 *B4++, B6   ; begin loading fifth value (1)
||      LDH     .D1 *A4++, A6   ; begin loading fourth value (2)
;       Values start arriving (1)
        LDH     .D2 *B4++, B6   ; begin loading sixth value
||      LDH     .D1 *A4++, A6   ; begin loading fifth value (2)
||      CMPGT   .L2  B5,B6,B0   ; compare first value to 0
;       Values start arriving (2)
;KERNEL (53 Cycles)
        LDH     .D2 *B4++, B6   ; begin loading seventh value
||      LDH     .D1 *A4++, A6   ; begin loading sixth value (2)
||      CMPGT   .L2  B6,B5, B0  ; compare second value to 0
||[B0]  ADD     .S2  A6,0, B5   ; if greater (on previous) store value
||[B0]  ADD     .S1x  A5,0, B6  ; if greater (on previous) store index
||      ADD     1, A5, A5       ; increment index for next
; This Instruction Occurs 60-7 = 53 times
;EPILOG (6 cycles)
;       Values 56 (1), 55 (2) arrive
        LDH     .D1 *A4++, A6   ; begin loading sixtieth value (2)
||      CMPGT   .L2  B6,B5, B0  ; compare second value to 0
||[B0]  ADD     .S2  A6,0, B5   ; if greater (on previous) store value
||[B0]  ADD     .S1x  A5,0, B6  ; if greater (on previous) store index
||      ADD     1, A5, A5       ; increment index for next
;       Values 57 (1), 56 (2) arrive
        CMPGT   .L2  B6,B5, B0  ; compare second value to 0
||[B0]  ADD     .S2  A6,0, B5   ; if greater (on previous) store value
||[B0]  ADD     .S1x  A5,0, B6  ; if greater (on previous) store index
||      ADD     1, A5, A5       ; increment index for next
;       Values 58 (1), 57 (2) arrive
        CMPGT   .L2  B6,B5, B0  ; compare second value to 0
||[B0]  ADD     .S2  A6,0, B5   ; if greater (on previous) store value
||[B0]  ADD     .S1x  A5,0, B6  ; if greater (on previous) store index
||      ADD     1, A5, A5       ; increment index for next
;       Values 59 (1), 58 (2) arrive
        CMPGT   .L2  B6,B5, B0  ; compare second value to 0
||[B0]  ADD     .S2  A6,0, B5   ; if greater (on previous) store value
||[B0]  ADD     .S1x  A5,0, B6  ; if greater (on previous) store index
||      ADD     1, A5, A5       ; increment index for next
;       Values 60 (1), 59 (2) arrive
        CMPGT   .L2  B6,B5, B0  ; compare second value to 0
||[B0]  ADD     .S2  A6,0, B5   ; if greater (on previous) store value
||[B0]  ADD     .S1x  A5,0, B6  ; if greater (on previous) store index
||      ADD     1, A5, A5       ; increment index for next
```

```
;        60 (2) arrives
 [B0]   ADD      .S2  A6,0, B5        ; if greater (on previous) store value
||[B0]  ADD      .S1x  A5,0, B6   ; if greater (on previous) store index
||      ADD      1, A5, A5          ; increment index for next
```

Thus it can be seen that a total of 66 cycles are required to perform this operation. After which, 5 cycles will be required to store the result; thus a total of 71 cycles are needed to perform this operation.

4.5.4 Demodulate I,Q

The complex differential demodulation operation requires the loading of four sixteen bit variables which can be performed in five cycles using the two .D units. Four multiplications, an addition, and a subtraction are then required. This can again be pipelined so it is performed in four cycles. These results would then have to be stored which would require another five cycles. Thus when implemented on the C6201, a total of fourteen cycles is required to perform this operation.

4.5.5 Error Calculation and Weight Offsets

Note that this configuration performs the same operations as in the Demodulate I, Q, configuration, except for the calculation of the error which could be performed in a single cycle by using three of the ALUs. Thus it is expected that this implementation would require a total of fifteen cycles.

4.5.6 Partial Weight Update

The Partial Weight Update is effectively the same operation as the filtering operation, but with the subtraction replaced by an addition. It also only requires two loads at a time and is expected that u_I and u_Q would be loaded in parallel in each prolog. Thus this configuration is expected to require 261 cycles to perform this operation.

4.5.7 Flexible m

The Flexible μ configuration requires the repeated loading of four variables, two multiplications, and two additions. This can be implemented in a manner similar to the filter operation with the following alterations:

1. Instead of multiplying together the two vectors, a constant μ is multiplied by the incoming vectors w_I^* and w_Q^*
2. Instead of accumulating the results of this multiplication, the results are added to w_I^* and w_Q^*.

Thus this basic operation should be performable in 127 cycles. During the prolog, it is expected that the constant μ will be generated by one of the idle .S units.

4.5.8 Weight Position Search

The weight position search could be implemented using the same code as used for acquisition code. However, this needs to be performed over 61 positions instead of 60 positions, thus this will require 78 cycles for a single position thus a total of 78 x 61 = 4758 cycles is required.

To perform the search operation, the peak search code can be reused. However, an extra cycle would be needed in the kernel to allow a search of 61 positions instead of 60. Thus a total of 72 cycles would be required. Thus a total of 4830 cycles are required to perform this operation.

4.5.9 C6201 Performance Summary

Table 4.5 and Table 4.6 summarize the cycles required to implement the acquisition and tracking algorithms on the C6201 by configuration. Note that no cycles are required for programming any configuration as stream based programming is not employed on the C6201.

Table 4.5 C6201 Acquisition Statistics

Configuration	Total Cycles
Acquisition Correlation	4680*n*
Peak Search	71
Total	$4680 \times n + 71$

Table 4.6 C6201 Tracking Statistics

Configuration	Total Cycles
Filter	261
Demodulate I, Q	14
Error Calculation and Partial Weight Update	15
Partial Weight Update	261
Flexible μ	127
Weight Position Search	4830*r*
Total	678+4830*r*

4.6 Single User Adaptive Receiver Summary

The implementation of the Single User Adaptive Receiver demonstrated the feasibility of using the Layered Radio Architecture for soft radio implementations with support for coarse and fine reconfiguration. However, the utilization statistics for the implementation do not fully capture the potential of the Layered Radio architecture in this instance. As shown in Figure 4.25, the Layered Radio Architecture barely outperforms the C6201 implementation during acquisition and actually fails to perform as well as the C6201 implementation during tracking. These results are actually even worse in light of practical considerations where the C6201 can be clocked at 200 MHz, four times the rate of Stallion. Certainly Stallion uses conservative technology, but the complex timing requirements of Stallion nearly guarantee that Stallion will not be able to operate at the same clock rate as the C6201.

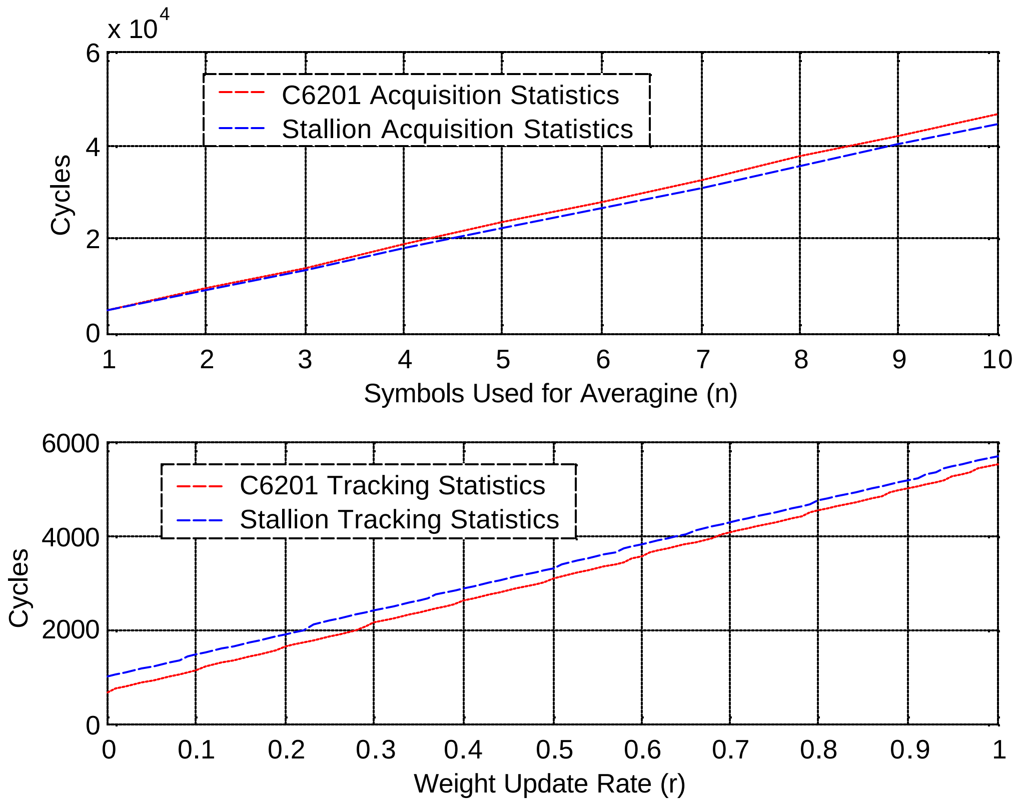


Figure 4.25 Utilization Statistics Comparison

At first these results seem counterintuitive as Stallion has better than eight times the processing resources of the C6201 and everything else being equal, should require around one eighth the cycles of the C6201 implementation. However, notice from Table 4.4 that the majority of the configurations have an adjusted utilization of less than one indicating that these configurations are not making maximum utilization of its resources, effectively reducing the benefit of the additional processing resources.

To summarize, the implementation of the Single User Receiver demonstrates that the Layered Radio architecture can be used to implement a complex receiver structure. However, the expected benefits of the Stallion CCM were not realized primarily due to low resource utilization and low processing efficiency. Similar to other processors, it is believed that Stallion exhibits a strong correlation between resource utilization and chip I/O. Although the gross resource utilization was too high for most configurations to benefit from increased I/O, the implementation examined in Chapter 5 demonstrates how an increase in I/O can greatly enhance performance.

As multiplication was the most commonly performed operation in this configuration, it is reasonable to expect that were a greater number of multipliers that performed signed multiplications available, then far fewer configurations would be needed and hundreds of cycles per symbol could be eliminated. However, Stallion only provides twice as many multipliers as the C6201, thus little benefit was achieved for this application. In the application examined in Chapter 5, the most commonly performed operation requires the use of ALUs, of which Stallion has 60, whereas the C6201 only has 6. By choosing an application that is better suited for the resources available on Stallion and by increasing the I/O bandwidth available to Stallion to facilitate the utilization of those resources, Chapter 5 shows a dramatic improvement in the relative performance of Stallion with respect to the C6201.

Chapter 5

5 WCDMA Receiver Implementation

To demonstrate the potential performance of a CCM within the Layered Radio Architecture, a single user Wideband Code Division Multiple Access Receiver was implemented. It was felt that the implementation of a three finger WCDMA receiver would give a better indication of the potential capabilities of the CCM approach since the despreading operation can be potentially performed in parallel, could potentially be performed with a minimum of reconfiguration, and need not be as dependent on multiplications as the Single User Adaptive Receiver. A clear limitation of the single user receiver was Stallion's input bandwidth. For this implementation, it was assumed that a modified Stallion was available that had twelve data ports instead of the normal four.

The primary goals of this implementation were to demonstrate the vast improvements in cycle performance when an implementation is well suited for a CCM architecture and when a sufficient level of input bandwidth is provided to fully utilize the processing resources of the CCM. A secondary goal was to get a feel for the level of power consumption for a CCM. The latter required additions to the original Stallion simulation that were made between the two implementations. These changes included incorporating bulk capacitance and operating voltage levels projected from the VLSI layout of Stallion as well as the inclusion of counts of bit flips. This chapter describes the implementation developed using the tools described in the Chapter 3 and the implementation's performance metrics.

5.1 WCDMA Receiver Algorithmic Description

The WCDMA receiver implementation is a three finger rake receiver for the Dedicated Physical Data Channel (DPDCH) using QPSK signaling. In particular, this receiver uses length 32 Orthogonal Variable Spreading Factor (OVSF) codes with eight pilot symbols per time slot. Note that for all possible spreading factors, there are exactly 2560 chips per time slot which arrive at a rate of 3.84 Mchips/sec. This results in a variety of different data rates.

For this receiver it is assumed that the input signal would be oversampled by a factor of two. It was further assumed that a delay searcher had been implemented independently of the WCDMA receiver and that these delay estimates are supplied to the receiver. It was also assumed that the received samples would be separated into the three complex fingers and delayed by the appropriate delay estimate external to the WCDMA receiver. Note that Gold code scrambling is also a part of the data channel, although this too is not implemented as part of this receiver. A detailed description of the operations of the WCDMA system can be found in [21] [22].

The despreading operation is essentially identical to a MAC operation except that the filter weights are constrained to the values ±1. During channel estimation the effects of the presumably Rayleigh channel corresponding to finger *f* are estimated from the pilot symbols located in the last eight positions of the timeslot output from the matched filter operation as

$$\hat{h}_f(m,n) = R_f(m,n) / p(m)$$

where (m, n) m refers to the m^{th} symbol in timeslot n, $\hat{h}(m, n)$ is a channel estimate, $R(m, n)$ is an output of the matched filter, and $p(m)$ is a pilot symbol. Since $p(m)$ only takes the values ±1, this operation can be rewritten as

$$\hat{h}_f(m,n) = R_f(m,n) \times conj(p(m)) / 2.$$

Eight of these channel estimates are formed and then averaged to form a channel estimate which is then used to compensate for the channel by evaluating the following equation for each finger, *f*,

$$z_f(m,n) = R(m,n) \times conj\left(\hat{h}_{f\ avg}\right)$$

where $\hat{h}_{avg}$ is the averaged channel estimate and *conj* denotes the conjugate operation. These finger symbol estimates are then summed to produce a final symbol estimate, $z(m,n)$.

A block diagram of the implemented WCDMA receiver is shown below in Figure 5.1. Note that the same spreading code, c_i is used for transmission on both the I and Q channels. Note that in this diagram, the operations are grouped by the physical layer mappings which are described in greater detail in Section 5.2.

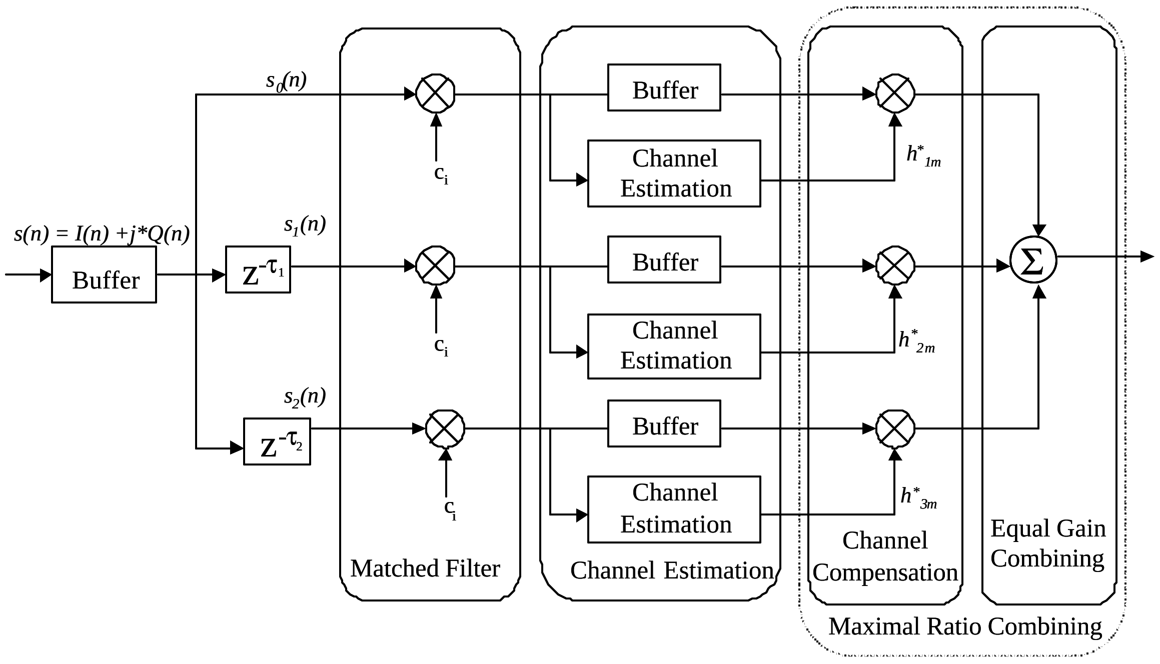


Figure 5.1 WCDMA Receiver Block Diagram and Physical Layer Mappings

5.2 Physical Layer Mappings

To support an implementation of the WCDMA Receiver on the Layered Radio Architecture, the single-user receiver was segmented into the following configurations:

- Matched Filter
- Channel Estimation
- Channel Compensation
- Equal Gain Combining

The Matched Filter configuration is used to perform the dispreading operation of the WCDMA receiver on three fingers. The Channel Estimation configuration is used to make estimates of the effects of the multipath channel corresponding to each finger of the receiver by using pilot symbols in the input sequence. These estimate are then used to weight each multipath component used in the Channel compensation configuration. The Equal Gain Combining configuration is used to sum together these weighted finger estimates to form the symbol estimates for a single timeslot. The Channel Compensation configuration and the Equal Gain Combining configuration together form a logical

Maximum Ratio Combining configuration. This breaking of the Maximum Ratio Combining configuration into two separate configurations was dictated by the resource limits of Stallion. The following describes how each of these physical layer mappings was implemented.

5.2.1 Matched Filter

The Matched Filter configuration, shown in Figure 5.2 is used to perform the despreading operation for the WCDMA receiver. A block of functional units is used to generate the length 32 spreading code on a sample by sample basis. The despreading operation is split into two steps, a signed multiply, denoted as SM, and an accumulation. The signed multiply is performed in a single functional unit by pointing the left and right registers to the same input source, negating the left register value and then using the conditional mode to choose the negated or unaltered variable based on an input flag. To complete the despreading operation these results are then accumulated by using the feedback capabilities of the functional unit. To minimize overhead time, the accumulators, denoted by ACC, can be reset setting the Left Input Register to clear to zero to clear if Flag2 is zero, which is then determined by the valid bit on the input. A similar approach is used to generate the reset (GR) needed to initialize the spreading code generator. Note that as the spreading code generator only changes every two cycles, the configuration layer must also insert an extra valid zero into the data stream, otherwise the spreading code would lead the input streams by a cycle.

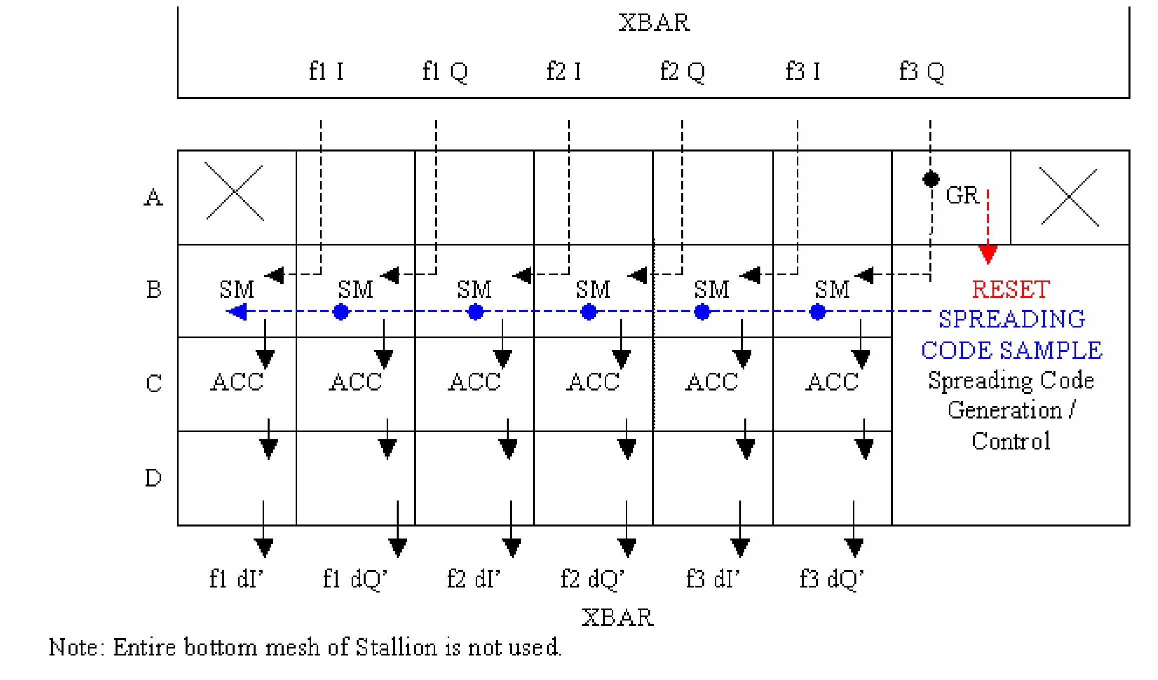


Figure 5.2 Matched Filter Physical Layer Mapping

The section of functional units used to implement the generation and control of the spreading code is shown below in Figure 5.3. Note that each constant is used to store 16 bits of the spreading code. Arbitrary spreading codes can be loaded into this structure if the configuration layer is instructed to treat these constants as embedded variables. The first sixteen chips are stored on the left and the last sixteen chips are store on the right. When the reset flag goes high, the constants are chosen as the output of the middle functional units. This is then sent to the functional units to the north for both functional units. These functional units perform a right shift, and the bit being shifted out is used as the chip used in the signed multiplication.

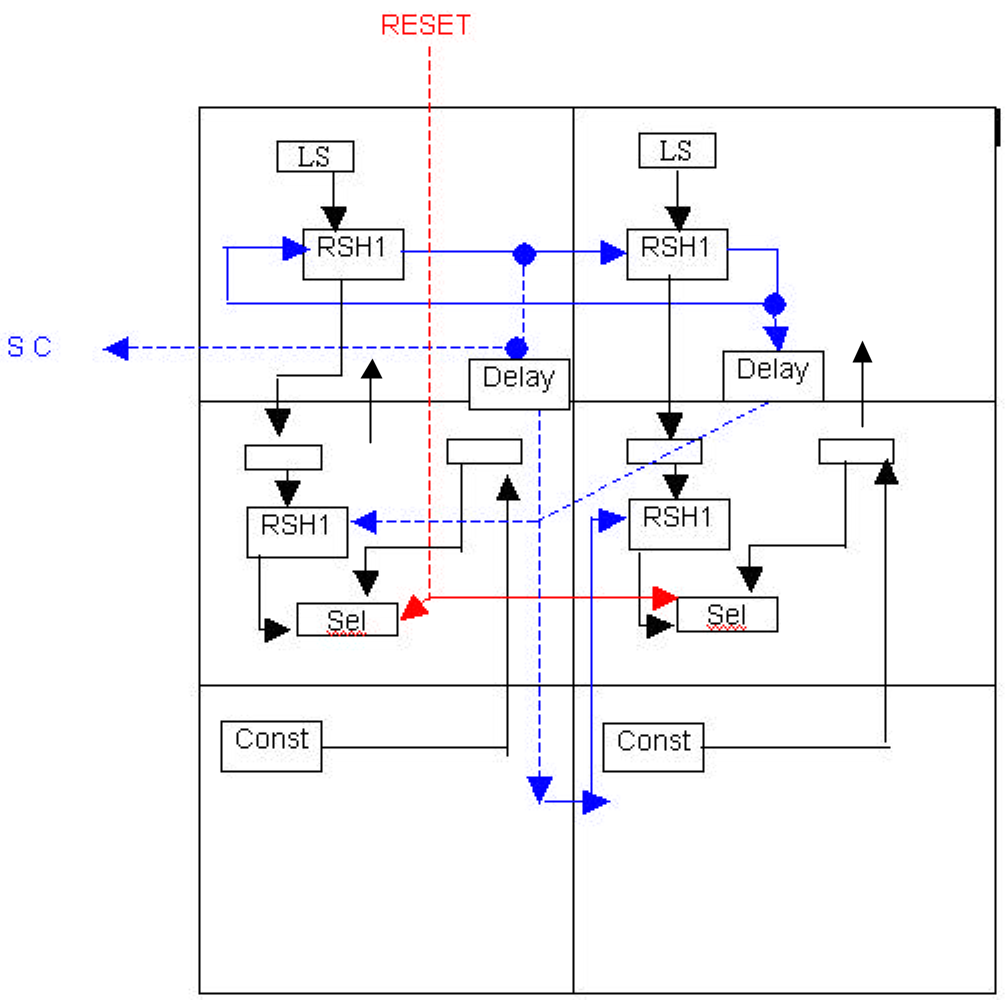


Figure 5.3 Spreading Code Generation and Control

This structure can be easily modified to increase or decrease the length of the spreading code by factors of 16 by replicating or removing the right half of this block. Note that as the length of the spreading code increases there are necessarily fewer functional units available. This has the potential to greatly impact the number of cycles required to implement the receiver. In order to operate with an oversampling rate other than two a counter has to be added which in general occupies two functional units. Also note that with longer spreading codes, scaling may be needed to limit overflow in the accumulators.

5.2.2 Channel Estimation

The Channel Estimation configuration, shown below in Figure 5.4, is used to form the channel estimates based on the effects that the pilot symbols encounter for a single finger. Note that the same configuration is replicated in the bottom mesh. Thus to support the operation across three fingers this configuration must be run twice. It is the responsibility of the configuration layer to ensure that the each finger's pilot bits have been extracted and arranged for use in this configuration. The basic operation of the Channel Estimation configuration is similar to the matched filter except that the channel estimates necessitate a true complex despreading operation and thus uses a complex signed multiplications. A complex signed multiplication consists of four signed multiplications (SM) an addition (L + R) and a subtraction (L – R). This is then averaged by right shifting (SHR1) by three and right shifted once more to ensure that the maximum value is at a level appropriate for the next stage (the complex sums mean that with noise the value could be 16 times larger than normal). This scaling requires a right shift by four which requires the use of four functional units each shifting right by one (SHR1). Provisions are again made for resetting this configuration so that this can be reused with a minimum amount of overhead.

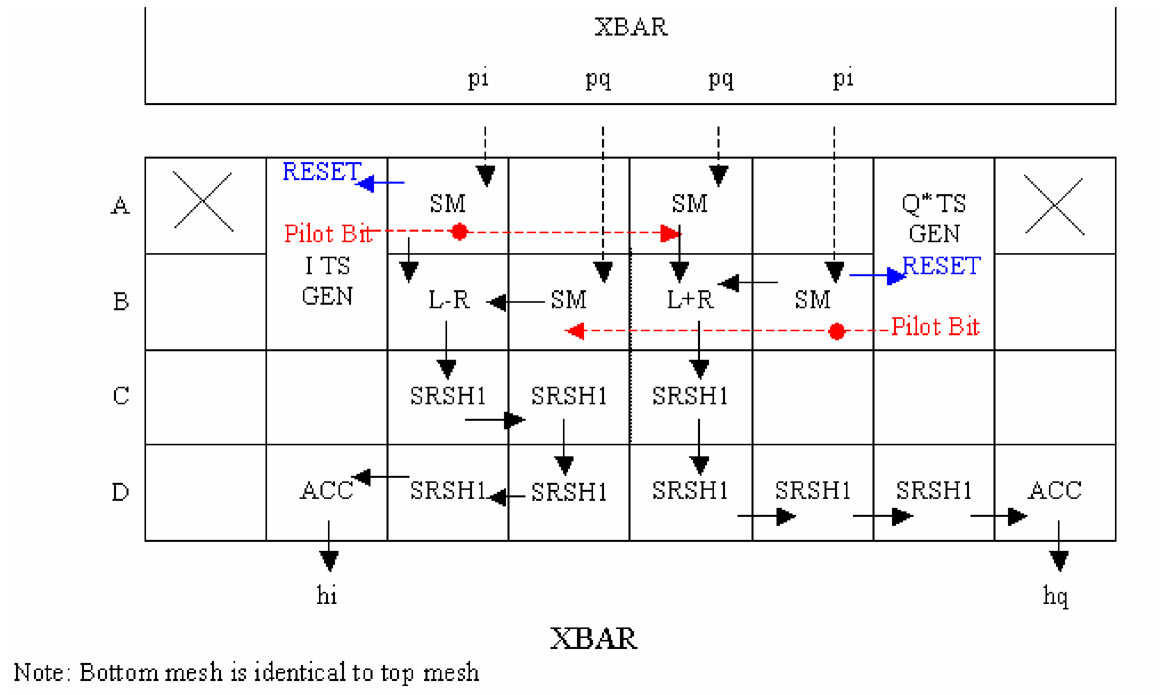


Figure 5.4 Channel Estimation Physical Layer Mapping

The I pilot bit generation block is shown below in Figure 5.5. The Q pilot bit generation is identical to this but inverted. The operation of the I pilot generation block is similar to the spreading code generation used in the matched filter. Note that if the configuration layer treats the constant as an embedded variable, then the pilot sequence can be readily changed. Also note that this same configuration can be used to support QPSK operation by using independent pilot sequences on the I and Q channels. Finally, note that since the pilot sequence is only of length 8, only the right eight most positions of the constant should be filled. In order to initialize these pilot bit generators, the stream must begin with an invalid word followed by a valid zero.

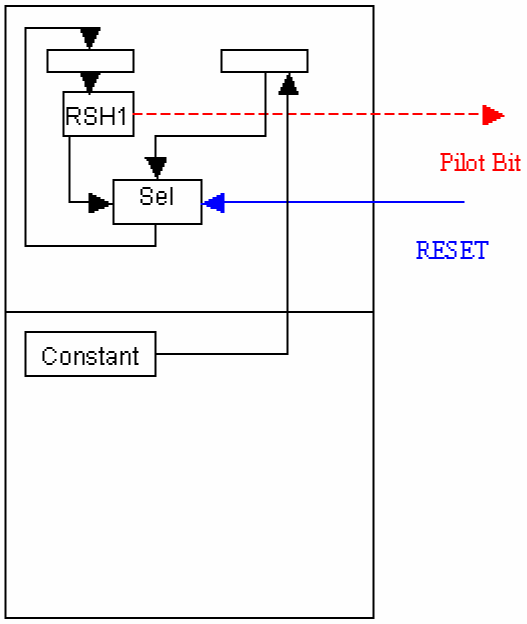


Figure 5.5 I Pilot Bit Generation Block

5.2.3 Channel Compensation

The Channel Compensation Configuration is implemented in a manner identical to the Weight Updating configuration shown in Figure 4.11.

5.2.4 Equal Gain Combining

The Equal Gain Combining Configuration, shown in Figure 5.6, is used to sum up the data vectors output from the Channel Compensation configuration. This is simply done by adding together the weighted data symbols of each finger. Note that the configuration layer must delay the input of the vectors from the second finger by a cycle.

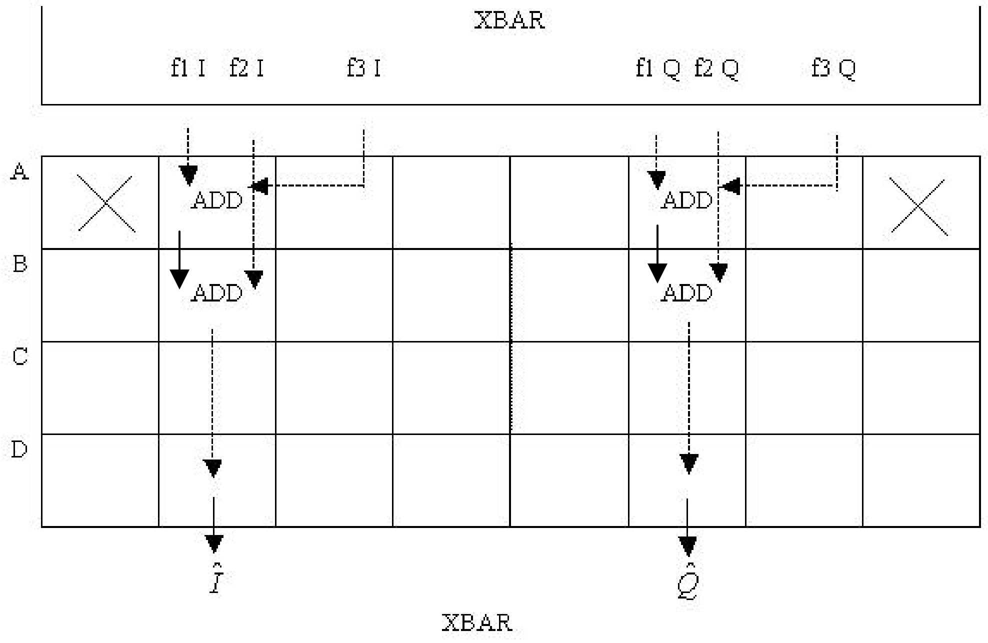


Figure 5.6 Equal Gain Combining Physical Layer Mapping

5.3 Configuration Layer

This section details how the Configuration Layer was programmed to support the implementation of the WCDMA Receiver. This Section details the embedded variables, nonembedded variables, data memory, configuration, and stream header information needed to properly load the configuration onto Stallion for use in the WCDMA Receiver used to set the parameters of the controller chip object and that could be used to program a configuration layer implementation. These details are broken down by controller module, configuration module and data module.

5.3.1 Memory Allocations and Controller Module State Machine

This Section describes the basic operations of the configuration layer. This includes the allocation of configuration memory and data memory and also describes the Controller

Module State Machine used to determine the general operation of the Layered Radio Architecture.

For the WCDMA Receiver, the Configuration Memory was allocated as shown in Table 5.1. The left column indicates the memory location and the right column gives the Configuration Mneumonic used to differentiate the Physical Layer mappings in Section 5.2. Note that these configurations are stored beginning at configuration number 9 under the assumption that configurations numbers 0-8 contain the Single User Adaptive Receiver Information. Note that the Matched Filter and Matched Filter (Pilot) use the same physical layer mapping, but have different sinks for their results. Similarly, three different channel compensation configurations are used due to the need to maintain three different sources. Since these configurations are specified to have 0 programming cycles, no penalty is incurred in the physical layer from having to separate these operations.

Table 5.1 Configuration Memory Allocations

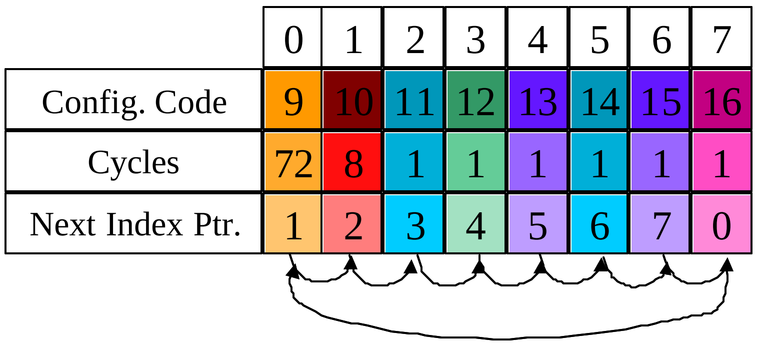
Configuration Number	Configuration Mneumonic
9	Matched Filter
10	Matched Filter (Pilot)
11	Channel Estimation (0,1)
12	Channel Estimation (2)
13	Channel Compensation (0)
14	Channel Compensation (1)
15	Channel Compensation (2)
16	Equal Gain Combining

For the Single User Adaptive Receiver, I and Q memory allocations remain the same. The NEV portion of the data memory is allocated as shown in Table 5.2. Note that the NEV 0 – NEV 5 are used to store the outputs of the matched filter operation as well as the channel compensation operation.

Table 5.2 Data Memory Allocation in Configuration Layer

NEV Number	Variable Mnemonic
0	$R_{0,I}(m,n)$ / $z_{0,I}(m,n)$
1	$R_{0,Q}(m,n)$ / $z_{0,Q}(m,n)$
2	$R_{1,I}(m,n)$ / $z_{1,I}(m,n)$
3	$R_{1,Q}(m,n)$ / $z_{1,Q}(m,n)$
4	$R_{2,I}(m,n)$ / $z_{2,I}(m,n)$
5	$R_{2,Q}(m,n)$ / $z_{2,Q}(m,n)$
6	$p_{0,I}(m,n)$ / $h_{0,I}(n)$
7	$p_{0,Q}(m,n)$ / $h_{0,Q}(n)$
8	$p_{1,I}(m,n)$ / $h_{1,I}(n)$
9	$p_{1,Q}(m,n)$ / $h_{1,Q}(n)$
10	$p_{2,I}(m,n)$ / $h_{2,I}(n)$
11	$p_{2,Q}(m,n)$ / $h_{2,Q}(n)$
12	$z_I(m,n)$
13	$z_Q(m,n)$

The Control Module State Machine is shown in Figure 5.7. The configuration code, listed in the top row refers to the configuration number in configuration memory shown in Table 5.1. Note that the Matched Filter configuration runs 80 times to form the symbol estimates for each of the 80 symbols in the time slot and constitutes the majority of the processing cycles.



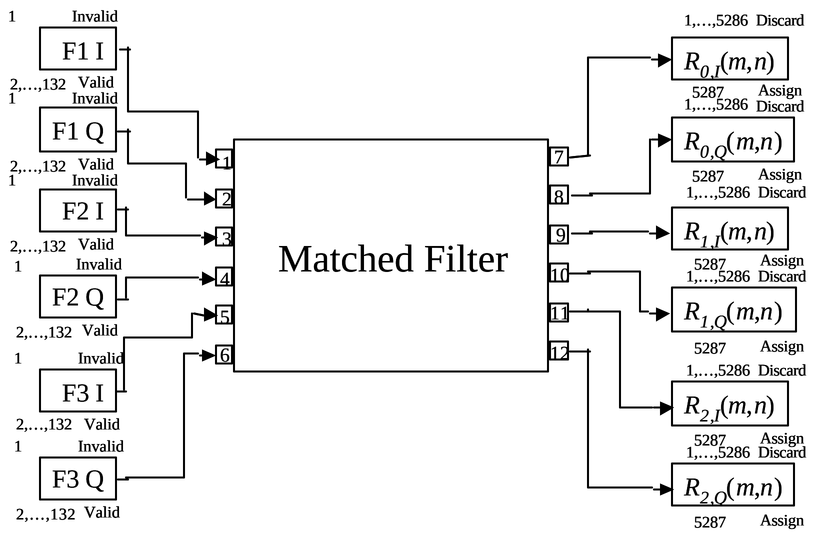
	0	1	2	3	4	5	6	7
Config. Code	9	10	11	12	13	14	15	16
Cycles	72	8	1	1	1	1	1	1
Next Index Ptr.	1	2	3	4	5	6	7	0

Figure 5.7 Control Module State Machine

5.3.2 Configuration Information

This Section describes the information needed to load and execute each of the configurations required for this implementation through a series of diagrams. A description of the presentation conventions used in the following diagrams is given in Section 4.3.2.

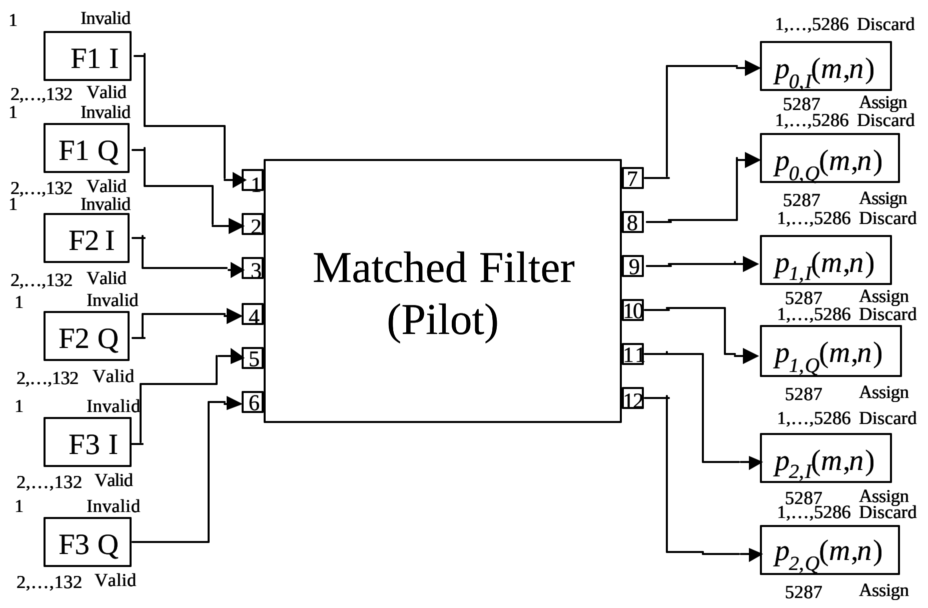
Figure 5.8 shows the information needed to load and run the Matched Filter configuration. Note that this configuration only operates on the data bits (1-72) of the timeslot. Figure 5.9 shows the information needed to load and run the Matched Filter configuration as used for the pilot bits (73-80) of the timeslot. The need for different configurations is a result of the combination of having different stream sinks. Note that the pilot bit configuration has no programming cycles which means the existing physical layer configuration remains.



Cycles	
Programming	73
Processing	5287
Reprogram	False

Embedded Variables		
c_i (0-15)	HD7	Direct
c_i (16-31)	HD8	Direct

Figure 5.8 Matched Filter Configuration for Data Bits



Cycles

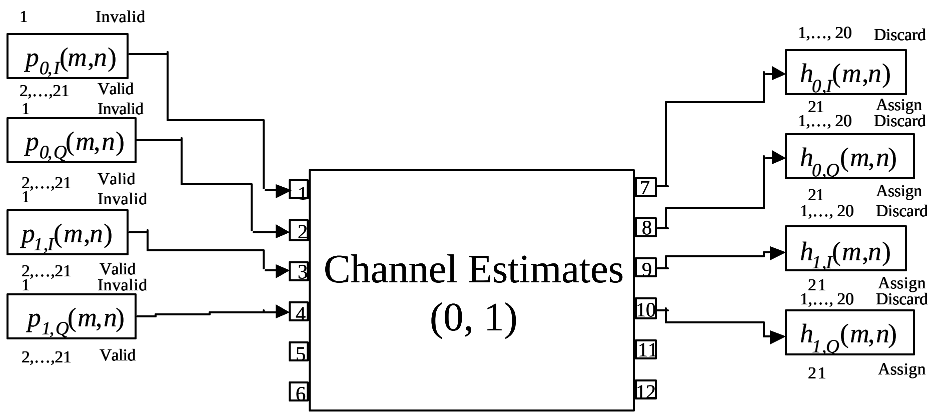
Programming	0
Processing	5287
Reprogram	False

Embedded Variables

none

Figure 5.9 Matched Filter Configuration for Pilot Bits

Figure 5.10 shows the information needed to load and run the Channel Estimation configuration for fingers 0 and 1. Note that in order to initialize the pilot symbol generator, an invalid word and a valid zero must be inserted at the beginning of the stream. Though not shown, there is also a configuration reserved for Channel Estimation on finger 2 which takes 0 programming cycles and has different sources and sinks.

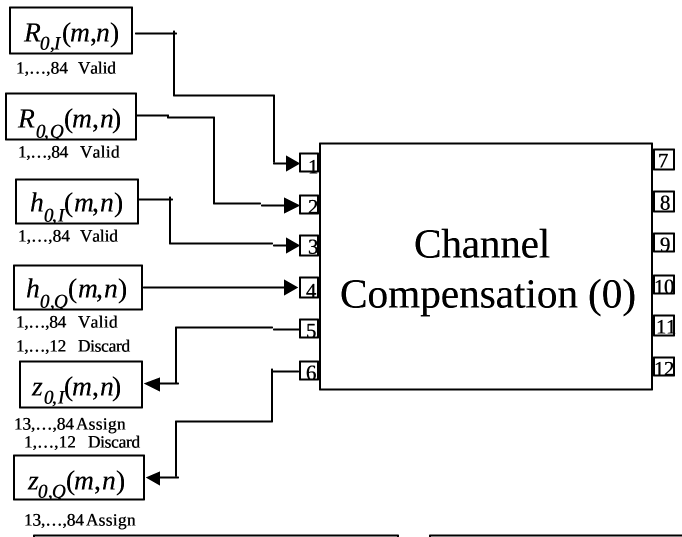


Cycles	
Programming	69
Processing	21
Reprogram	False

Embedded Variables		
p_I	HB2	Direct
p_Q	HA7	Direct

Figure 5.10 Channel Estimates Configuration

Figure 5.11 shows the information needed to load and run the channel estimation configuration for finger 0. Though not shown, there are also configurations in configuration memory for fingers 1 and 2 which are identical to this configuration except for having different sources and sinks and requiring 0 programming cycles.



Cycles	
Programming	132
Processing	84
Reprogram	False

Embedded Variables

None

Figure 5.11 Channel Compensation Configuration

Figure 5.12 shows the information needed to load and run the Equal Gain Combining configuration. Note that for finger 1, the first cycle loads a valid zero instead of from the NEV.

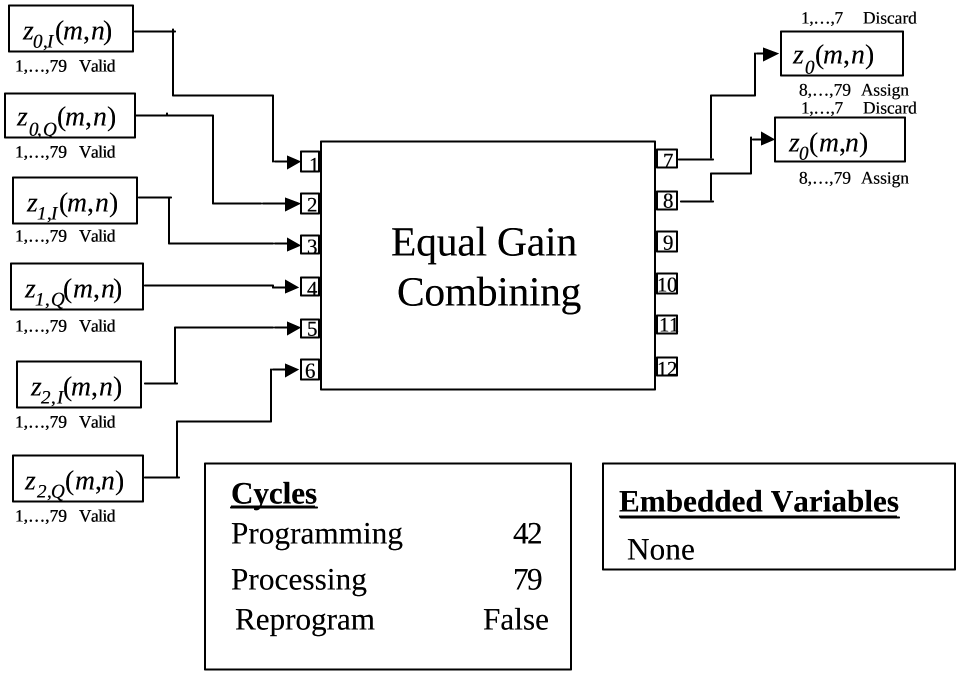


Figure 5.12 Equal Gain Combining Configuration

5.4 Performance Results

This Section describes the results of this implementation of the WCDMA-TDD receiver. For this implementation, the primary objectives were measuring the performance of the Layered Radio implementation in terms of cycles, utilization and power consumption.

5.4.1 Resource Utilization

The resource utilization statistics for the WCDMA receiver are tabulated in Table 5.3. Notice that the Matched Filter configuration achieves an extremely high utilization ratio as it is able to spend the bulk of its time processing and little time programming. As this is the most cycle intensive portion of the implementation, this also results in a reasonably high utilization for the entire implementation and a relatively low cycle count. Also note that most configurations achieve higher adjusted utilization statistics than for the single user adaptive receiver, which serves to emphasize the connection between input bandwidth and chip utilization. For convenience, the configurations that are only differentiated by source or sink are grouped together.

Table 5.3 WCDMA Performance Statistics

	Matched Filter	Channel Estimation	Channel Compensation	EGC	Total
Functional Units	30	44	56	10	31.53
Multipliers	0	0	4	0	0.26
Gross Component Utilization	46.88%	68.75%	93.75%	15.63%	49.67%
Adjusted Component Utilization	29.69%	43.75%	15.63%	6.25%	28.57%
Programming Cycles	73	69	132	42	316
Processing Cycles	5287	42	252	79	5660
Processing Efficiency	98.64%	37.84%	65.63%	65.29%	94.71%
Number of Streams	12.00	9.00	6.00	8.00	11.48
Streaming Efficiency	100%	75%	50%	66.67%	95.65%
Gross Utilization	46.24%	19.51%	30.76%	6.80%	43.95%
Adjusted Utilization	29.29%	12.42%	5.13%	2.72%	26.88%

After a cursory comparison to the Single User Adaptive Receiver, it is readily apparent that the choice of application greatly impacts utilization and performance. To give a better feeling for how the choice of operating parameters impacts the utilization of Stallion, the WCDMA receiver was mapped onto Stallion for a variety of different spreading code lengths and oversampling factors. The percent cycle utilization, *CU%*, was then estimated for each of these mappings. *CU%* is simply calculated as the ratio of the cycles required for a timeslot compared to the total number of cycles available for a timeslot. The total number of cycles available for a timeslot can be found as follows:

Total time = (2560 chips / timeslot) / 3.84 (Mchips / sec) = 667 μs / timeslot.

Available Cycles = Clock Rate (Hz) × (time / timeslot)

= (50 MHz) × 667 μs / timeslot = 33,333 cycles / timeslot

Since the number of cycles used by the Layered Radio Architecture for a timeslot is equal to 5976 cycles, *CU%* can be calculated as

CU% = (5976 cycles / timeslot) / (33,333 cycles / timeslot) × 100% = 17.93%.

5.4.2 Parameter Sensitivity

After completing the implementation and analysis of the WCDMA receiver, it was decided to explore the sensitivity of the utilization statistics to the choice of WCDMA parameters, in particular to OVSF and F. So theoretical mappings to the Stallion processor were done for these same configurations over the range of allowable OVSF lengths (4 to 512) and for a number of different oversampling factors. The projected cycle utilizations for these mappings are shown in Figure 5.13 where each line represents a different oversampling factor and each point represents a mapping at a different OVSF length.

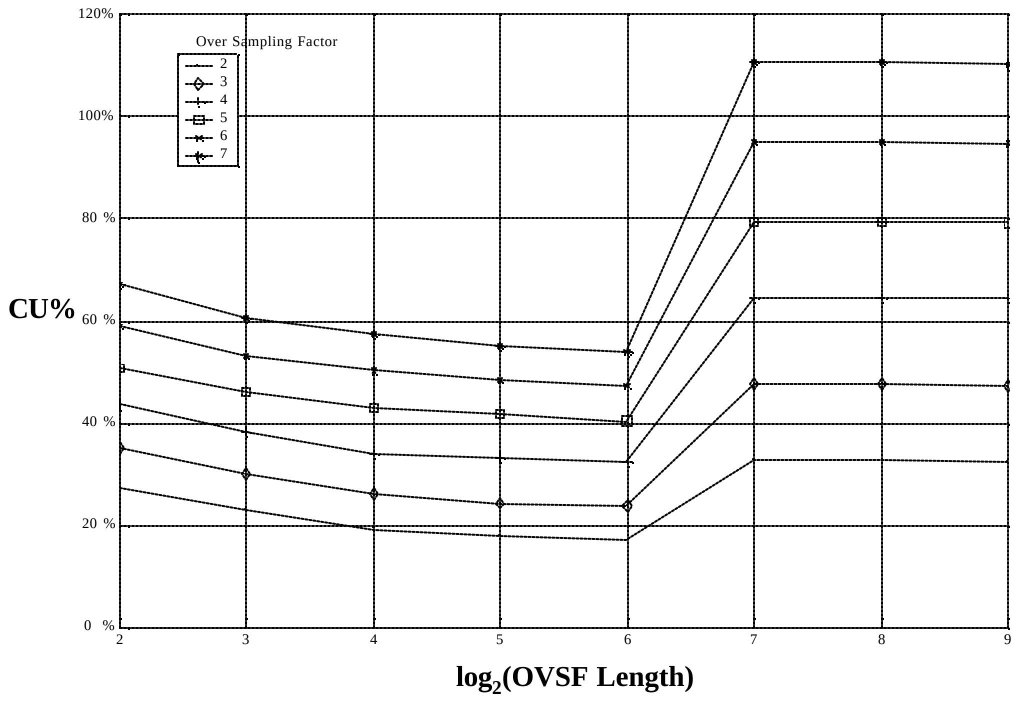


Figure 5.13 Projected Relationship Between Stallion Utilization, Sampling Rate, and Spreading Code Length for WCDMA Receiver

Notice that the general trend is a decrease in CU% as the OVSF length increases. This results from a general reduction in the amount of overhead the implementation has to incur from handling additional bits. This overhead comes from two primary sources:

- two cycles must be wasted for each bit as part of the matched filter configuration in order to reset the spreading code generator.
- with more bits, more iterations are spent in the less efficient configurations.

Also, as expected, there is a nearly linear increase in %CU as the oversampling factor increases. However, notice that at an OVSF length of 128, there is a sudden and sharp increase in *CU*%. This is caused by having an OVSF that requires an OVSF generator that is so long that there is not enough space to perform the signed multiply and accumulate operations for all of the fingers simultaneously. This suggests that the utilization performance of a CCM will be “chunky” so that depending on the waveform and the CCM, small changes in a parameter may result in large changes in utilization statistics.

This effect was also dependent on the number of processing elements that the CCM has. For instance if Stallion had many more processing resources, then the matched filter configuration could have been implemented on a single Stallion. Fortunately, one of the advantages of the Stallion architecture is that it can be readily extended to multiprocessor configurations simply by connecting the dataports of one Stallion processor to the data ports of another Stallion processor [14]. However, this necessitates the use of longer programming streams, but allows more calculations to be performed in parallel potentially further reducing the cycle count. As a tradeoff, however, area increases significantly.

To further understand the interrelation between these parameters, the WCDMA receiver with *OVSF* = 128 and *F* = 2, was mapped to a number of different chip areas as varied by quarter Stallions. The choice of these parameters merit some comment. First, an *OVSF* of 128 was chosen as that is the point at which the matched filtering configuration can no longer be performed on a single Stallion. As the matched filter configuration is the dominant configuration, there would be little benefit seen from increasing area with a smaller OVSF. Second, quarter Stallions were chosen as that is the smallest subdivision of Stallion that preserves the basic ratio of components in the chip (15 FUs to 1 Multiplier).

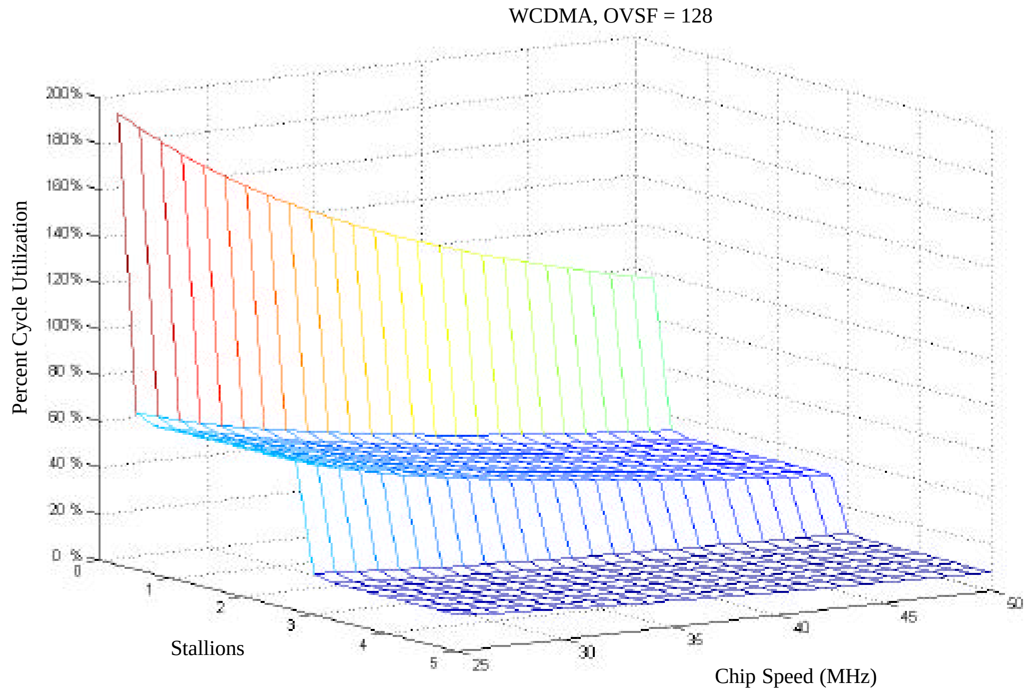


Figure 5.14 Projected Tradeoffs Between Utilization, Area, and Speed

5.4.3 Power Consumption

As part of this implementation, the Stallion simulation was enhanced to perform power estimates by using estimated bulk capacitance measurements and operating voltages from [15] and by counting the number of bit flips that occur at the boundary of the functional units. Then the power consumption could be calculated by applying the following equation

$$P = \frac{1}{2} nCv^2 f$$

where n is the number of toggling bits, C is the capacitance, v is the operating voltage, and f is the clock frequency.

Based on this calculation the power consumption over a timeslot was measured from the physical layer simulation. These results are summarized in Figure 5.15. The top graph shows the activity over the entire timeslot, while the bottom graph depicts only the portion of the timeslot for which the physical layer is active. During the active portion of the processing, 2.9 W are consumed on average. However over the entire timeslot, only 500 mW is consumed due to the long inactive period as only 17.8% of the available cycles are occupied. This compares reasonably well with a low power 3 finger ASIC designed at UC Berkeley which consumes 22 mW when operating under the same supply voltage [23].

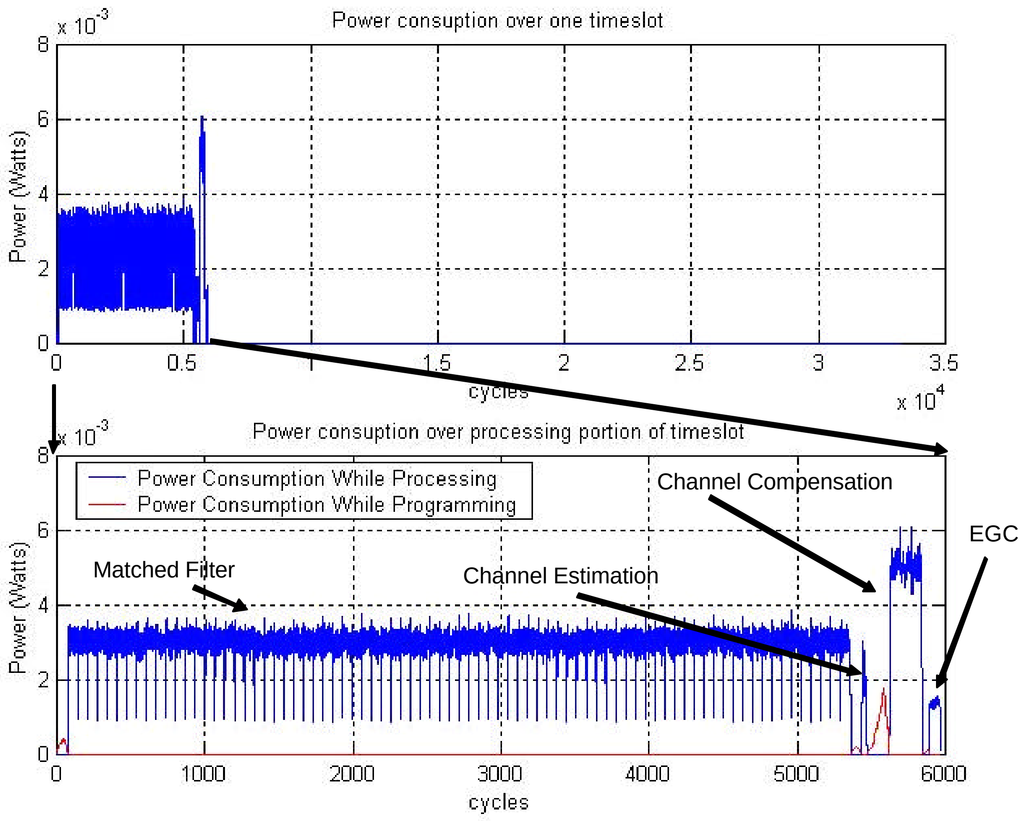


Figure 5.15 Estimated Power Consumption in WCDMA Implementation

There are a few things to notice from these plots. First, the peak instantaneous power consumption is relatively high. This is primarily due to the fact that little effort was made in the chip design to minimize power consumption or to use state-of-the-art technology. It would be expected that a commercial layout would achieve much better results. However, the power consumption average over the timeslot is significantly lower since so few cycles are required to implement the WCDMA radio. Finally, comparing this plot to the utilization statistics, it can be seen that the gross device utilization directly impacts power consumption in a CCM.

5.5 *TMS320C6201 Comparative Implementation*

Again, in order to make a fair evaluation of the performance of the Layered Radio Architecture, these same configurations were also mapped to an implementation on TI's TMS320C6201 processor. This Section describes the expected cycle counts for each for these same configurations.

5.5.1 Matched Filter

This is just a complex filter and can be implemented in a manner similar to that described in Section 4.3.1 without the final addition. However since no cross terms are required, both the I and Q components can be simultaneously created. The total number of cycles required to form a single symbol estimate with C chips oversampled by a factor of F is then given by

Prolog = 5 (initial load) + 2 (initial multiply) = 7

Kernel = 1 (Load, Multiply & Accumulate) * ($C \times F$ -2) = $C \times F$ - 2

Epilog = 1 (Final Multiply, Penultimate Accumulate) + 1 (Final Accumulate) = 2

Store = 1

Total = 7 (Prolog) + 118 (Kernel) + 2 (Epilog) + MAC Sum = $C \times F$ +8 cycles.

This would need to be repeated for each symbol and for each finger. Thus the total cycles required for running the matched filter configuration for a timeslot is given by

Cycles per Timeslot = $3\times n\times (C\times F +9)$

Note that since $n \times C = 5120$ per timeslot, this configuration is relatively insensitive to changes in terms of which OVSF code is used. However, longer spreading codes with fewer symbols per timeslot, will suffer less of an overhead penalty. Note that the exact same performance could be achieved by conditionally using the ADD2 and SUB2 instructions of the two .S units.

5.5.2 Channel Estimation

The Channel Estimation configuration is essentially a complex filtering operation repeated three times. Following the logic employed in Chapter 4, this can be expected to be given by

Prolog = 5 (initial load) + 2 (initial multiply) = 7

Kernel = 1 (Load, Multiply & Accumulate) * (8 -2) = 6

Epilog = 1 (Final Multiply, Penultimate Accumulate) + 1 (Final Accumulate) = 2

Total = 7 (Prolog) + 118 (Kernel) + 2 (Epilog) = 15 cycles

These two MAC results would then be added together and then divided by 16, which can be performed by a right shift by 4 in a single cycle and stored (a single cycle is required to start writing the results). This means that a total of 18 cycles are required to form the I portion of the channel estimate. This is then for Q for a total of 36 cycles per finger and a total of 108 cycles for all three fingers.

5.5.3 Channel Compensation

This is essentially just a complex multiplication performed on each symbol on each finger for a total of 3*n* complex multiplications. However, it is possible to load the coefficient for each finger before proceeding to perform the required multiplications. After the initial loading of the complex coefficient for a finger, the process can be pipelined so that a complete symbol is generated each cycle. Then it is expected that the total time required for a finger will be equal to

total time for finger = 1 (coefficient load) + 5 (symbol load latency) + 1(multiplier latency) + 1 (ALU latency) + 1(coefficient store) + n = 9+n

The total number of cycles required for all the processing associated with the Channel Compensation configuration for three fingers over an entire timeslot is equal to three times that number or 27 + 3*n*.

5.5.4 Equal Gain Combining

For each output from the equal gain combining stage, two additions, three loads and a store must be performed. This process must be repeated for both of the *I* and *Q* outputs. Unfortunately, this cannot quite be pipelined so that an output can be generated every cycle as a .D unit cannot simultaneously load and store and no .D unit can perform more than two simultaneous loads or stores (which also presumes data packing anyways). Realistically, these values are not going to be interspersed such that data packing can be exploited. Thus it will not be possible to pipeline this operation in a meaningful way (though a little is possible); therefore performance suffers in a C62 implementation. The following shows the code needed to form one output symbol (both I and Q). Notice that for each symbol to be calculated, 10 cycles are required.

```
;assume A4 has fI1, A5 has fI2, A6 has fI3
;assume B4 has fQ1, B5 has fQ2, B6 has fQ3
;assume A10 is the output address for I
;assume B10 is the output address for Q
        LDH     .D1 *A4++[2], A7
||      LDH     .D2 *B4++[2], B7
        LDH     .D1 *A5++[2], A8
||      LDH     .D2 *B5++[2], B8
        LDH     .D1 *A6++[2], A9
||      LDH     .D2 *B6++[2], B9
        NOP     4
        ADD     .L1 A7, A8, A0
||      ADD     .L2 B7, B8, B0
        ADD     .L1 A0, A9, A1
||      ADD     .L2 B0, B9, B1
        STH     .D1 A1, *A10++[2]
||      STH     .D2 B1, *B10++[2]
```

5.5.5 C6201 Performance Summary and Comparison to Stallion

Table 4.6 summarizes the cycles required to implement the WCDMA receiver on the C6201 by configuration. Notice that the vast majority of the cycles are consumed in the Matched Filter configuration. From examining this table, it can be seen that the C6201 implementation requires approximately three times as many cycles as the implementation using Stallion. This is to be expected as the Matched Filter configuration is the most cycle intensive configuration for both implementations and Stallion is capable of performing the calculations for all three fingers simultaneously while the C6201 can only perform the operation on one finger at a time.

Table 5.4 C6201 WCDMA Implementation Statistics

Configuration	Total Cycles	Cycles for C = 32, F = 2
Matched Filter	$3\times n\times (C\times F + 9)$	17520
Channel Estimation	108	108
Channel Compensation	$27+3n$	267
Equal Gain Combining	$10n$	800

Total	$135+n\times(3C\times F+40)$	18,695

5.6 WCDMA Receiver Summary

The WCDMA receiver implementation further demonstrates the flexibility of the Layered Radio Architecture by showing that it can support two very different waveforms. A relative weakness of the configuration layer was also highlighted wherein multiple configurations are required for a single physical layer mapping. These sorts of problems would likely be alleviated by implementing the basic functionality of the configuration layer on a microprocessor rather than a FPGA.

This implementation also demonstrates the relative costs/benefits of implementing a waveform on a CCM as opposed to a top of the line DSP. Consider how changes in operating parameters affect changes in device utilization for the C6201. Figure 5.16 shows how the number of cycles required to implement the WCDMA receiver for varying OVSF lengths and a oversampling factor of 2. Notice that the C6201 mapping reduces in cycles with increasing *OVSF* length faster than the Stallion mapping does. This is a result of the C6201's extra overhead required overhead required to initially fill and clear the pipeline for each bit (9 cycles for the C6201 vs 2 cycles for Stallion) as well as the fact that for *OVSF* lengths less than 128, the C6201 must incur three times this overhead because of repeating the operation for each finger. So notice that with the increased I/O and the choice of an application that is better matched for Stallion's resources, Stallion is able to perform much better relative to the C6201 than was seen for the Single User Adaptive Receiver implementation.

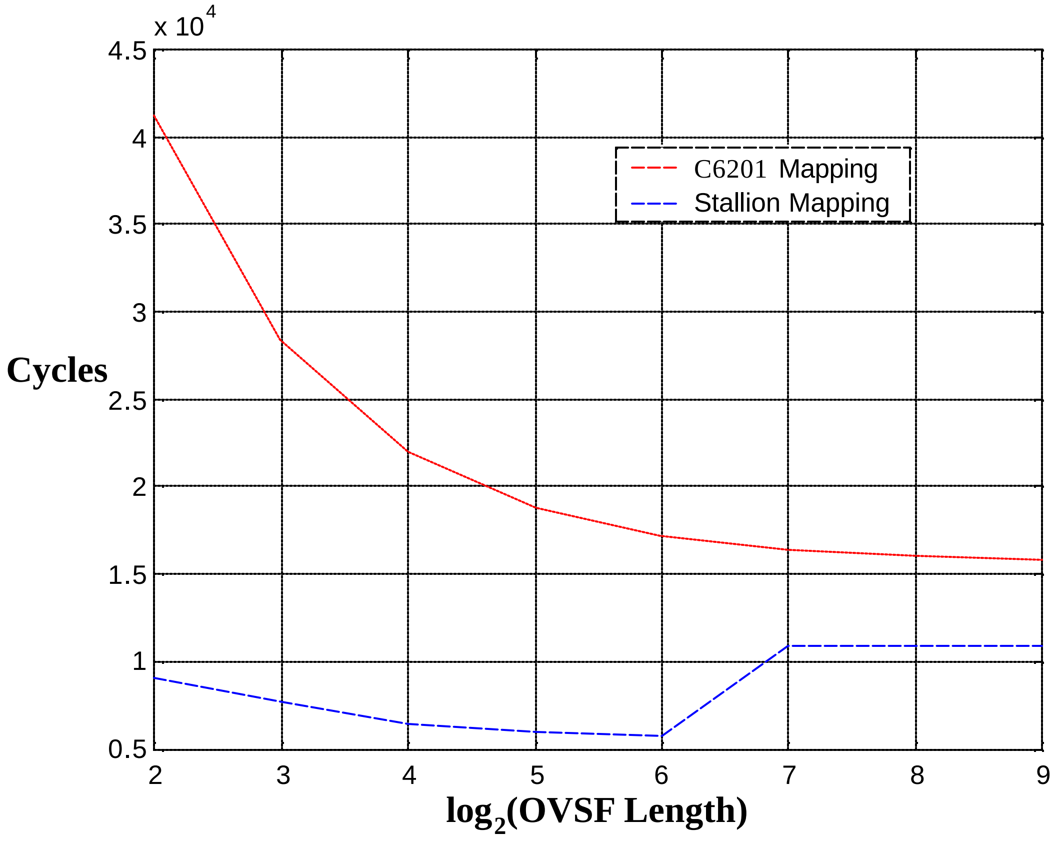


Figure 5.16 Cycle Consumption for Varying OVSF Lengths

Recall, however, that the C6201 operates at a clock rate four times the rate of Stallion (200 MHz vs 50 MHz). So the available cycles per timeslot for the C6201 is equal to 133.333 kcycles. Thus in terms of available cycles and cycle utilization, the interpretation of processing efficiency changes somewhat. Figure 5.17 repeats this analysis but in terms of cycle utilization. Note when treated in terms of *CU%*, the C6201 actually outperforms Stallion. Again, this is somewhat to be expected as the C6201 has a factor of 4 advantage in clock rate with only a disadvantage of a factor of 3 in terms of parallelism. Additionally, as previously noted the C6201's overhead decreases much faster than for Stallion whose overhead is primarily a result of the time required to program each configuration which is relatively insensitive to changes in OVSF.

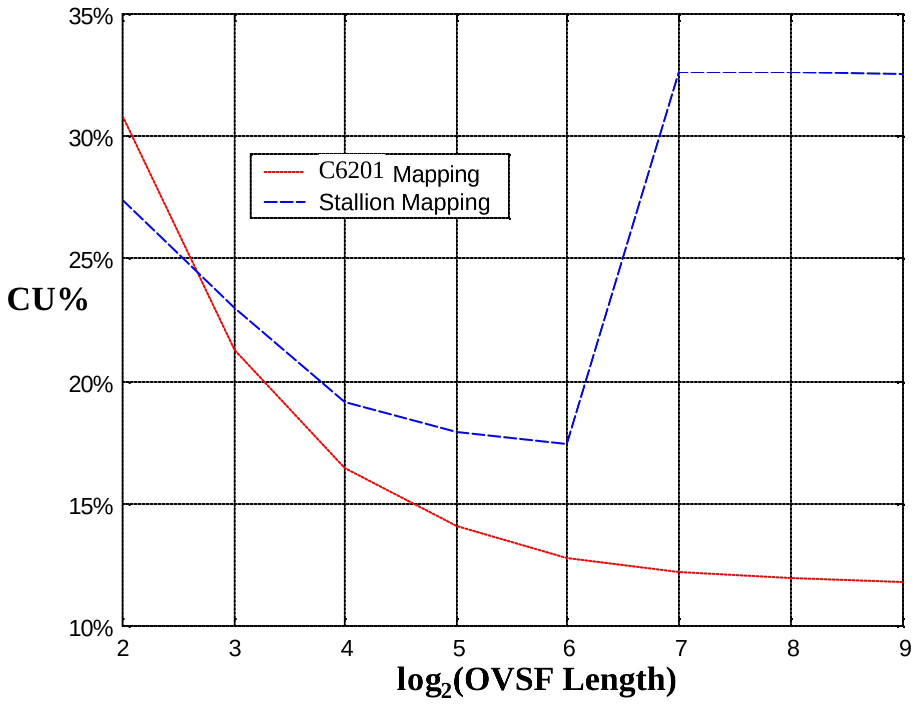


Figure 5.17 Percent Cycle Utilization for Different Physical Layer Implementations

A comparison of the raw computational power, as opposed to the utilized computational power, provides a generalized indication of the capabilities of the architecture in general. This can be computed by determining the amount of remaining resources after performing the WCDMA implementation. In this implementation, Stallion used 17.9% of its clock cycles and 28.6% of its computational resources thus using a total of 5.1% of available resources. The C6201 used 14.0% of its cycles, but 72.9% of its computational resources, meaning that it used 10.2% of its available resources. Thus it is shown that the Stallion chip has more remaining resources and thus has a higher raw computational power.

It should also be noted that Stallion achieves these gains using a smaller area. The Stallion chip's die is 7.95 mm a side [15] whereas the die for the C6201 is 9.78 mm a side [25]. Thus the Stallion chip occupies 63.2 mm^2 while the C6201 occupies 95.64 mm^2; thus the Stallion chip only occupies 66.1% of the area of the C6201 chip. Since the Stallion chip occupies a smaller area and demonstrates a higher computational power than the C6201, Stallion can also be said to achieve a higher computational density than the C6201. Again this is as expected, as CCMs do not have as much overhead as DSPs and include significantly more computational components. It is also important to note that the Stallion chip uses .25 µm technology whereas the C6201 uses .18 µm technology, so these gains should be attributed to the CCM approach instead of the fabrication technology.

A factor that can be compared between these two processing solutions is in terms of power consumption. In [24], it is given that the C6201 consumes 1.94 W during high activity operations and 0.82 W when idle. Applying the same logic as used to calculate the average power consumption for Stallion, we see that the total power consumption is 1.04 W which is roughly twice as much as the 0.50 W required for Stallion. This is not entirely the result of Stallion being clocked at one-fourth the rate of the C6201. While increasing the clock rate by a factor of 4 would increase the active power level by a factor of 4, the fraction of time in which the chip is active per timeslot would also decrease by a factor of 4, offsetting any increase in power consumption. Also note that since the C6201 uses 1.8 V for internal signals as opposed to the 3.3 V used in Stallion. So even though the C6201 has a significant advantage in power consumption over Stallion due to voltage (a factor of approximately 3.3), the CCM still outperforms the C6201. It should be further noted that Stallion was not designed with a particular intent to minimize power consumption, so this result is especially surprising. Thus, from this result, it is expected that a CCM will in general be more power efficient than a DSP. In general, this corresponds with what would be expected from a more specialized processor that requires less overhead for its basic operation.

Chapter 6

6 Summary and Conclusions

This thesis examined the performance of the Layered Radio Architecture and CCM technology for use in a software radio through a number of implementations. Software radio is becoming increasingly important while waveforms are becoming increasingly complex. An effective software radio design requires flexibility at the application, algorithm, and parameter level. The combination of both of these trends requires the emergence of a powerful, flexible processing solution that can be rapidly reconfigured and that consumes a small amount of power. For the handheld domain, any processing solution should occupy as small of a form factor as possible. The coupling of the Layered Radio Architecture and CCMs appear to provide a solution suitable for the handheld domain. However, its performance is highly sensitive to the choice of application. This implies that in order to achieve the full potential of the CCM approach, the Stallion, or some other CCM, should be refined or designed with wireless applications in mind. The most commonly performed operations should be identified and components should be included in the CCM to maximize the performance of these operations.

This thesis also demonstrated the power of applying object oriented programming approaches to the simulation of complex processors. The complex interactions of Stallion were readily modeled using objects as well as the complex operations of the Configuration Layer's operations. The flexibility that object oriented simulation provides also permitted rapid alterations to the initial implementation through the addition of extra data ports and the inclusion of power operations. The approaches used in development of the simulator could be applied to simulate any arbitrary collection of processing cores.

6.1 Suitability of Layered Radio Architecture for Software Radio Applications

This thesis showed that the Layered Radio Architecture was a viable approach to the creation of a software radio. The Layered Radio Architecture provides flexibility at the waveform level and is not specific to any specific waveform. It also allows flexibility at the module level by permitting configurations to be paged into and out of the physical layer through hardware paging. The Single User Adaptive Receiver and WCDMA implementations demonstrate that the Layered Radio Architecture can be used to make adjustments to parameters of an algorithm. Through these implementations, it is demonstrated that the Layered Radio Architecture achieves the levels of flexibility needed for an effective software radio solution. These implementations also show that the Layered Radio Architecture is capable of providing sufficient processing power for the support of emerging wireless standards within a software radio framework.

This thesis also showed that stream based processing has both benefits and drawbacks. Its primary benefit is a reduction in the complexity of the circuits needed to support programming and a reduction in the number of paths that must be maintained. However, stream based processing necessarily incurs additional overhead setting up processing elements for operation. This is particularly noticeable when an implementation necessitates numerous reprogrammings as seen in the Single User Adaptive Receiver implementation.

It should be noted that stream based processing, in general, does not require the traditional separation of program and data memory used in DSP applications as there can never be a memory contention between programming and data for a particular stream. This fact could be used to further simplify the implementation of the configuration layer. As the configuration layer's instantiation was designed with Stallion in mind, the configuration layer specifically uses hardware paging which is counter to the streaming concept. To be truer to the stream based approach, each stream should be capable of operating independently, effectively with its own configuration layer. However, this would introduce significant additional complexity to the implementation. The configuration layer may benefit from implementation on a microprocessor. This should limit the amount of wasted memory seen in these implementations.

6.2 Evaluation of Potential of CCM Technology

The implementations of this thesis demonstrated the sensitivity of the relative performance of CCMs to the choice of application. The WCDMA receiver implementation, in particular, highlighted the following critical characteristics of CCM implementations:

- The potential for performing numerous calculations in parallel is an attractive feature of CCMs
- The utilization of a CCM is “chunky” - small parameter variations can have dramatically different effects depending on the parameter and configuration
- CCMs appear to have the potential for achieving low power consumption consuming half as much power for the same implementation as the C6201 with a higher supply voltage.
- Even with increased I/O the efficient utilization of Stallion’s resources remains a problem for certain applications.

However, because a CCM has significantly fewer, though larger, functional units than a FPGA contains CLBs, the interconnection mesh for a CCM is much less intrusive than for a FPGA. Additionally because each CCM includes dedicated circuitry for arithmetic operations, a CCM can theoretically achieve a higher computational density, computational power, and power efficiency than a FPGA as more optimized silicon is actually used for computations. Also the coarser granularity gives a CCM intrinsic programming advantages over a FPGA. Instead of requiring several Mb of information for programming, a CCM can be programmed to perform a similar operation in only a few kB [13]. Primarily due to this massive reduction in required programming bits, even a slowly clocked CCM can be completely reprogrammed in a relatively short period of time. For instance, the Stallion CCM, which is clocked at only 50 MHz, can be completely reprogrammed in a few microseconds [5]. While significantly better than a FPGA, a CCM’s reconfiguration time is still slower than a DSP or a GPP.

Consider the six parameters of the handheld domain’s solution space: computation power, computational density, power efficiency, inverse reconfiguration time, application flexibility, and hardware flexibility. As shown in the WCDMA implementation, a CCM achieves superior computation power, computational density, and power efficiency as compared to a top of the line DSP. The reconfiguration time for a CCM is longer than that of a DSP or a GPP. However, a CCM's reconfiguration time is significantly faster than that of a FPGA and, as shown in the implementations in this thesis, suitable for multiple reconfigurations in a single real-time application. The implementations also demonstrated that a CCM has complete application flexibility. Also a CCM has significant hardware flexibility, more than a DSP or GPP, but less than that of a FPGA due to the word level reconfigurations of a CCM as opposed to the bit level reconfigurations of a FPGA. From this comparative analysis, the theoretical solution space for CCMs, shown in Figure 6.1, comes closer to matching the ideal solution space for the handheld domain than any of the other technologies considered in this thesis.

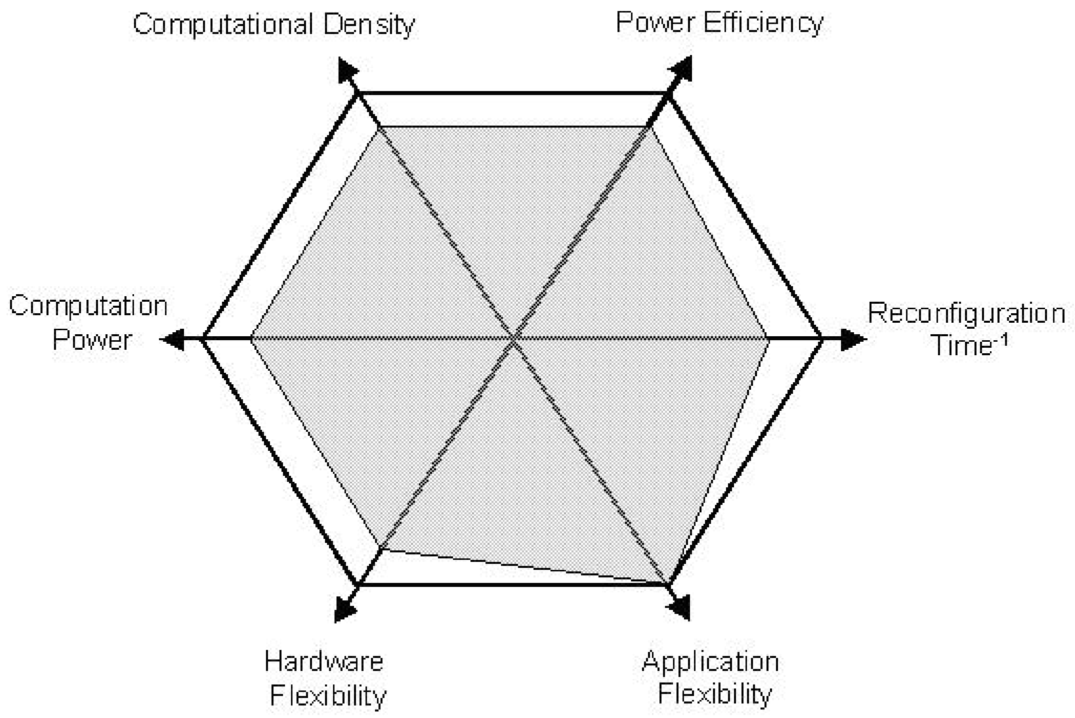


Figure 6.1 Theoretical Solution Space for CCMs

6.3 Suggestions for Improving Stallion

This section lists a number of suggestions for improving Stallion, particularly in terms of resource utilization, by addressing issues specific to the functional units and multipliers as well as general architectural issues.

6.3.1 Specific Suggestions for Improving FUs and Multipliers

Signed multiplication would eliminate many of the headaches associated with Stallion implementations such as the Single User Adaptive Receiver and would also greatly reduce functional unit utilization which can either be used to reduce the number of or to free up resources.

Greatly increase the number of multipliers. This could be accomplished by replacing two functional units with multipliers at the top of the mesh. Based on the implementations made as part of this thesis, if signed multiplication were included, functional unit utilization would be exceedingly small (less than half), but multiplier utilization would remain very high. This implies that the addition of more multipliers would simplify implementations. Additionally, more multipliers would provide the additional resources required for the precise scaling needed for fixed-point filter implementations. In this project, scaling was implemented solely through the use of shifters.

Implement multipliers as multiply-and-accumulate (MAC) cores. This would effectively halve the number of functional units that are required for this relatively common operation. Of course a full implementation would require the MAC to be programmable, but this should be specifiable in a single packet as only the source for resetting the accumulator would need to be programmed. These accumulators should also perform saturation addition to limit the deleterious effects of overflow.

Eliminate flag1 and flag2 inputs that point to bit fields other than <16:15> of the input registers. Proper scaling design coupled with a signed multiplication should eliminate the need for these flags and free up programming bits for other operations.

Latch the outputs of the Flag 1 and Flag 2. Currently the flags can propagate to any point in the mesh in a single cycle over the local busses. This means that it is possible to create implementations with extremely long propagation paths for which it will be difficult to ensure proper performance in the final

Allow the Left Input Mux to be dynamically determined. This would allow a single functional unit to implement a counter which is a commonly used component in communication and thus allowing more efficient use of resources

Support right shifts by more than one position. This is a logical extension and for shifts up to four should not impact the time required to evaluate the operations of a FU.

The direction of data flow along skip buses should be able to turn left and right. This would also make better use of processing resources.

6.3.2 General Architectural Suggestions for Improvement

Specifically, Stallion is not a complete processing solution in itself as a complete system cannot fit on a single chip (which it was never intended to support). Since Stallion must be reconfigured multiple times to support a waveform implementation, the following issues arise:

- input bandwidth is a performance limiter
- external memory is required to store initial data, to support intermediate results and to store programming packets
- an external processor is required to format the streams for Stallion

In reality, this limitation is not specific to Stallion and is rather a characteristic of any pure CCM implementation. Thus these should not be viewed as shortcomings of the Stallion design.

To address the input bandwidth issue, the number of data ports could be increased as was done in the WCDMA implementation. While this would be relatively simple to lay out, this could create significant packaging issues as Stallion already requires 123 input pins just for the data port inputs and three stall lines. Doubling the number of data ports as done in the WDMA implementation, would thus necessitate 246 input lines.

Stallion could be redesigned to directly support external memory as suggested in [13] to further speed up configuration, simplify the use of intermediate results, and to effectively increase I/O bandwidth. However, if Stallion were to be redesigned to support external memory, this process may limit the flexibility of Stallion and greatly complicate the chip, potentially negating the beneficial aspects of Stallion (high computational density, parallelism and deep pipelines). This might partially address the input bandwidth issues, but Stallion would still need an external processor to successfully operate. In reality input bandwidth would still be a performance limiter if Stallion had access to external memory as packaging might again limit the number of connections that a chip could realistically use.

However, neither approach handles the issues associated with setting up nor controlling the streams for Stallion which in the Layered Radio Architecture is the responsibility of the configuration layer. A solution that addresses all of these issues is to add a small processor, perhaps a modified ARM processor, onto the chip and use Stallion as a reconfigurable coprocessor. Thus packaging would not be an issue as nearly any number of traces can be laid out between the processor and Stallion. Thus input bandwidth could be addressed by adding additional data ports. The processor would handle the setting up and controlling of data streams and could effectively handle the higher layer issues for which Stallion is not well suited (decisions to significantly alter performance). The processor's internal cache and external memory lines can be used for storing intermediate variables as in a DSP. The inclusion of dedicated DMA support for each of the CCM streams would free up the DSP core for other activities. Also the combination of a DSP and a CCM will allow each technology to perform the tasks that it does best.

As a coprocessor, the CCM will be very concerned with creating as small a footprint as possible. This can be done by first implementing the specific suggestions of Section 6.3.1. With this implemented, the gross resource utilization would be equal to the adjusted resource utilization. From the two implementations of this thesis, it was seen that no configuration had a more than 50% adjusted utilization. This suggests that only one mesh is required. Without the need to support two meshes, and with the integration of the CCM as a coprocessor, the crossbar can be eliminated, greatly reducing the footprint of the CCM.

For the mesh, it is specifically suggested that it take the form shown in Figure 6.2. This addresses both the need for additional multipliers as well as maintaining a minimum pipeline depth of three. Notice that the MACs should be programmable and also should be bypassable, perhaps by a skipbus.

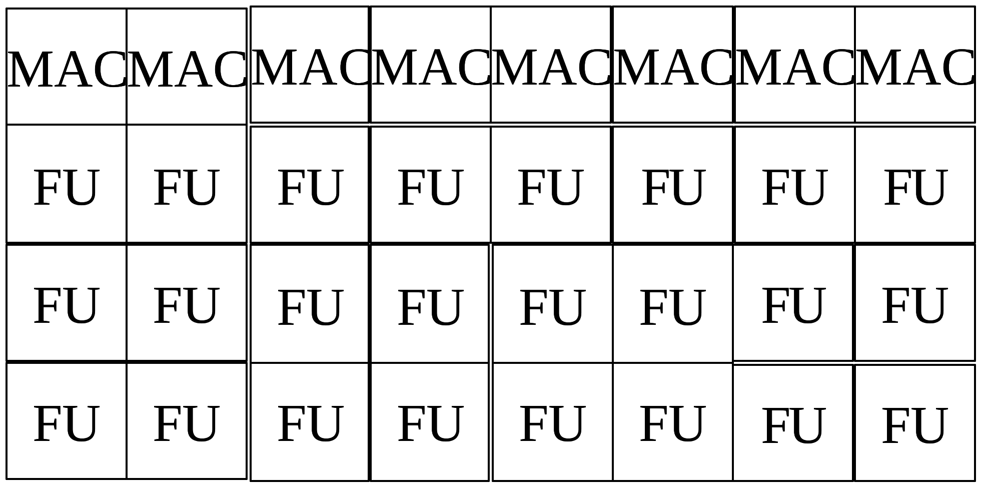


Figure 6.2 Suggested Mesh Arrangement

Assuming stream processing is still used, unique addresses should be used for each component in the mesh. There are seven bits allocated for addressing, thus 128 unique addresses can be configured while there are only 60+22+6 = 88 components that take programming. This would simplify the handling of streams that cross over between mesh halves.

6.4 Directions for Future Work

The CCM concept should be further refined. As the CCM architecture has shown great sensitivity to the waveform being implemented on it, it may be advisable to design specific CCMs with suites of target applications in mind. For example, in future generations of CCMs intended to support 3G applications, it may be advisable to also include specialized cores for the generation of sequences, whether spreading codes, scrambling sequences. It may also be advisable to include a handful of cores that support common error correcting functions, such as support for Galois Field arithmetic.

However, any specialized CCM should first examine the type and number of operations that its target waveforms require. From this examination, the most beneficial reprogrammable cores can be identified. After these cores have been identified, the interdependencies between these cores should be examined. These interdependencies can be used to determine the number and type of connections that should be established between these cores. From this an optimal connection scheme can be established.

A final consideration for the use of CCM as a processing element in a software radio is its development system. While the graphical development system developed as part of this thesis served the purposes of this thesis, it requires the developer to have extensive knowledge of Stallion. A more palatable approach would abstract away the details of the chip so that C and or assembly like instructions can be used to develop applications. This is a particularly challenging task as the required compiler would be for a chip that's never had a compiler written for it. Plus the number of simultaneous operations that the chip supports makes this operation more difficult. As this thesis also shows, the maximal utilization of a CCM's resources is difficult to achieve even when. Typically with compilers, some sacrifices are necessarily made due to the abstraction process. As the compiler matures, this reduction in utilization will lessen, but any initial compiler can be expected to achieve very low utilization.

This process may be simplified, somewhat, by the combination of a DSP and a CCM on a single chip where the CCM is treated as a coprocessor. All operations performed on the DSP could be compiled as normal. Operations performed on the CCM could be treated as specialized assembly level instructions with the typical latencies, delays sources and destinations. The initial CCM instruction set could be expanded later by mapping new operations onto the CCM. Thus the CCM coprocessor would support a redefinable and extendable instruction set adding new flexibility to the DSP concept. The creation of new instructions could be performed using hand mapping or through the application of genetic algorithms which has been used in the past for similar architectures [26]. Note it may be beneficial to develop a coprocessor scheduler in order to maximize the use of this coprocessor when instructions do not occupy all of the available resources.

References

[1] Mitola, III, Joseph. Software Radio Architecture: *Object Oriented Approaches to Wireless Systems Engineering*. John Wiley and Sons, 2000.

[2] Software Defined Radio Forum www.sdrforum.org

[3] Reed, Jeffrey H. *Software Radio: A Modern Approach to Radio Engineering*. Prentice Hall PTR, 2002.

[4] Software Communications Architecture v 2.2. www.jtrs.saalt.mil

[5] Srikanteswara, Srikathyayani, James Neel, Dr Jeffrey H. Reed, Dr. Peter M. Athanas "Soft Radio Implementations for 3G and Future High Data Rate Systems" Globecom 2001.

[6] Reed, Jeffrey H., Srikathyayani Srikanteswara, Max Robert, and Jody Neel "Critical Choices for Designing Software Radios." Tutorial. MPRG Symposium 2002.

[7] Berkley Design Technology Inc. www.bdti.com

[8] Xilinx web pages *www.xilinx.com*

[9] Analog Devices web pages www.analog.com

[10] Texas Instruments web pages www.ti.com

[11] Virtual Computer Corporation www.vcc.com/fpga.html

[12] Hauck, Scott. "The Roles of FPGA's in Reprogrammable Systems" *Proceedings of the IEEE*, vol. 86, April 1998.

[13] Srikanteswara, Srikathyayani. "Design and Implementation of a Soft Radio Architecture for Reconfigurable Platforms," Ph.D. Dissertation 2000 Virginia Tech.

[14] Ray Bittner Jr., "Wormhole Run-Time Reconfiguration: Conceptualization and VLSI Design of a High Performance Computing System", Ph.D. Dissertation, Virginia Polytechnic Institute and State University, 1996.

[15] Soni, Maneesh. "VLSI Implementation of a Wormhole Run-time Reconfigurable Processor." MS Thesis, Virginia Polytechnic Institute and State University, 2001.

[16] Tsuang-Hen Yang, "A Stream-Based In-Line Allocatable Multiplier for Configurable Computing", MS Thesis, Virginia Polytechnic Institute and State University, 1997.

[17] Cherbaka, Mark F. "Verification and Configuration of a Run-Time Reconfigurable Custom Computing Integrated Circuit for DSP Applications," Master's Thesis, Virginia Tech, Department of Electrical and Computer Engineering, Blacksburg, Virginia, July 1996.

[18] Object Management Group Document formal/2001-09-67, "OMG Unified Modeling Language Specification Version 1.4" www.omg.org.

[19] Hosemann, Michael. "Investigation of Synchronization Techniques for Direct-Sequence Spread Spectrum Signals and Implementation on the Layered Soft Radio

Architecture Using Reconfigurable Hardware," Internal Report, Mobile and Portable Radio Research Group. Department of Electrical and Computer Engineering, Virginia Tech, 1999.

[20] TMS320C6000 CPU and Instruction Reference Set SPRU189F, October 2000.

[21] Third Generation Project Technical Specification Group Radio Access Network Working Group 1, "Physical Channels and Mapping of Transport Channels onto Physical Channels (FDD)," TS 25.211 V2.2.1 (1999-08).

[22] Chong, Peter. WCDMA Physical Layer (Chapter6) http://www.comlab.hut.fi/opetus/238/lecture6_ch6.pdf

[23] UC Berkeley Web Site, http://infopad.ees.berkley.edu/infopad-ftp/papers/1994/cmos_cdma_radio.wireless94.

[24] Castille, Kyle. TMS320C62x / C67x Power Consumption Summary Application Report SPRA486C – July 2002.

[25] TMS320C6201 Data Sheet SPRS051G, Nove mber 2000.

[26] Kahne, Brian. " A Genetic Algorithm-Based Place-and-Route Compiler For A Run-time Reconfigurable Computing System," Master's Thesis, Virginia Tech, Department of Electrical and Computer Engineering, Blacksburg, Virginia, May 1997.

Vita

James “Jody” O’Daniell Neel received his Bachelor of Science from Virginia Tech in 1999. His research interests include wireless communications, software radio, DSP implementation of communications algorithms, and game theory. He has enrolled in Virginia Tech to pursue a Ph.D. and is pursuing a dissertation on forming a methodology for applying game theory to the analysis and management of adaptive wireless networks.