Multi-channel Hardware/Software Codesign on a Software Radio Platform

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Jason M. Bales
Bachelor of Science
George Mason University, 2004

Director: Dr. David D. Hwang, Professor
Department of Electrical and Computer Engineering

Fall Semester 2008
George Mason University
Fairfax, VA

Dedication

I dedicate this thesis to my lovely Rheanna who has stood by waiting patiently while raising our three children.

Acknowledgments

I would first like to thank the faculty in the Electrical and Engineering department at George Mason University. Notably, Dr. Hwang for guiding me through the refinement of this research, helping my mind solidify the concepts and innovation behind it all. Dr. Gaj for giving me opportunities to share with other students the experiences I have gained throughout my time at George Mason University and in industry. Also, Dr. Hintz for providing examples and teaching me what it means to be a true engineer.

I would also like to thank my family for being supportive of the time investment required to complete this work, and all their encouragement along the way. Finally, my God for replenishing the spiritual and mental strength that drained daily from me while accomplishing this feat, and the understanding that science and faith can coexist.

Table of Contents

List of Tables

List of Figures

Abstract

MULTI-CHANNEL HARDWARE/SOFTWARE CODESIGN ON A SOFTWARE RADIO PLATFORM

Jason M. Bales, MS

George Mason University, 2008

Thesis Director: Dr. David D. Hwang

Software Defined Radios (SDRs) are growing in popularity in our increasingly multi-standard wireless world. As a result, platforms have been developed to provide powerful and flexible SDR implementations. These platforms tend towards a heterogeneous mix of software (SW) and hardware (HW) components, requiring HW/SW codesign to be an integral part of SDR development. Previous work in the area of SDR codesign has focused on finding optimal HW/SW partitions of a physical layer design that minimize the latency of a single channel. The purpose of this research is to provide a new focus for SDR codesign, that of multi-channel HW/SW codesign. These codesign efforts focus on maximizing the number of simultaneous transmission channels that can run on a codesign SDR platform. A method for deriving models that represent the HW/SW components and the interface between them is presented. The results of this method are general equations describing the HW/SW timing constraints for a multi-channel implementation. An application of the method and models is provided in a Codesign Decision Engine (CDE) that takes as inputs characteristics of a case-study platform, and outputs a predicted number for transmission channel capacity for both a baseline and improved impelmenation. The method and models are verified by actual application performance on the platform under test.

Chapter 1: Introduction

1.1 The Maximum channel Optimal Latency Approach

Past and current research on hardware-software codesign has focused design criteria on minimizing channel latency, design time for codesign implementation, proper tool simulation environments, and software based partitioning strategies. While decreasing the required processing time for generating a channel stream (i.e. minimizing latency) is one approach at a codesign solution, it is not the only optimal solution. A channel stream for a radio specification will have a predetermined maximum symbol rate. Once the channel processing latency required to meet this symbol rate has been obtained, further speed optimizations increase the complexity of the codesign approach while providing little added benefit to the solution.

An optimal solution is one that meets specified criteria. In the case of a transmission communications channel, the optimal channel implementation is one that provides an output stream that satisfies the stated symbol rate. The processing that occurs in the implementation must only have a small enough latency to achieve the output specified rate under a set of operating conditions, given by the operators of the system. It is the role of the codesigner to determine a hardware and software partitioning solution that achieves this optimal solution.

Some metrics that may be used to determine codesign partitioning are flexibility and maintainability of design, and also cost and schedule required for implementation. While a specific design will be affected by the use of these metrics, this work focuses on key resource utilization in the hardware and software components that make up the codesign environment.

This work is novel because we developed a codesign approach with an emphasis on a

new criterion: maximum number of simultaneous channels. The focus for this multi-channel codesign is to provide an analysis method that derives resource estimating models for both software and hardware components on a codesigned platform that correctly predicts the channel capacity in a given configuration.

1.2 The Method

While the models that will be arrived upon in this research are tailored to a specific target platform, the method used in the codesign analysis can be generally applied to many different codesign platforms and architectures. The goal of this research is to propose a method, and show by application the accuracy and benefits, in terms of capacity awareness, of implementing this method on a codesign platform. A very general overview of the method is

1. Define pertinent resources of both the software and hardware components on the target platform.
2. Model usage of the resources in the software target execution environment
3. Model usage of the resources in the interface between software and hardware
4. Model usage of the resources in the hardware component
5. Use the models to predict existing channel capacity
6. Identify candidate stages of the platform that can be shifted from software to hardware

Once candidate stages have been identified, work can begin to move them from software to hardware, and the prediction can be repeated to show forth any changes in capacity achieved.

1.3 Outline of Thesis

After the introduction we present Chapter 2 on related work in the area of codesign describing methods, ideology, and examples of existing research. Chapter 3 covers background on the communications channel, software defined radio (SDR), and the target platform used in this work. Chapters 4 and 5 provide an analysis of both the software, hardware and interface portions of the codesign environment. Chapter 6 describes the codesign engine, or the application software that implements the models of this work, followed by a specific and generic case study used to prove the analysis method. Finally, a conclusion is provided in Chapter 7 with ideas for future work.

Chapter 2: Related Work in the Area of HW/SW Codesign

Hardware Software codesign is not a new approach in industry. Many engineers have recognized the need for designing for a platform that contains both hardware and software components for some time [3][18]. As industry platforms become more flexible, allowing for field upgradable components (i.e. firmware updates) these platforms are mixing hardware components such as application specific integrated circuits (ASICs) and/or field programmable gate arrays (FPGAs) with control software components such as digital signal processors (DSPs) and Microcontrollers. Therefore the need for an optimized design approach that takes both of these components into account becomes stronger.

2.1 Brief History of Codesign Methods

To begin the study of hardware software codesign, we first take a brief look at some past and current methodologies for codesign, then describe some ideologies that have been followed while following the codesign approach, and then describe several existing codesign frameworks that have been used in academia for implementing codesign methods and ideologies. The following section describes some of our thoughts on potential for improvement upon the material described in this section.

2.1.1 Past and Current Methodology

A Generic Codesign Methodology

In 1993 Asawaree Kalavade and Edward Lee presented a paper titled "A Hardware-Software Codesign Methodology for DSP Applications." This paper presented a generic codesign methodology, reproduced here briefly in Figure 2.1. It defines codesign as a task "to produce

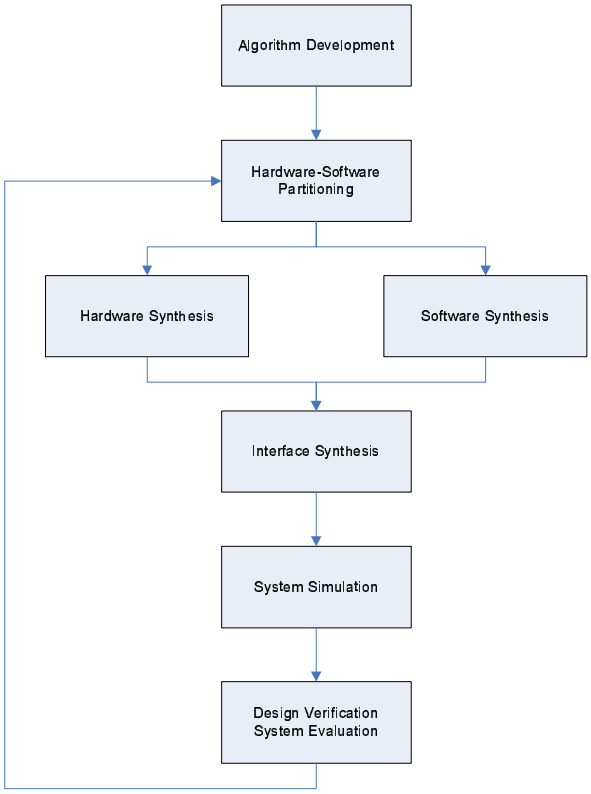

Figure 2.1: A generic codesign methodology, presented in [3]

an optimal hardware-software design that meets the given specifications, within a set of design constraints" [3]. It begins with the algorithm development being designed at a high (i.e. abstracted) level of simulation, done without a focus for the implementation on the platform. Once the algorithm has been established, it is then partitioned into stages that are distributed among the hardware and software components. The stages are then synthesized, or compiled, for the respective component that it will be executed on. Afterwards the synthesized blocks are brought back together for simulation to verify the original design constraints are met, and then the entire design is evaluated for performance. The entire process is iterative, with the number of iterations based on the results after evaluation. If a design's performance is lacking, the engineers would start over at the partitioning stage, re-evaluating the algorithm's stage distribution, or even the algorithm itself [3].

Simulation Enviornment

In order to perform the codesign simulation block shown in 2.1, an environment must exist that can interpret the results from the synthesis stage, bringing them together from both components into a single solution that can be properly simulated for a target platform. A codesign simulation environment is just that: simulation. It is strongly dependent on the models used to represent the component being simulated. Codesign platforms have been known to have strict physical constraints on their embedded components, so some effort has been made to develop the models of the components working together on a codesign platform [10][15][16][17]. As industry matures, models become stronger and simulations become more accurate in revealing the true behavior of the target platforms. However, issues still exist with simulation results not matching true behavior. Because of this, a simulation can only provide so much verification that a design truly performs according to provided design criteria.

With the flexible programmablility of todays codesign platforms that utilize processors and FPGAs, testing can occur on the actual platform itself [22]. Therefore, a different approach would be to have the codesign analysis occur before the algorithm partitioning is accomplished. In this way, the codesign analysis can actually provide information about where to partition the stages of the algorithm before synthesis.

2.1.2 Past and Current Ideology

Reconfigurable Hardware

Many codesign platforms involve one or more FPGAs. These FPGAs have an advantage over ASICs by allowing one to reconfigure them (i.e. load a different build into the FPGA) in real-time. Efforts have been directed towards using this reconfigurable property to create custom datapaths in heterogeneous codesign systems [20], embedded tools to assist in the profiling of designs on target platforms [22], and custom reconfigurable coprocessors [25].

While most FPGAs must be reconfigured as a whole, some research has progressed with

devices that allow portions of their architecture to be reconfigured independently of each other, providing essentially many small hardware components available for executing tasks [12][21]. These works typically focus on homogeneous systems rather than heterogeneous[1] ones that involve both software and hardware components. As embedded systems continues to grow in industry, more heterogeneous systems are coming into play, requiring codesign efforts to focus on both hardware and software design.

Parallelism versus Sequentialism

One approach that is common across hardware-software codesign methods is that of treating the hardware component as a collection of smaller execution devices, or reconfigurable coprocesssors [1][8][23][25]. The hardware can be reconfigured to run specific tasks, and then driven and scheduled by the main software component (i.e. a DSP or microcontroller) [19][1]. While the parallel nature of the hardware can still be kept in the implementation of the task execution, the tasks still require instigation by sequential code running on the software component.

Interfaces between HW/SW

As platforms have begun to incorporate more of a heterogeneous component mix, the communication that occurs between these components becomes an area of concern [24][10]. To ensure that all the components can "play-well" together, they must be able to exchange information in a timely fashion. This brings to light the idea of a new type of design approach discussed in [9], which emphasises that both the software and hardware designers should work together throughout their respective design flows to ensure that the interface between the software and hardware components is modeled and designed against appropriately.

[1]Heterogeneous when used in terms of a codesign platform refers to a platform that incorporates different types of components, such as a software based CPU and a hardware based FPGA.

2.1.3 Some Existing Codesign Frameworks

Ptolemy

Ptolemy[4] provides a framework where all the models of a codesign platform can be brought together for complete system analysis. A major emphasis of Ptolemy is on the method used for designing the codesign solution using these models. Therefore, Ptolemy stresses that the models should appropriately portray the behavior of the hardware-software components, in order to correctly describe them during the design phase. If a model does not properly represent a component, the accuracy of the codesign method will be penalized [4]. Ptolemy relies heavily on the models of the components being provided as inputs into the framework. In other words, before utilizing the Ptolemy framework completely, one must have already obtained models that appropriately represent the components in the codesign platform.

POLIS

POLIS is another framework for hardware-software codesign that centers around modeling the codesign platform as a finite-state-machine FSM with each element of the FSM being a component of the codesign system [5]. POLIS requires tools to synthesize the hardware and software portions of the design based on the target software and hardware components. It also requires formulas and benchmarks for the target component in order to properly feed a timing estimator which aids in the codesign analysis. This estimator obtains estimates of software procedure execution time by appending C code intrusively inside each procedure modeled in the POLIS FSM. This requires the POLIS framework to have an understanding of the target component software environment (i.e. C programming semantics). While most embedded software components utilize an ANSI C language variant, this stipulates a limitation that software components utilized in the POLIS framework support the C code language that will be inserted by the POLIS framework.

Also, each procedure, or task, of the software component is also mapped to a simple real-time Operating System (OS) that the POLIS framework uses for modeling system interface

interaction as well [5]. This provides a challenge for codesign platforms that utilize industry components that already provide a real-time OS[2].

The Delft Workbench: A quantitative prediction model

An approach to moving basic blocks of software into hardware, and the estimation of area resource requirements that will be required for these blocks is described in [6]. In this work, a linear regression model is used that takes as input a collection of software functions, with associated software complexity metrics (SCM) for each function. The SCMs quantify complexity of the software function, or basic block, with a score. The model then uses the score for each SCM describing a single function to determine area resource requirements for implementing that function in hardware (see Figure 2.2). All of the input code is written in C, which is then complied with the DWARV C-to-VHDL compiler to produce VHDL source for the hardware portion. The VHDL is then compiled and synthesized using industry tools. The research deals with the reconfigurable MOLEN platform described in [7], based on the Xilinx Virtex-II Pro, as the target platform to perform the statistical experiments.

The results from [6] show as high as 92% prediction accuracy for Flip-Flops required, about 63% accuracy for LUTs required, and a low prediction accuracy of around 5% for multipliers, by using the linear regression SCM model. While not perfect, these results do show that software SCMs can be used to correlate to hardware area requirements for moving functions from software to hardware. The focus of [6] is not on multiple instantiations of the functional basic blocks, but instead on the estimation of each functional block's hardware requirements individually.

PeaCE: A hardware-software codesign approach

The PeaCE codesign environment described in [8] takes the approach of specifying a heterogeneous system's behavior from a top level, such as with dataflow and FSM tools, an

[2]Such as the case with Texas Instruments' line of TMS320 DSP processors which contain a multi-threaded OS called DSP/BIOS.

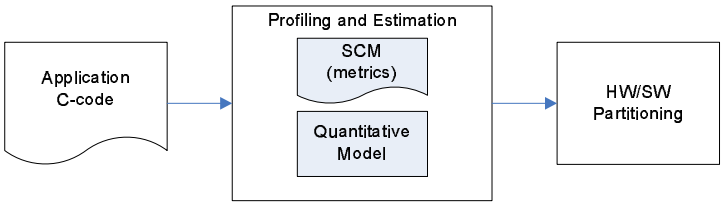

Figure 2.2: Simplified Delft Workbench approach. For a more thorough flow diagram see [6].

approach that has been taken in previous work as well [18]. This top level system specification is provided as input to a design space exploration (DSE) stage. The DSE takes as inputs the top level system specification, a provided library of processing elements (PE)[3], and a profile table of the PEs (such as cycle requirements, memory access count, etc.). The research assumes all the provided PEs are connected through a bus interface with the protocol of the bus unspecified until a later stage of the design process. It then makes decisions about PE allocation of specific tasks, or portions, of the system behavior. Once these PEs are chosen, a HW/SW cosimulation is performed to provide a memory trace depicting the exchange of data among the PEs required to implement the system behavior provided. Once the memory trace has been recorded, candidate communication interfaces are chosen and the DSE iterates again with this interface included[4]. Then cosimulation occurs again, providing a final hardware/software partitioned solution.

While this is a thorough approach, it currently focuses on minimizing the schedule length provided in the system behavior, rather than maximizing the capacity of the system[5]. The technique also assumes that once a system task has been assigned to a PE, all the resources from the PE will be consumed. The research recognizes that not all of the PE is consumed in reality, such that a chosen PE could actually run multiple portions, or instantiations, of the system specification.

[3]PE is a physical processing component, such as CPU, FPGA, etc.
[4]The interface implementation will also require PEs.
[5]How many instances of a channel versus how fast the channel runs.

Dynamically Constrained HW/SW Partitioning

The work in [11] uses a hybrid genetic algorithm (GA) with a dynamically-weighted fitness function to determine what portions of a software program should be moved to hardware. It focuses on a hardware implementation of the transport triggered architecture (TTA) processor which implements a single *Move* instruction to transport data between functional units (FU) that are instantiated in hardware. The FUs perform the work, and a separate FU is needed for each unique operation that will be carried out in hardware (e.g. multipliers, adders, register files, etc.) [11]. A fitness function is used to pick out interesting functions of the software, and determine if those blocks should be implemented in hardware as a new FU.

This work requires the software program to be profiled before being input into the GA. It also relies on basic blocks that will be implemented in the hardware as FUs to be profiled as well, providing area constraint information by use of synthesis tools on sample FU implementations for the target hardware component (in this case a Xilinx Virtex II). Once the software has been profiled to provide time constraints on the various functions, and sample hardware FU area requirements have been generated, a weighted fitness function can be applied on each software block individually, along with the hardware block that would implement it. The fitness functions is described as such:

$$k = \frac{hwLimit - s}{hwLimit} \tag{2.1}$$

$$f(s,t) = \begin{cases} \frac{\alpha}{t} * \left(k * \frac{\beta}{s} - k + 1\right) & \text{if s} \leq \text{hwLimit} \\ (log(s - hwLimit) + 1)^{-1} & \text{otherwise} \end{cases} \tag{2.2}$$

where α is the time to execute the given software program on a processor, t is the required time for a functional block to execute, β is the size of the hardware available, s is the hardware size of a block (i.e. the area required to generate that functional software block

in hardware), k is the dynamic factor, and *hwLimit* is the maximum hardware size allowed to implement the software block as an FU in hardware.

The first ratio (fractional portion containing the α) corresponds to speedup obtained with this block, while the second ratio corresponds to the hardware size increase. The general idea behind the fitness function is that if the speedup obtained from reducing the time required to execute a functional block (i.e. by moving it to a hardware FU) is relatively larger than the size increase, the fitness score will be high, designating this functional block as a candidate for moving into a hardware FU [11].

While this work makes great strides, in its current state it requires the input program to be provided in a restrictive version of the C language that does not lean well towards true applications, and also focuses on the speedup of an individual block rather than an increase in the maximal capacity of the system (as will be discussed later).

Hardware/Software Interface Modeling

Because a heterogeneous system involves an interface that touches both software and hardware components, designing a codesign platform requires proper models of this interface. [10] addresses this by describing an executable model of the hardware software interface using SystemC. The result is a model that can be used in both the software simulation and hardware simulation design scope. While the focus of [10] is not on the partitioning of the system, it brings out a key focus on the communications interface between the heterogeneous components of the codesign platform, and emphasizes the need for a more accurate model of this interface.

2.2 Ideas Behind Our Research

While reviewing the related works, a few points began to stick out that drove the initial idea behind this research. We feel these points are places where further research can occur beyond the described related efforts, and represent a starting point for what will follow in this work. They are listed below.

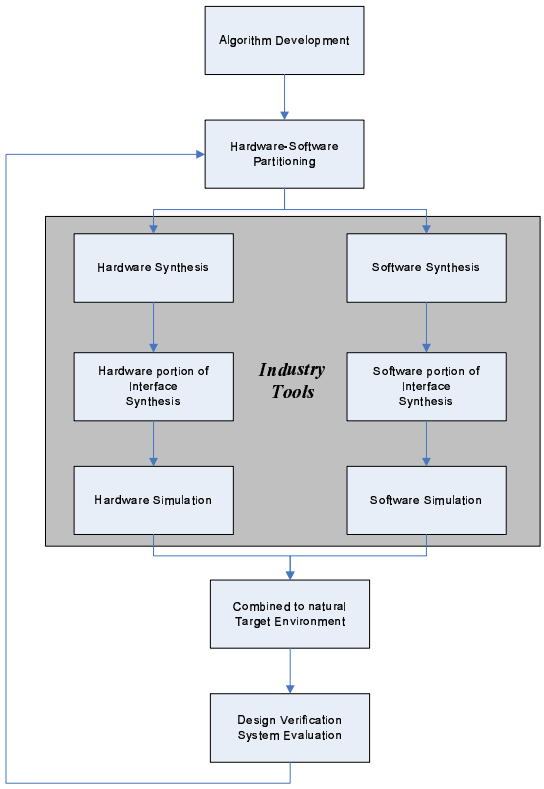

Figure 2.3: A Modified codesign methodology

Codesign analysis should occur before partitioned synthesis.

By performing a thorough codesign analysis before partitioning, the synthesis and simulation can be performed in the hardware and software components' respective natural environments, as portrayed in Figure 2.3. In other words, *if we can trust the results of a codesign analysis before partitioning and synthesizing, then we can treat the separate parts of the codesign individually during synthesis and simulation.*

For example, after a hardware-software codesign platform has been analyzed using a pre-partitioned algorithm, the software developer can use his current industry tools to develop and simulate their portion, while the hardware developer can develop their portion using industry tools as well. These standard tools will have outputs in industry expected formats that can be used across multiple simulation environments. The requirement for a new environment that takes both hardware and software designs and merges them together for simulation analysis is unnecessary. Since these types of environments are still developing

[5][4], and struggle with target platform specific details, it is easier on development time to use standard tools both developers will be accustomed to.

Codesign Analysis can also Benefit Existing Codesign Implementations

While most related works deal with creating new designs, in some situations system designers would like to utilize existing platforms to achieve new capability. In the example of a Software Defined Radio (SDR) platform, a new communications channel may exist that could leverage existing software algorithms and hardware implementations on the SDR, with minimal new design required. In this scenario, with the proper models that characterize the existing SDR, a developer can utilize a codesign method to determine what capacity (i.e. how many simultaneous channels) the existing SDR can provide of the new communications channel with minimal design modifications[6].

Hardware: Maintaining the power of parallelism

Treating the hardware component as a series of coprocessors can increase processing quantities, however, at a possible expense of loss of the parallelism capability of the hardware[7] Creating coprocesssors that perform separate tasks in an FPGA, for example, essentially turns the FPGA into a software component that executes sequentially. *The parallelism provided by hardware, mixed with flexible sequential execution provided by software, together provide the true power in a codesigned platform.*

Interface Inclusion in Codesign

A key block in the generic codesign methodology that is often overlooked is that of the interface (see Figure 2.1). The interface becomes a critical part of the analysis when designing a codesign environment where data is exchanged back and forth between the hardware

[6]This approach is inline with the case study presented in this research. Models of the existing SDR are made, and an analysis is performed giving insight into what design modifications are necessary to achieve a given capacity.

[7]The degree of parallelism capability that is traded using the hardware-becomes-coprocessor approach depends on individual implementations. Not all approaches will lose all parallel capability.

and software components. The difficulties in designing both the software and hardware portions of a heterogeneous codesign platform are emphasized in [9] which describes the evolution of multiprocessor platforms. Interface consideration is a part of codesign that while discussed and given credit for being a key factor [1][2][24][10] has not been addressed fully in most related works. This research includes the interface directly in the derivation of the hardware-software codesign analysis models.

Multichannel Approach: Capacity Increase versus Latency Reduction

One focus in codesign research has been obtaining a hardware-software partition solution that creates the fastest signal path through the platform [1][8]. While this approach has produced designs that can accommodate existing heterogeneous platforms, it can over complicate system designs that have a focus on maximizing the capacity of the system (i.e. multiple signal paths) instead of speeding up a single path. In the case of the SDR, which is addressed in this work, increasing the maximum number of simultaneous channels provides greater benefit to the system than a single faster communications channels. This relies on the fact that once the optimal[8] speed of a channel has been met, there is no need to further reduce the latency of the channel. Specifications call for specific baud rates that once met require no further speedup of the channel. Therefore, the approach taken in this research is an increase in capacity versus reduction in latency.

[8]Optimal does not imply most efficient, but instead that a provided specification has been met, and not necessarily exceeded.

Chapter 3: Background

3.1 The Communications Channel

A radio transmission channel, when viewed as a block box, has a single input/output pair, a bit stream and a sample stream, respectively. The implementation of a single channel can be approached as a series of stages that perform processing on the input bit-stream to produce an encoded and modulated output sample stream. As the HW and SW components that make up these radio platforms increases in complexity, the decision about where to implement the channel stages becomes more complex as well.

The transmission channel implements the physical layer (and sometimes more) of a given specification. Information flows through the channel first as a user provided bitstream which is then conditioned and modulated. Some common modulation schemes for wireless communications are BPSK, QPSK, and $\pi/4$ QPSK. The modulation sequence is then filtered and mixed to an Intermediate Frequency (IF) for distribution to a Digital to Analog Converter (DAC). Once the sequence has been converted to a stream (continuous or burst depending on the specification), it is typically sent through analog stages of upconversion to Radio Frequency (RF) and amplification for transmission across a medium.

The algorithm used in this research to generate the communications channel stream can be broken up into intermediate stages that handle portions of the implementation. Figure 3.1 depicts the arrangement of these stages and a description of each follows.

1. **Bitstream:** This stage is preliminary to the data path. It acquires the desired bitstream of interest that will be modulated, and pushed through the channel. This is where the system user specifies the information to be sent. This stage is associated with architecture overhead on the SW platform for interacting with the input/output

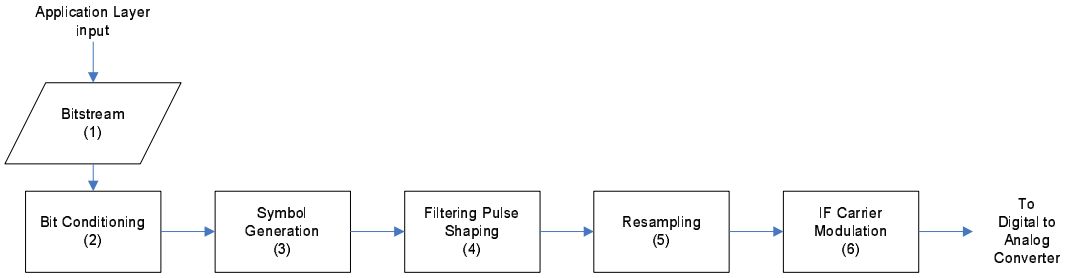

Figure 3.1: Stages of a Transmission Communications Channel

control interface for the user. Its detail will not be addressed in this research, but instead will be represented as a given overhead.

2. **Bit Conditioning:** Any necessary bit manipulation of the input stream is done in this stage. Depending on the specification the channel implements, the input bitstream may need to be encoded and scrambled, producing a conditioned bitstream which will continue through the remaining stages of the channel.

3. **Symbol Generation:** When the bitstream is in its final form (i.e. all necessary specified conditioning is completed), the individual bits are mapped to symbols. The number of bits per symbol is again a function of the specification the channel implements. For example, a QPSK symbol set contains four unique symbol locations in the IQ constellation mapping. The number of bits that can be represented per symbol in this case is

$$\begin{aligned} \#\text{ Bits} &= log_2(n) \\ &= log_2(4) \\ &= 2 \text{ bits} \end{aligned} \tag{3.1}$$

where n is the number of uniquely defined symbols, in this case 4.

4. **Filtering / Pulse Shaping:** This stage provides any pulse shaping required by the specification (i.e. root raised cosine filtering to avoid inter-symbol-interference).

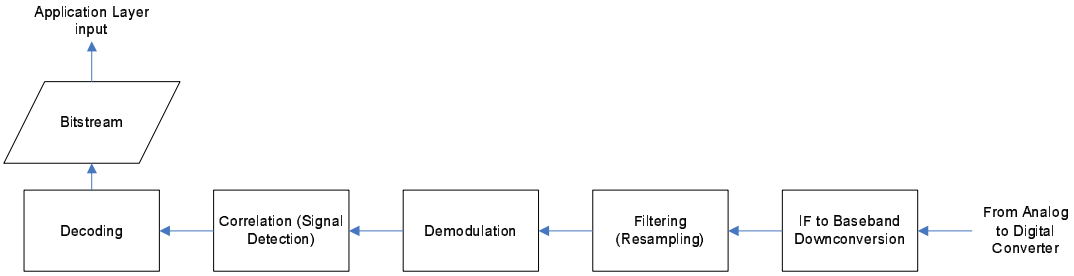

Figure 3.2: Stages of a Receiver Communications Channel

The input to this stage is the *symbol set* while the output is generally viewed as a *sample set* (i.e. a sampled set of the modulation symbols). In some cases, this stage is combined with the Resampling stage to utilize a single filter to perform both stages, thereby minimizing processing.

5. **Resampling:** Any necessary rate conversion to bring the sample set up to a rate commensurate with the carrier wave mixing that will occur in the next step is done in this stage. This stage could very well be split into multiple substages depending on how the resampling process is broken up between the HW/SW in the codesign.

6. **IF Carrier Modulation:** This stage mixes the sample sequence generated from the previous stages with a generated carrier wave sequence. The result is to mix the information containing samples up to a carrier wave frequency for transmission. Many wireless protocols exist in the VHF (30 to 300 MHz) and UHF (300 to 3000 MHz) bands. Because it would require extreme processing power to mix directly to these bands, many implementations first mix to an Intermediate Frequency (IF) while still in the digital domain. The resulting sample sequence is sent to the platform Digital to Analog Converter (DAC) for discrete to continuous time conversion. This IF analog signal is then mixed through analog hardware up to the final Radio Frequency (RF) carrier destination in the VHF/UHF bands.

Like the transmission side of the communications channel, the receiver side performs the reverse order of stages. In actuality the receiver is typically more complicated than

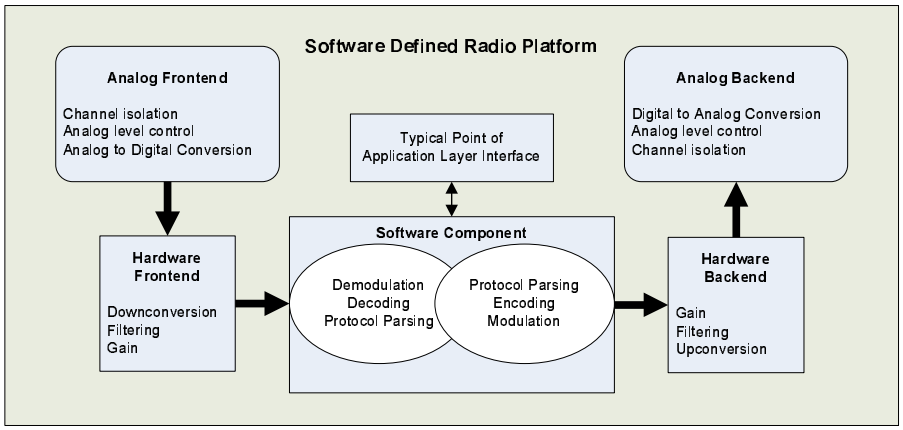

Figure 3.3: An example of a generic software defined radio platform.

the transmitter, since it must perform symbol timing and frequency estimation, as well as equalization depending on the environment of the communications medium used. However, Figure 3.2 shows a high-level view of a receiver based off of the reverse of Figure 3.1.

3.2 Software Defined Radio

3.2.1 A General SDR

Due to the increasing number of utilized wireless specifications, having the capability to move from one specification to another becomes a key factor for flexible radios. Flexible communications platforms must interact with multiple protocols. Working in these multiple protocols requires many signal processing tasks to be performed by the radio[13]. As the number of supported protocols increases, historically, additional hardware was required to implement the different signal processing tasks between the protocols. This resulted in increases to cost, system design complexity, size, and power consumption. The concept behind Software Defined Radio is to move these differing tasks to a generalized software environment, where they can all be implemented with the same resources, leaving the common task (e.g., mixing, gain adjustments, etc.) in hardware components. Put simply by Mitola in [35], "The software radio delivers dynamically defined services through programmable processing capacity..."

Having a programmable software environment on a Software Radio platform allows the radio to switch protocol implementations on the fly. This is a defining characteristic of a Software Radio: the ability to implement new protocols without requiring additional hardware components. Current Software Radio platforms usually follow this hybrid approach, including both software and hardware as depicted in Figure 3.3. A common analog and hardware front-end and/or back-end can be put in place for the processing components that are implemented in every radio (e.g., IF to RF upconversion), while the application specific portions (e.g. demodulation, decoding, protocol bitstream parsing) can be placed in a software component for easier control and switching[1][36].

3.2.2 Past work on SDR

As the flexibility of software processors increases, efforts are being taken to implement more of the physical layer of the communications channel in software to provide as much flexibility as possible for changing specifications [26][30][31][35]. So much so, that some processor architecture designs are carried out with SDRs in mind as a primary application [27][28][29].

However, implementing the physical layer on a sequential based execution component, such as a Digital Signal Processor chip, has begun to show the intensive cycle requirements for some of industry's increasing wireless standards sets [32]. Therefore, SDR platforms have begun to come forward that employ both software and hardware components, utilizing the hardware for the parallel power to increase capacity, while maintaining flexibility in the system through the software [33][34][36][37]. Because of these heterogeneous platforms, codesign becomes a critical part of SDR design.

3.2.3 SDR Codesign

Radio platforms are evolving towards an approach involving a mixture of hardware (HW) and software (SW) components interfaced together to provide programmable radio channels. Due to this mixture of HW/SW on a single platform, studies have been made on

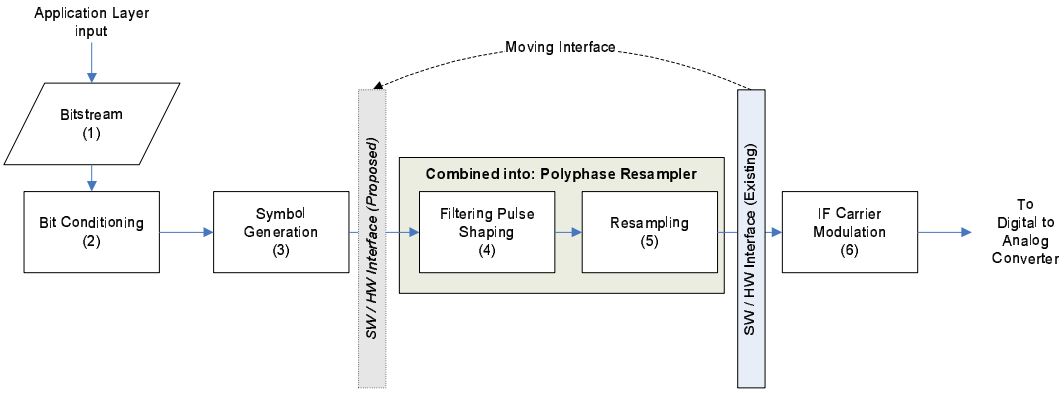

Figure 3.4: Codesign HW/SW interface adjustments.

design approaches focusing on dividing the overall signal processing algorithms between the HW/SW components. In some cases, the programmable portions of the communications channel that the Software Radio implements can be moved to a configurable HW component (e.g. Field Programmable Gate Array) to ease the processing load of the SW component (usually a Digital Signal Processing chip), freeing up cycles that can then be directed towards other tasks. One such example used as a case study in this research, depicted in Figure 3.4, is the implementation of a polyphase resampler in hardware to fulfill the pulse shaping and sample rate conversion stages previously performed in software.

3.3 Target Platform Used in this Research

The target platform used in this research is the ArgonST Gen5 Intelligent Terminator Card (ITC), as shown in Figure 3.5. The ITC is the transmission card that sits at the bottom of a ArgonST Gen5 stack. At its core, these Gen-5 systems are Digital Signal Processing (DSP) and Field Programmable Gate Array (FPGA) based software radios that can digitize up to six simultaneous IF inputs, each with up to 80MHz of contiguous bandwidth. This is done with an Analog to Digital (A/D) card. Once digitized, the data is passed down a stack of up to ten digital receiver cards. These digital receiver cards are designed with powerful and flexible digital signal processor components. Each digital receiver card contains two 1 GHz DSPs (TI C6416) and three Xilinx FPGAs (two Virtex-4 SX55 and one Virtex-4 FX60).

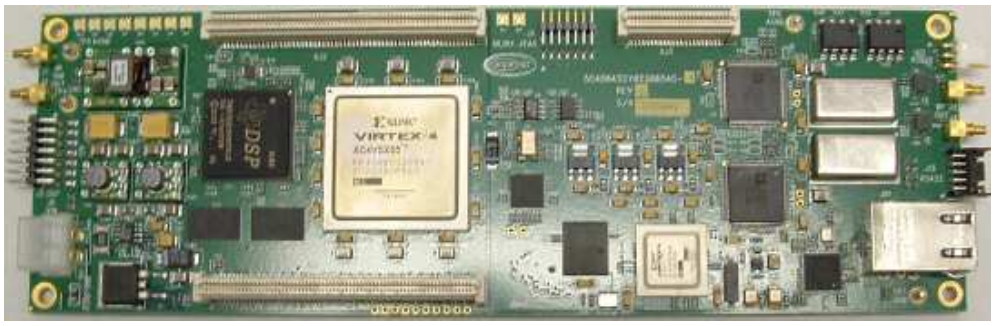

Figure 3.5: ArgonST Gen5 Intelligent Transmitter Card (ITC).

Terminating this stack of cards is the Intelligent Terminator Card (ITC), which contains one 1GHz DSP (TI C6416), one Virtex-4 FPGA (SX55), one Virtex-4 FPGA (FX60), two Digital-to-Analog (D/A) converters (AD9779), two digitally controlled analog attenuators (provides two TX paths), a stand-alone ethernet interface, and RS422/232 interfaces. The ITC can also accept information directly from any of the receiver cards in the stack above for pertinent information regarding transmissions (e.g., transmission information based off of received information). Each receiver card also provides stand-alone gigabit ethernet interface and serial options (RS232) for communicating with the controlling Application. The ArgonST Gen5 cards are actively used in complete SDR systems in industry.

Both the receiver card and ITC are separate complex codesign platforms, involving a mix of both software, hardware, and analog components as well as the interfaces between them. To limit the scope of the research, and to design against an existing application which can provide real-time bench marking of any improvements provided, the ITC was chosen as the target platform for the case study.

ITC: Software Component

A basic block diagram of the ITC is shown in Figure 3.6. The software component is the TI C6416 DSP, configured to run at 1 GHz. This CPU has access to 32 MB of SDRAM that has been populated on the ITC and connected to the CPU through it's EMIF-A interface. While

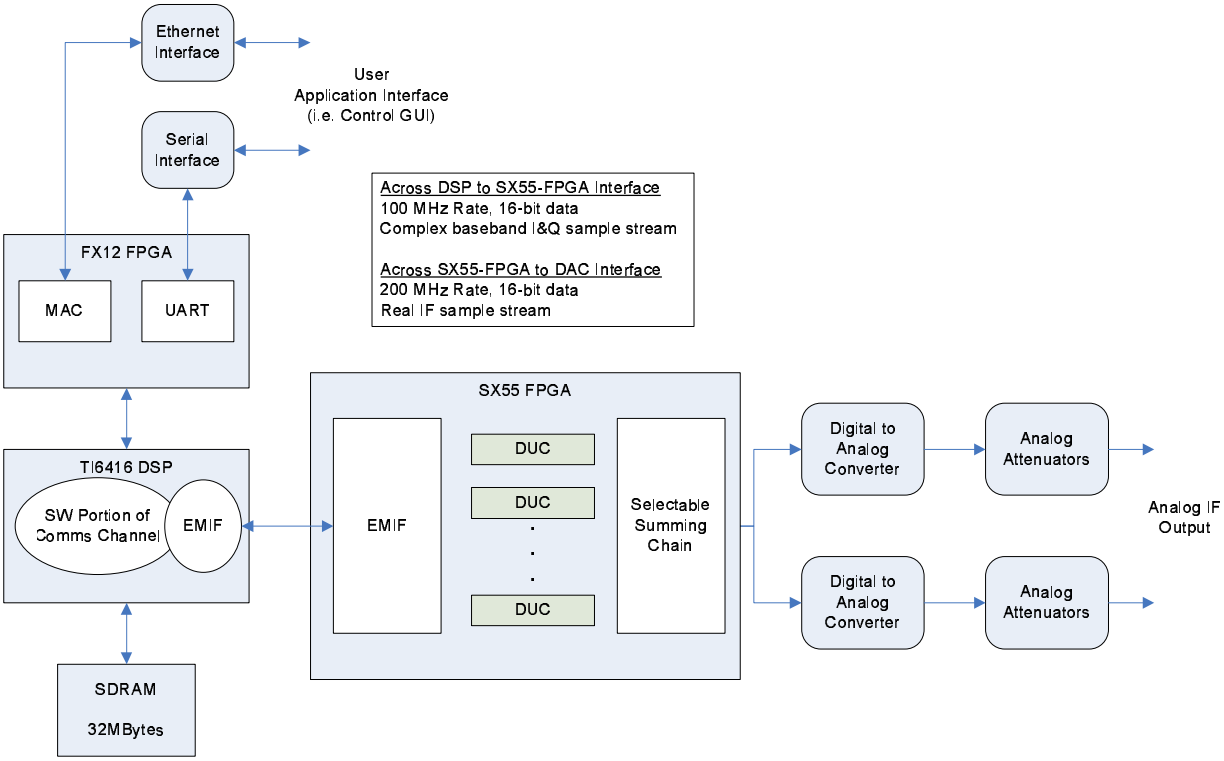

Figure 3.6: Block Diagram of Interconnections of ArgonST ITC.

the ITC has many functions in the Gen5 stack, we will consider mainly the transmission capability it is responsible for. With this in mind, the ITC's main objective is to implement stages 1 through 5 of Figure 3.1. The output of the CPU is a sample stream at baseband that can be sent to the hardware component for further processing through the External Memory Interface (EMIF) that connects the DSP to the FPGA. This interface runs at 100 MHZ and is a 16-bit data bus, with a 32-bit address bus. Any data that moves between the software and hardware components of the ITC must travel through this interface, and therefore it becomes an integral part in the derivation of models pertaining to the platform.

ITC: Harware Component

The main ITC hardware component is a Virtex-4 SX55 FPGA, whose primary responsibility is to complete the communications channel. It does this by providing any remaining upsampling to convert the sample stream from baseband to intermediate frequency (IF)

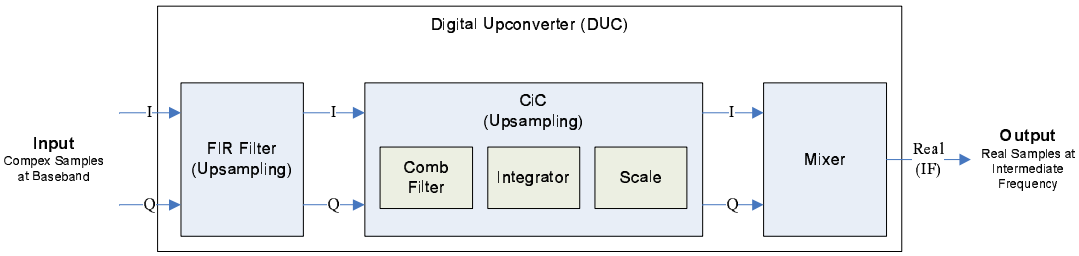

Figure 3.7: A Digital Upconverter (DUC) block diagram

before being sent to backend hardware components for conversion from a digital to analog signal, and from IF to RF. The critical instantiated block inside the FPGA that meets this requirement is the Digital Up Converter (DUC).

Figure 3.7 shows a simplified block diagram of a DUC implementation utilizing a Cascaded Integrator Comb (CiC) method. In this implementation, the complex baseband samples are first sent through a FIR filter that performs initial upsampling to bring the sample stream to an acceptable input rate for the CiC. The CiC then performs additional upsampling and gain scaling to bring the sample stream the rest of the way up from Baseband to IF. The digital complex IF sample stream is then pushed through a mixer that produces a real-valued signal. This real-valued signal can then be forwarded out of the FPGA to a separate onboard DAC in prep for analog conversion to RF.

3.4 Target Wireless Specification

The target wireless specification that will be implemented in the communications channel is Geostationary Earth Orbit Mobile Radio Interface (GMR-1), which is a Mobile Satellite based specification derived from the ground based GSM standard [39]. The details of the layers above the bitstream (i.e. Data Link Layer and above) are unimportant to this research. The Physical Layer (e.g., modulation scheme, symbol rate) however, is important.

GMR-1 is a TDMA based specification. Therefore, many burst types are defined, some of which are periodic and require re-transmission according to a schedule, and some of

Table 3.1: GMR-1, BCCH Specification [40]

Characteristic	**Value**
Burst Category	Normal Burst
Burst Type	BCCH
Symbol Rate	23.4 ksymb / sec
Modulation	$\pi/4$ QPSK
Pulse Shaping	RRC 35% Rolloff
Burst Length	10 msec, 6 slots
Repetition Cycle	320 msec
Symbol Accuracy	1/4 Symbol[1]

which are asynchronous. For this research, we focus on creating a single BCCH channel implementation, which is concerned with the generation of a frequency Sync burst (FCCH) and a broadcast control burst (BCCH). The specific burst of interest is the BCCH. It's characteristics are described in Table 3.1.

The actual implementation of the burst is unimportant to this research. As will be shown hereafter, the codesign method that this research focuses on, takes as input the required cycles to implement a communications channel, and is not as concerned with how the implementation was done. This falls on the designer ensuring they receive the desired levels of optimization and efficiency for their development costs. What is important is that the communications channel must continually repeat a BCCH transmission every 320 msec, with a time based accuracy of 1/4 symbol period, or 10.6 usec. This dictates the rate that the DUC must run in order to achieve the desired time resolution. It also dictates the number of samples that are generated by the software component of the channel, as it must keep up with a 320 msec periodic signal set. In order to reach this specifications, the DUC will run at 195.312 Ksamples/sec. This provides a time resolution of 5.12 μsec. For more information on the details of the GMR-1 specification, including channel coding and interleaving requirements, see http://www.etsi.org.

Chapter 4: Software Analysis

To perform a thorough analysis of the software side of our codesign platform, we need to understand the resources used by the software component and the architecture[1] implementation on the target platform.

4.1 Resources

4.1.1 Memory

Resources for a software component are usually viewed as less restrictive than hardware counterparts. Memory is one of the key resources of a software component. From a macro perspective, memory can be seen to host two types of information in an embedded system: data and program space. While these spaces are both critical to the function of an embedded system, memory is typically considered fairly inexpensive. Most microprocessors or digital signal processor chips have memory management capability supported out of the box to allow external memory (e.g., RAM) to be mapped into the address space of the chip. This allows boards to be manufactured with external memory interfaced directly to the software component. Therefore, external memory on the software component is not considered as a key resource in this work.

However, there is a hierarchy of memory. External memory is not the only form used in an embedded system. One form of memory that is very important to monitor is internal memory (i.e. IRAM). Most processors contain a limited amount of internal memory that is fixed at fabrication. This memory is important because it is usually fabricated closer to the core of the processor, allowing for faster access times than external memory which

[1] Architecture here refers to the software execution environment that runs on the software component. This includes an understanding of the multi-threaded nature of the OS running on the component, and any interfaces to external components.

must move information through an interface to access. Internal memory is a common place to find CACHE storage, and critical data or program elements that must run at increased speeds. Since this memory is physically located on the processor chip, increasing its capacity increases the size of the processor, resulting in lower yields per wafer, and increased price. Therefore, the amount of internal memory available to a processor plays a key role in balancing cost versus performance when designing an embedded system [14]. This internal memory is also arranged in several levels. Because the utilization of this memory can affect performance, this research includes it, labeled as IRAM, as a monitored resource. Its use in the codesign analysis is simplified to how much is required when increasing channel capacity.

4.1.2 Cycles / Time

When talking of resource utilization in context of Software, the words *time* and *cycles* are used interchangeably. This is because time can refer to the amount of relative CPU cycles that pass by during an operation, instead of time in units of seconds. When tasks are waiting for CPU context, they must wait a number of *cycles* for the existing task to finish its execution. Therefore it naturally flows when analyzing the software component to think of a *lapse-in-time* as a *lapse-in-cycles*. However, to relate this delay to real-world time, we still need to achieve time in terms we are most familiar with, seconds. To acheive time in units of seconds from CPU cycles we use the equation,

$$time[sec] = \frac{cycles}{f_{cpu}} \tag{4.1}$$

Where f_{cpu} is the frequency of the software component's CPU clock, which translates to units of $\frac{cycles}{sec}$, or Hz. For the target platform this frequency is 1 GHz.

With time established as the latency required to execute a task, the more time a task takes, the less time in a given period is available for other tasks to execute. Therefore, time (or cycles) required by a task for complete execution becomes a critical resource in our

analysis.

4.2 Target Software Architecture

4.2.1 Multi-threaded Execution environment

The target platform's software environment, TI's DSP/BIOS, all tasks (or threads) have equal priority and the execution is run-to-completion. This means that once a task gets CPU context, it will hold onto the CPU until it finishes, or intentionally yields to other tasks. Likewise, if a task yields its execution, it must wait until all other tasks in the execution loop have also executed before it receives CPU context again. Tasks of higher priority can interrupt if they become ready to execute[2], and interrupts always have priority to steal the CPU from any task[3].

The task of interest to this research is the Modulator Task. This task runs sequentially through a number of *modulator frames*, each of which implements a single communications channel. The Modulator Task is responsible for controlling the execution of all the available modulator frames, only allowing those who are currently active to execute, thus saving cycles by not calling inactive framing functions. Since all other tasks are important to this research only by the time they consume to execute, they will be collectively referred to as overhead tasks.

However, we will make the distinction between overhead tasks and interrupts, as the interrupts play an important role. Specifically, the DMA transfer interrupt that transfers the samples generated by the modulator frame across the HW/SW interface to the FPGA hardware.

[2]An example would be if a higher priority task intentionally put itself to sleep for a period of time, and then woke back up during the execution of a lower priority task. This can be done in TI's DSP/BIOS with TSK_Sleep().

[3]Unless a foreground task has explicitly disabled interrupts (e.g. critical code blocks).

4.2.2 Modeling the Target environment

A proper set of equations to model the resource requirements (e.g. cycles required to execute) for the target environment will be derived in this section. By starting with only a single modulator active, we can build an appropriate model of the software execution environment. Then we will add multiple modulators, modifying the equations where necessary.

Overhead

When a task yields the CPU, it must wait until all other tasks in the execution loop have also executed before it receives CPU context again. Our modulator framing function, which implements the communications channel by generating the samples (in a single frame or burst) at baseband, executes in the Modulator task. By denoting this modulator framing function with *mf*, we define the time duration between subsequent calls to the modulator frame as

$$T_{mf} = t_{mf} + t_{ov} \tag{4.2}$$

where t_{mf} is the time it takes to execute the modulator frame, and t_{ov} is the time it takes to execute all the other tasks collectively grouped into the overhead of the environment. Note that this number reflects the time required to execute normal foreground tasks on the software component. It does not take into account added time that may be spent in interrupt service routines for interrupts that may trigger during the normal execution loop. This time will be handled by a separate variable.

Interface to Hardware (FPGA)

The software component on the target platform is the Texas Instruments TMS320C6416 Digital Signal Processor (hereafter referred to as the DSP). It is connected to the hardware component, a Virtex4 SX55 FPGA, through an External Memory Interface (EMIF)[38]. The EMIF is configured to act as a programmable synchronous interface with a 16-bit data

and 32-bit address bus. If n 16-bit words are transferred across the interface then the time it takes to complete the transfer is

$$t_{emif}[sec] = (n[words]) \left(n_{\substack{cyc \\ word}}[cycles/word] \right) \left(\frac{1}{f_{emif}}[sec/cycle] \right) \tag{4.3}$$

Where $n_{\substack{cyc \\ word}}$ is how many CPU cycles are spent *on average* for every word written to the EMIF[4].

Note that this value is dependent only on the number of words being transferred, and the frequency of the clock driving the EMIF. Therefore, changing the sample rate of the DUC has no effect on the time it takes to transfer a block of words across the SW / HW interface. It will, however, effect how *often* the block transfer is requested.

The DSP uses the EMIF to interface with the Digital Upconverter (DUC) inside the FPGA. When the DUC's data FIFO reaches a 1/2 empty state, it sends a signal to a wire that is directly connected to one of the External Hardware Interrupt (HWI) lines on the DSP. The DSP is configured to interrupt its current task execution and service the DUC 1/2 empty HWI Interrupt Service Routine (ISR). The time it takes to execute the ISR will be referred to as $t_{\substack{hwi \\ isr}}$.

Direct Memory Access (DMA)

This DSP has a DMA onboard that is used to transfer data from the DSP to the hardware component (i.e. the FPGA). Using this DMA engine requires some cycles to configure it for a transfer. However, because the DMA engine is a separate component from the DSP (i.e. it runs independent of the DSP), the cycles that would have been spent transferring the data from the DSP to the FPGA are now returned to the DSP as free cycles that more work can be performed with. We can model the DMA cycle time by

[4]This is technically a function of the clock driving the EMIF, and is solved as a constant to simplify the equation for this platform. Other target platforms will most likely have their SW / HW interfaces configured differently, and must account for it as such.

$$t_{dma} = t_{\substack{dma \\ fore}} + t_{\substack{dma \\ back}} \tag{4.4}$$

Where $t_{\substack{dma \\ fore}}$ is the time required to configure the DMA for a transfer that is spent in the foreground, while $t_{\substack{dma \\ back}}$ is the time the DMA spends in the background transferring the data. When modeling the DMA in the target platform, for the purposes of cycle analysis, $t_{\substack{dma \\ back}}$ will be treated as *free* time and ignored. Therefore, every time a DUC 1/2 empty HWI is triggered from the FPGA to the DSP, the DSP will incur a penalty of $t_{\substack{dma \\ fore}}$ to utilize the DMA engine. To simplify things, we can absorb this value into the DSP HWI ISR which is where the DMA is configured.

$$\hat{t}_{\substack{hwi \\ isr}} = t_{\substack{hwi \\ isr}} + t_{\substack{dma \\ fore}} \tag{4.5}$$

4.3 Single Modulator Channel

4.3.1 Periodicity of the DUC HWI

We start by defining $n_{\substack{mf \\ samp}}$ as the number of samples generated by a single call of a modulator framing function, which are stored in a transmit buffer for retrieval at a later time by the DMA engine. We then define $n_{\substack{dma \\ samp}}$ as the number of samples the DMA engine pulls from that transmit buffer every time the DUC 1/2 empty HWI triggers, which samples are then transferred by the DMA to the FPGA. We can then calculate how many times the DUC 1/2 empty HWI will trigger before all the samples generated by the modulator framing function are drained out of the transmit buffer as

$$N_{hwi} = \frac{n_{\substack{mf \\ samp}}}{n_{\substack{dma \\ samp}}} \tag{4.6}$$

Once the DUC 1/2 empty HWI triggers N_{hwi} times, the transmit buffer will have run dry, and the modulator framing function will need to be called and finish executing to place new samples in the transmit buffer before the next DUC 1/2 empty HWI triggers again. Otherwise, when the next HWI occurs, the DMA will not have samples to transfer, and the DUC will run dry (i.e. underflow condition).

It therefore becomes important to know how often the DUC 1/2 empty HWI will trigger, or how many cycles are available for work between subsequent HWIs. As previously defined, if $n_{\substack{dma \\ samp}}$ are transferred on every HWI, then the DUC's data FIFO will be filled with $n_{\substack{dma \\ samp}}$ above its 1/2 empty point. If the DUC is configured for a sample rate of $f_{\substack{duc \\ rate}}$, then it will have to drain all $n_{\substack{dma \\ samp}}$ samples before the next HWI is triggered. The time that this takes can be calculated as

$$T_{\substack{next \\ hwi}} \text{ [sec]} = n_{\substack{dma \\ samp}} \left(\frac{1}{f_{\substack{duc \\ rate}}} \right) \tag{4.7}$$

This seems like the logical answer. However, one must remember that while samples are being transferred over the EMIF to the DUC's data FIFO, the DUC is *still running* and draining samples. The number of samples that are drained by the DUC during the DMA transfer can be found as

$$n_{\substack{samp \\ drained}} = (\hat{t}_{\substack{hwi \\ isr}} + t_{emif}) f_{\substack{duc \\ rate}} \tag{4.8}$$

Since these samples drained while the DMA was transferring new samples across the EMIF, we have effectively decreased the number of samples that are now sitting above the 1/2 empty line in the DUC's data FIFO. This gives an *effective* time between subsequent DUC 1/2 empty HWI triggers of

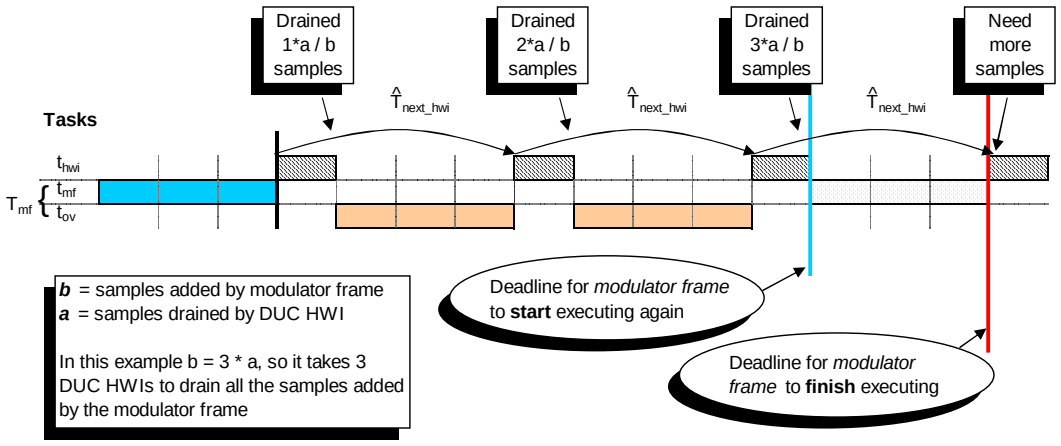

Figure 4.1: Timing Diagram showing the deadline for the modulator frame task, denoted by t_{mf}, to ensure that the transmit buffer does not run dry between subsequent DUC HWIs.

$$\hat{T}_{\substack{next \\ hwi}} \text{ [sec]} = (n_{\substack{dma \\ samp}} - n_{\substack{samp \\ drained}}) \left(\frac{1}{f_{\substack{duc \\ rate}}} \right) \tag{4.9}$$

Figure 4.1 provides a timing diagram portraying the relationship between the variables derived up to this point. This diagram assumes that the modulator frame was just finished being called, and has added its samples to the transmit buffer before the first DUC HWI occurs. In this example, after three DUC HWIs, the transmit buffer has been emptied of all the samples that the modulator frame added. Therefore, before the third $\hat{T}_{\substack{next \\ hwi}}$ occurs, the modulator frame must *finish* executing to add the next block of samples to the transmit buffer, so that the DUC HWI ISR has samples to transfer.

4.3.2 Periodic Time constraint

We are now ready to derive the periodic constraint to prevent the underflow condition.

$$T_{mf} + N_{hwi} \hat{t}_{\substack{hwi \\ isr}} < N_{hwi} \hat{T}_{\substack{next \\ HWI}} \tag{4.10}$$

We can simplify this in terms of important system parameters. First, pluging 4.8 into 4.9 provides

$$\begin{aligned}
\hat{T}_{\substack{next\\hwi}} &= (n_{\substack{dma\\samp}} - n_{\substack{samp\\drained}}) \left(\frac{1}{f_{\substack{duc\\rate}}} \right) \\
\hat{T}_{\substack{next\\hwi}} &= (n_{\substack{dma\\samp}} - (\hat{t}_{\substack{hwi\\isr}} + t_{emif}) f_{\substack{duc\\rate}}) \left(\frac{1}{f_{\substack{duc\\rate}}} \right) \\
\hat{T}_{\substack{next\\hwi}} &= \left(\frac{n_{\substack{dma\\samp}}}{f_{\substack{duc\\rate}}} - (\hat{t}_{\substack{hwi\\isr}} + t_{emif}) \right) \\
\hat{T}_{\substack{next\\hwi}} &= \left(\frac{n_{\substack{dma\\samp}}}{f_{\substack{duc\\rate}}} - \hat{t}_{\substack{hwi\\isr}} - t_{emif} \right) \text{ [sec]} \qquad (4.11)
\end{aligned}$$

Next we plug 4.6 and 4.11 into 4.10 to obtain

$$\begin{aligned}
T_{mf} + N_{hwi} \left(\hat{t}_{\substack{hwi\\isr}} \right) &< N_{hwi} \left(\frac{n_{\substack{dma\\samp}}}{f_{\substack{duc\\rate}}} - \hat{t}_{\substack{hwi\\isr}} - t_{emif} \right) \\
T_{mf} + \frac{n_{\substack{mf\\samp}}}{n_{\substack{dma\\samp}}} \left(\hat{t}_{\substack{hwi\\isr}} \right) &< \frac{n_{\substack{mf\\samp}}}{n_{\substack{dma\\samp}}} \left(\frac{n_{\substack{dma\\samp}}}{f_{\substack{duc\\rate}}} - \hat{t}_{\substack{hwi\\isr}} - t_{emif} \right) \\
\left(\frac{T_{mf}}{n_{\substack{mf\\samp}}} \right) + \left(\frac{\hat{t}_{\substack{hwi\\isr}}}{n_{\substack{dma\\samp}}} \right) &< \frac{1}{n_{\substack{dma\\samp}}} \left(\frac{n_{\substack{dma\\samp}}}{f_{\substack{duc\\rate}}} - \hat{t}_{\substack{hwi\\isr}} - t_{emif} \right) \\
\left(\frac{T_{mf}}{n_{\substack{mf\\samp}}} \right) + \left(\frac{\hat{t}_{\substack{hwi\\isr}}}{n_{\substack{dma\\samp}}} \right) &< \frac{1}{f_{\substack{duc\\rate}}} - \frac{\hat{t}_{\substack{hwi\\isr}} + t_{emif}}{n_{\substack{dma\\samp}}} \\
\left(\frac{T_{mf}}{n_{\substack{mf\\samp}}} \right) + \left(\frac{2\hat{t}_{\substack{hwi\\isr}} + t_{emif}}{n_{\substack{dma\\samp}}} \right) &< \frac{1}{f_{\substack{duc\\rate}}} \qquad (4.12)
\end{aligned}$$

Now, rearranging and expressing in terms of the DUC sampling period, we obtain

$$T_{\substack{duc \\ period}} > \left(\frac{T_{mf}}{n_{\substack{mf \\ samp}}}\right) + \left(\frac{2\hat{t}_{\substack{hwi \\ isr}} + t_{emif}}{n_{\substack{dma \\ samp}}}\right) \text{[sec/sample]} \tag{4.13}$$

where $T_{\substack{duc \\ period}} = \frac{1}{f_{\substack{duc \\ rate}}}$. To better analyze the value that 4.13 provides to the codesign effort, we will rewrite it in general descriptive terms, while maintaining the mathematical relationship.

$$\underset{\text{Period}}{\text{HW}} > \left(\frac{\text{Time Generating Samples}}{\text{\# Samples Generated}}\right) + \left(\frac{\text{Time Transfering Samples}}{\text{\# Samples Transfered}}\right) \tag{4.14}$$

The resulting 4.14 has some key points. The main software execution loop that contributes to the periodic constraint, consisting of the modulator frame, and other overhead routines, is normalized by the samples they generate. Likewise, the architecture portion of the software that contributes to the periodic constraint, consisting of the HWI ISR and EMIF transfer, is normalized by the number of samples it is responsible for transferring. Both of these add together to provide a normalized time it takes to generate samples to be sent out the DUC into the spectrum, and must occur in less time than one full period of the DUC, the HW block under test, to avoid the underflow condition.

Interestingly, this concept of normalizing the work performed by the output of the work for the different portions of the components on the platform is also apparent in Equation 2.2 from [11] described in Section 2.1.3. This also took the form of normalizing portions of work by the requirements to implement them, creating the fitness function score. However, the relationship between the components of Equation 2.2 is a multiplicative one versus an additive one expressed here in Equation 4.14.

With 4.13 defined, all the system designer needs to do to ensure they meet this criteria is

define time requirements for the software components of their system. This can be done by borrowing from previous designs, profiling existing implementations, or using computation equations for the basic signal processing algorithms that will be implemented on the target platform. Many chip developers provide cycle benchmarks for their products for common algorithm implementations.[5]

4.4 Multiple Modulator Channels

To determine how many modulators can run on the software component of the codesign platform, we take two key software constraints and evaluate them independently: cycle and internal memory requirements of the software modulator function. Both of these are addressed independently, and their results are compared together to determine the true upper limit of the maximum number of modulators that can run on the software portion of the platform.

Cycle Requirement Viewpoint

When the system developer has enough information to try 4.13, and verify that the constraint is met in the design, the next question this research focuses on follows: How many instances of the modulator frame can run simultaneously?[6]

The answer to this question can be approached by first looking back at the result from 4.13. Assuming that the right hand side of 4.13 has been solved for, we define

$$\beta = \left(\frac{T_{mf}}{n_{\substack{mf \\ samp}}} \right) + \left(\frac{2\hat{t}_{\substack{hwi \\ isr}} + t_{emif}}{n_{\substack{dma \\ samp}}} \right) \tag{4.15}$$

which is in [sec/sample]. This β can be seen as the total time[7] required to implement a

[5]For example, Texas Instruments provides specific equations involving clock frequency for FIR filter implementations. Refer to www.ti.com.

[6]In this work, separate *instances* refer to additional instantiations of the software side of a communications channel, i.e. the modulator frame.

[7]Technically the units of β are [sec/samp], but the general idea is a unit of time required to do work

single modulator instance, or a single communications channel. By dividing β by $T_{\substack{duc \\ period}}$, we obtain the percentage of time used during a single period of the DUC HWI by the modulator. All modulators that are currently active must complete their work within a single period of the DUC to ensure none of them underflow. Therefore, a good rule of thumb for the total number of modulators that can run simultaneously is the inverse of this number, or

$$N_{\substack{max \\ active \\ mods}} = floor\left[\frac{T_{\substack{duc \\ period}}}{\beta}\right] \tag{4.16}$$

Memory Requirement Viewpoint

Notice that the result from 4.16 does not depend on memory requirements. We know that memory is limited, and that with each increase in number of active modulators, there is an increase in memory usage. This is due to each modulator requiring usage of portions of the STACK or internal ram (IRAM) for saving state information. Since each modulator's transmit buffer resides in IRAM to give un-obstructed access to the DMA engine for transferring the modulated samples to the FPGA, and IRAM is seen as the most costly, and finite, memory portion of the CPU on the platform, our memory constraint deals specifically with IRAM. Therefore, we define an upper limit dictated by memory available to use for modulators as

$$N_{\substack{mods \\ mem \\ limit}} = floor\left[\frac{MEM_{iram} - MEM_{unavail}}{MEM_{mod}}\right] \tag{4.17}$$

where MEM_{iram} is how much IRAM is available on the system, $MEM_{unavail}$ is the amount of IRAM that is used for other things, such as overhead, that run on the CPU and is unavailable for use by new modulator functions, and MEM_{mod} is how much IRAM is used

relating to sample generation and transmission.

by each modulator. As described earlier, we are only interested in IRAM, as external memory is considered inexpensive, and available in large quantities in this analysis.

With 4.16 and 4.17 established, the total number of modulators that can be active as viewed from the software component is

$$N_{\substack{SW \\ mods}} = min \left[N_{\substack{max \\ active \\ mods}}, N_{\substack{mods \\ mem \\ limit}} \right] \tag{4.18}$$

4.4.1 Note on Overhead Increase

Equation 4.16 takes some liberties. Specifically, it assumes that the overhead required to run multiple instances of the modulator frame scales linearly with the number of active modulators. Overhead does increase with multiple active modulators, but the rate at which it increases depends on the software architecture implementation of the platform. One approach would be to model the overhead increase as

$$T_{\substack{ov \\ total}} = t_{ov} + (M-1)\Delta_{ov} \tag{4.19}$$

where M is the number of modulators, and Δ_{ov} is a specified amount of overhead increase that is incurred by the software architecture for each additional activated modulator beyond the first. However, even this is an approximation, as the overhead increase is most likely not a linear function. Many architecture factors may be effected by more modulators. For example, cache misses may increase with more modulators, requiring a greater load on the DMA engine since it will have to service execution loop tasks as well as the sample transfer across the SW/HW interface. The complexity of modeling overhead increase is dependent on the target platform software architecture implementation. This research does not focus on closely modeling the overhead increase, but instead found that 4.16 was a valid rule of thumb approach.

Chapter 5: Hardware Analysis

The previous chapter discussed how we derived a set of equations used to estimate software requirements on the codesign platform in terms of cycle and execution constraints. Likewise, this chapter will perform the same analysis on the hardware portion of the codesign platform. The approach will be centered around key resources inside the FPGA device, and develop constraints around these resources. The first approach was to estimate total hardware requirements by examining the hardware requirements of each sub-block and linearly summing these together. However, this chapter will show, by using the implementation of a Digital Upconverter (DUC) inside a Xilinx Virtex4 SX55 FPGA, that the estimation of total resources must take into account not only the resource requirements of the individual blocks that make up the FPGA build, but also the total device utilization percentage of the FPGA. A modified version of linearly summing the sub-blocks together is presented that is shown to minimize the error in predicted versus actual resource utilization. This new modified approach is then used further in the codesign analysis of the research.

5.1 Resources

In a software component, the most costly resource is usually cycles available for processing. The design implementation on the software component affects this resource directly, and not many others.[1] In contrast, hardware components have many resources to manage, all of which are typically directly affected by design implementation. This section lists the resources under consideration for the target platform.

Table 5.1: Key HW Onchip Resources, in the Xilinx Virtex4 SX55 FPGA

Resource	Available in SX55
CLB Slices	24576
Flip Flop Slices	49152
Look Up Tables	49152
Block RAM	320
DSP48	512

5.1.1 Onchip Resources

Table 5.1 shows the key resources that are considered in this research, as well as the quantity available in the SX55 FPGA (FPGA) component. These resources are seen as critical. As the design implementation changes, the quantity of each resource changes as well. When designing a block, it becomes important to watch the utilization of the block throughout the design phase.

Many synthesis tools are available to take an FPGA design written in a hardware description language, such as VHDL or Verilog[2], and given a specific hardware component (e.g. Xilinx V4 SX55 FPGA), provide utilization statistics. If a specific design block becomes too large (e.g. requires too many BRAMs), the designer risks not being able to fit their block into the physical component. Designs then have to be scaled back and possibly re-engineered.

5.1.2 Device Utilization Threshold

When designing logic blocks independently inside an FPGA, to be later dropped into the overall FPGA build, one can monitor the resources the block synthesizes to when built alone. However, this resource utilization can change when the block is eventually dropped into the final build with surrounding FPGA logic. This can be due to availability of key resources changing as other blocks that synthesize might require them as well.

When the overall FPGA design becomes close to full capacity, the industry tools begin

[1]Memory usage by the code is typically another resource considered limited in many embedded applications. However, in this research, memory is readily available on the SDR.

[2]All the FPGA code for this research is in VHDL

to struggle with routing the various components together to complete the logical circuit inside the FPGA. As an FPGA reaches higher utilization, there are fewer paths for the tools to route the signals through. This can result in very long FPGA builds, sometimes failing to meet timing after hours of iterations (i.e. the tool just gives up). Because of this, another key resource is *Device Utilization Threshold*. When a device (i.e. FPGA) begins to reach a certain level of utilization, this becomes a warning sign that designs may need to be re-scaled, or routing issues can occur. This threshold can be used as an early warning in the codesign analysis.

5.2 Target Hardware Architecture

5.2.1 The Digital Up Converter (DUC)

As mentioned previously in Section 3.3, the main portion of hardware on the target platform is the Digital Up Converter (DUC), which provides any remaining upsampling to convert the sample stream from baseband to intermediate frequency (IF) before being sent to backend hardware components for conversion from a digital to analog signal, and from IF to RF.

5.3 Single & Multiple DUC Channels, Resource Prediction

Since the codesign method involves possibly moving stages of the communications channel between the software and hardware components, it becomes necessary to properly predict the resources required by the key components inside the hardware device. This section uses the DUC as a case study to attempt a prediction of hardware device utilization percentage by a linear summation of the critical components. It will be shown that this approach may not provide enough accuracy in the prediction. A modification to the approach will be presented that brings the error between predicted and actual device utilization percentage into acceptable levels.

Table 5.2: DUC Device Utilization

Resource	DucUtil	Available in SX55	% of Chip
CLB Slices	1165	24576	4.8 %
Flip Flop Slices	1881	49152	3.9 %
Look Up Tables	812	49152	1.7 %
Block RAM	6	320	1.9 %
DSP48	21	512	4.1 %

Table 5.3: FPGA Base Build Utilization

Resource	FPGA Base Build	Available in SX55	% of Chip
CLB Slices	1974	24576	8.0 %
Flip Flop Slices	2597	49152	5.3 %
Look Up Tables	3212	49152	6.5 %
Block RAM	11	320	3.4 %
DSP48	16	512	3.1 %

5.3.1 Linearly Summed Device Utilization Increase

Table 5.2 shows the device utilization for instantiating a single DUC inside the FPGA. The synthesis tool used in this research is included in the Xilinx ISE toolkit.

To have a starting point to attempt the linear summation prediction method, we built the entire FPGA with no DUCs instantiated, and recorded the device utilization. This is referred to as the FPGA Base-Build and the device resource utilization is shown in Table 5.3.

For the linear summation approach, ideally the device utilization should increase by

$$DevUtil_{n-blocks} = DevUtil_{fpga-bb} + n * BlockUtil \tag{5.1}$$

where $DevUtil_{n-blocks}$ is the total FPGA device utilization when instantiating n blocks, in this case n DUCs, $DevUtil_{fpga-bb}$ is the device utilization of the FPGA Base-Build (i.e. from Table 5.3), and $BlockUtil$ is the device utilization provided by synthesizing that block alone, or outside of the full FPGA build (i.e. DucUtil, Table 5.2). Table 5.4 shows the device utilization for building the full FPGA build with one DUC instantiation.

Using 5.1 we can come up with a prediction of the FPGA device utilization for 2, 4, 8,

Table 5.4: FPGA Device Utilization (1 DUC)

		1 DUC Instant	
Resource	**DucUtil**	**DevUtil**	**% of Chip**
CLB Slices	1165	3292	13.4
Flip Flop Slices	1881	4725	9.7
Look Up Tables	812	4169	8.5
Block RAM	6	17	5.4
DSP48	21	37	7.3

Table 5.5: Predicted Device Utilization (2,4,8 & 12 DUCs)

	2 DUCs		*4 DUCs*		*8 DUCs*		*12 DUCs*	
Resource	**DevUtil**	**% of Chip**	**DevUtil**	**% of Chip**	**DevUtil**	**% of Chip**	**DevUtil**	**% of Chip**
CLB Slices	4304	17.5	6634	27.0	11294	46.0	15954	64.9
Flip Flop Slices	6359	12.9	10121	20.6	17645	35.9	25169	51.2
Look Up Tables	4836	9.8	6460	13.1	9708	19.8	12956	26.4
Block RAM	23	7.2	35	10.9	59	18.4	83	26.0
DSP48	58	11.3	100	19.5	184	35.9	268	52.3

and 12 DUC instantiations. Table 5.5 shows these predicted values. To verify this result, the full FPGA was built (with DUC instantiations) to provide an actual utilization for 2, 4, 8, and 12 DUC instantiations. These actual utilization values are shown in Table 5.6. Comparing these two tables shows a discrepancy between predicted and actual device utilization. To determine if this discrepancy is of importance to our codesign analysis (i.e. is large enough to matter) we compute the error between predicted and actual, shown in Table 5.7, and plot this error for each device resource in Figure 5.1.

The error appears to exhibit different behavior for different resources. The BRAM exhibits no error, thereby showing that equation 5.1 is sufficient to determine utilization for a specified number of DUCs. However, LUT resources show close to a linear increase in error as the number of DUCs increases, while Slices, SliceFF, and DSP48s show exponential

Table 5.6: Actual Device Utilization (2,4,8 & 12 DUCs)

	2 DUCs		*4 DUCs*		*8 DUCs*		*12 DUCs*	
Resource	**DevUtil**	**% of Chip**	**DevUtil**	**% of Chip**	**DevUtil**	**% of Chip**	**DevUtil**	**% of Chip**
CLB Slices	4592	18.7	7210	29.3	12473	50.8	17882	72.8
Flip Flop Slices	6765	13.8	10974	22.3	19389	39.4	28066	57.1
Look Up Tables	5089	10.4	6965	14.2	10555	21.5	14201	28.9
Block RAM	23	7.2	35	10.9	59	18.4	83	26.0
DSP48	60	11.7	106	20.7	198	38.7	298	58.2

Table 5.7: Predicted Utilization Error

Resource	1 DUC	2 DUCs	4 DUCs	8 DUCs	12 DUCs
CLB Slices	0.6%	1.2%	2.3%	4.8%	7.8%
Flip Flop Slices	0.5%	0.8%	1.7%	3.5%	5.9%
Look Up Tables	0.3%	0.5%	1.0%	1.7%	2.5%
Block RAM	0.0%	0.0%	0.0%	0.0%	0.0%
DSP48	0.0%	0.4%	1.2%	2.7%	5.9%

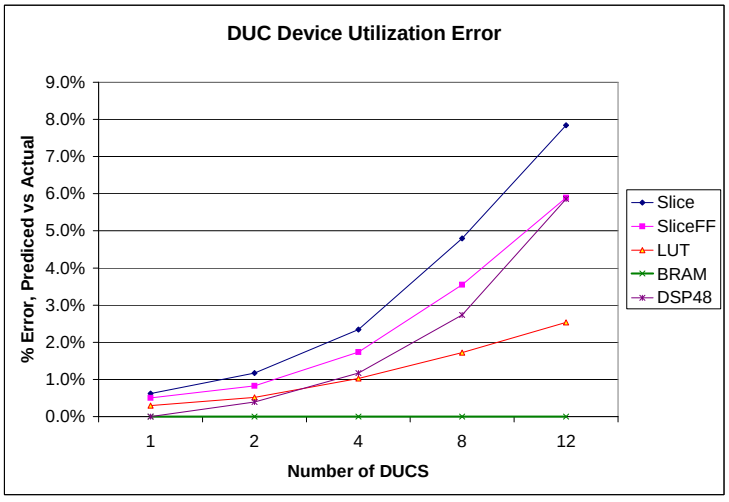

Figure 5.1: Error in DUC predicted versus actual device utilization using equation 5.1.

increase in error. This error occurs due to the stress put on the synthesis and routing tool when the device utilization reaches high levels. It becomes harder for the tool to connect all of the circuitry inside the FPGA, while still meeting timing requirements specified by the tool's constraints file.

The error for all of these resources is still under 10%, which may be acceptable for a particular codesign analysis. If the particular analysis is close to 100% utilization, or if greater accuracy is preferred in utilization prediction, then one could consider modifying equation 5.1. While the focus of the research is not entirely on increasing accuracy in prediction of device utilization on the HW side, an alternative model is described next that provides greater accuracy. Future work might focus on greater accuracy of device utilization prediction on the HW side of a codesign platform.

5.3.2 Linearly Scaled Device Utilization Increase

As the number of required DUCs increases, the device utilization increases providing greater complexity in the routing process of the FPGA. Considering the case of the LUT resource error shown in Figure 5.1, the error appears to scale upward linearly with the utilization increase. Therefore, adding a linear multiplicative factor on the number of DUCs might provide a closer predicted utilization, as shown in 5.2.

$$DevUtil_{n-blocks} = DevUtil_{fpga-bb} + (n * \alpha)BlockUtil \tag{5.2}$$

where α is a multiplicative factor to account for routing conditions. A procedure for using 5.2 in place of 5.1 is,

1. Run required number of devices through 5.1 to obtain a $DevUtil_{n-blocks}$.
2. Check if

$$DevUtil_{n-blocks} > DeviceUtilizationThreshold$$

where Device Utilization Threshold[3] is a predetermined utilization level, then...

[3]This threshold value is considered an input, and will change depending on the platform, and the concerns

Table 5.8: Predicted Utilization Error: Linearly Scaled Model

Resource	**1 DUC**	**2 DUCs**	**4 DUCs**	**8 DUCs**	**12 DUCs**
CLB Slices	0.0%	0.0%	0.0%	0.0%	0.7%
Flip Flop Slices	0.0%	-0.1%	-0.2%	-0.3%	0.1%
Look Up Tables	0.1%	0.1%	0.2%	0.1%	0.0%
Block RAM	0.0%	0.0%	0.0%	0.0%	0.0%
DSP48	-0.5%	-0.6%	-0.9%	-1.4%	-0.3%

3. Rerun required number of devices through 5.2 to obtain a refined $DevUtil_{n-blocks}$.

To prove this iterative process provides increased prediction accuracy, Equation 5.2 was used for all resources in Table 5.1 except for Block Ram[4] which had 0% error after the first iteration. The results, shown in Table 5.8 and Figure 5.2, show that using Equation 5.2 for the second iteration with an $\alpha = 1.1253$ provides a predicted error with a magnitude of less than 1.5%. The α value was chosen based on a value that minimized the LUT error. After that value was chosen, it was used for the other resources, and accepted when it showed the small level of error between predicted and actual device utilization. Therefore, for this research, 5.2 will be used since the goal of maximizing number of communications channels will require high levels of utilization in the target platform FPGA.

While 5.2 provides greater accuracy in prediction, it does require more offline work as one must use a case example on their platform to first determine the value of α that will be used to scale the hardware requirements. Once found, while not perfect, this value will provide greater accuracy than a straight summation as presented previously in Equation 5.1.

of the FPGA design engineers.

[4]Block Ram is physical RAM chips dropped into the FPGA at fabrication, and are not *created* when synthesizing an FPGA. They still need to be routed, but there is a finite number of them. If one properly infers X BRAMs then X BRAMs will be used.

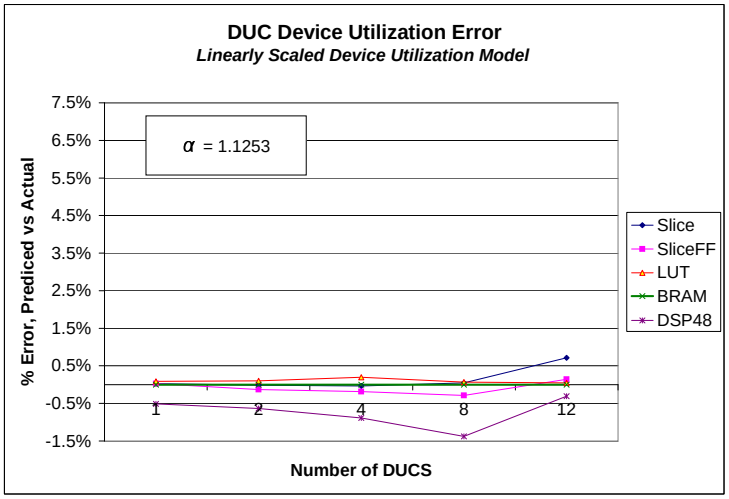

Figure 5.2: Error in DUC predicted versus actual device utilization using equation 5.2 with an α of 1.1253.

Chapter 6: Codesign Engine Implementation(CDE)

This section describes the CDE implementation. The CDE basically takes the equations and implements the method resulting from the analysis of the previous chapters, and realizes them into an application that can be executed by a developer, allowing flexible control of the inputs.

The first version of the CDE came in the form of Chapters 4 and 5, derived equations on paper formed from the study of an existing architecture on a codesign platform. Once these base equations were obtained, they were then implemented in MATLAB to allow the input parameters to be tweaked and analysis re-run with ease.

6.1 MATLAB Implementation

The SW and HW analysis were implemented as two independent .m files, each with its own utilization and constraints file. The two sets of files can be manipulated by the developer for their baseline implementation of the stages of the communications channel by editing the values directly in the file, and then re-running the .m files. The output of the .m files are graphs of the baseline implementation, before any movement of resources across the HW/SW interface, and then graphs of the improved implementation, with a dialog box that informs the developer how many SW and HW portions of the channel can fit into the target platform.

6.1.1 Inputs to the CDE

As CDE matures, and builds a database, it should be able to increase its prediction and suggestive capacity, thereby decreasing the number of application specific inputs required.

Table 6.1: CDE Software Constraints File

Parameter	Description
SYS_SW_NumRequestedModulators	Number of desired modulators, or channels
SYS_SW_OverheadIncrPerMod	Additional overhead cycles incurred for each additional modulator
SYS_SW_IRAMUsagePerMod	IRAM Memory usage per modulator
SYS_SW_TotalIRAM	Total IRAM available on the target platform's CPU
SYS_SW_IRAMUnavailable	Total IRAM unavailable for use by new modulators
SYS_SW_SamplesPerHWI	Number of samples transferred across the HW/SW interface every DUC HWI
SYS_SW_SamplesPerMFrame	Number of samples generated by a single call to the modulator frame
SYS_SW_SymbolsPerMFrame	Number of symbols generated by the modulator frame
SYS_SW_CPURate	Number of cycles / sec of the clock running the CPU software component
SYS_SW_EmifClockRate	Number of cycles / sec of the clock running the EMIF interface
SYS_SW_EmifCPUCyclesPerWord	CPU cycles that pass to transfer one 16-bit word across the EMIF
SYS_SW_SymbolRate	Symbol rate of the protocol the communications channel is implementing
SYS_SW_DucRate	Number of samples / sec the DUC inside the FPGA drains from its FIFOs

In its current form, it requires all the information to solve for the equations in Chapters 4 and 5. There are two files that provide this information for both the SW and HW portions.

Constraints File

The constraints file provides information to the CDE describing the SW component's existing architecture characteristics. For example, the clock rate of the CPU, or the device utilization threshold. However, these do not describe the resources available explicitly, such as number of cycles of existing overhead, or number of Slices available in the FPGA. Table 6.1 lists the SW constraints, and Table 6.2 lists the HW constraints used in the MATLAB CDE. Each constraints variable is prefaced with a "SYS_SW_" to distinguish it in the analysis from other derived variables.

The information from these tables can be used to calculate key parameters that are necessary for the analysis of a component. For example, Source 1 shows how constraints

Table 6.2: CDE Hardware Constraints File

Parameter	Description
SYS_HW_DevUtilThresholdPerc	A device utilization threshold that once breached will stop the CDE
SYS_HW_MultiFactorIncrease	The α value used in the linear scaled device utilization increase
SYS_HW_useLinearScaledModel	Boolean, 1 to use the linear scaled device utilization increase, 0 to not

from the SW constraints file are used to calculate the time it takes to transfer a specified number of samples across the EMIF, which is Equation 4.3 in Chapter 4.

```
% ------------------------------------------------------------
% Get EMIF Cycles Spent Transfering in terms of sec
% ------------------------------------------------------------
% We need to calculate how much time it takes to transfer the cycles
% for HWI across the EMIF.  We are transfering complex sample IQ pairs
% so we multiply by 2 to get total words transfered (16-bit I,16-bit Q)
% Assumes 16-bit EMIF data bus.
SW_EmifTransferTime =
    2*SYS_SW_SamplesPerHWI*SYS_SW_EmifCPUCyclesPerWord/SYS_SW_EmifClockRate;
```

Source 1: Calculating time spent transfering cycles across the EMIF.

And the Source 2 shows that after we have arrived at the EMIF transfer time, we can use this results to calculate the $n_{\substack{samp \\ drained}}$ from Equation 4.8, which can be used to give an effective samples transferred, which in turn provides the time between DUC HWIs, $\hat{T}_{\substack{next \\ hwi}}$ from Equation 4.9. Notice how all derived values used inside the CDE are prefaced with only a "SW_" instead of "SYS_SW_" as are inputs from the constraints file.

Just as has been presented in these two examples, all the remaining equations derived in Chapter 4 and 5 are solved for. However, in order to do so, the developer must provide information concerning the cycle requirements and device utilization of the current and proposed implementations. These inputs are provided in the software cycle requirements file and hardware device utilization files, collectively referred to as the utilization files.

```
% ------------------------------------------------------------
% Get Effective Sample Per HWI
% ------------------------------------------------------------
% Now that we have the EMIF transfer time we can calculate how many samples
% drain from the DUC FIFO during the HWI ISR overhead to get an effective
% sample count added to the DUC FIFO on each HWI call.  We convert all
% variables to [sec] to ensure matching units during the calculation.
% This is important because some parts run at different clock rates, e.g,
% the EMIF runs at a different rate than the HWI ISR code.
SW_TimePerDucHwiIsr = CycReq_DucHwiIsr/SYS_SW_CPURate;
SW_NumSamplesDrained = (SW_TimePerDucHwiIsr + SW_EmifTransferTime)*SYS_SW_DucRate;
SW_NumEffectiveSamplesPerHWI = SYS_SW_SamplesPerHWI - SW_NumSamplesDrained;

% ------------------------------------------------------------
% Get Time between HWI's, T_next_hwi
% ------------------------------------------------------------
% Now that the effecgive sample per HWI has been calculated, since we know
% the Rate at which the DUC drains its DATA FIFO, we can then calculate
% how often the DUC Data FIFO 1/2 empty HWI will trigger the Software DSP
% chip.
SW_TimeNextHWI = SW_NumEffectiveSamplesPerHWI / SYS_SW_DucRate;
```

Source 2: Calculating the time between DUC HWIs.

Table 6.3: CDE Software Utilization File

Parameter	**Description**
CycReq_MFStageNames	Array of names for each stage of the modulator frame
CycReq_MFUsed	Array of cycles consumed by the stages of the modulator frame
CycReq_Overhead	Cycles consumed by the overhead of the SW component
CycReq_DucHwiIsr	Cycles consumed by the Duc HWI ISR

Utilization File

The utilization files provide numbers for how many resources are consumed by the various stages of the communications channel in the SW and HW components. Table 6.3 and 6.4 show the utilization file contents for the case-study, with a description of each entry.

Note that the software cycle requirements are prefaced with "CycReq_" while a "DevUtil_" prefaces the parameters for the hardware file. These are the main numbers that are manipulated by the developer when working with the codesign platform. By changing the values in these files, the developer can determine where the most effort in design needs to

Table 6.4: CDE Hardware Utilization File

Parameter	Description
DevUtil_ResourceNames	Array of names for each key resource in the FPGA
DevUtil_ResourcesAvailable	Total amount available of those key resources
DevUtil_FPGABaseBuild	Resources consumed by the FPGA base build
DevUtil_SingleDUC	Resources consumed by a single DUC instantiation
DevUtil_SingleResampler	Resources consumed by a single Resampler instantiation

be spent in order to obtain the desired channel capacity in the system.

6.1.2 Outputs from the CDE

With the constraints and utilization files provided as inputs, the CDE can calculate the channel capacity of the codesign platform for both the existing configuration, and an improved configuration. The existing configuration follows from the values provided in these two files. To obtain the improved version for the software side, the CDE removes the stage closest to the interface, which in our case-study is the Resampler, and re-runs the analysis. The hardware side adds in the new component (i.e. the polyphase resampler implementation), and iteratively increases the number of instantiated DUCs and Resamplers using the equations for increase derived in Chapter 5. At the end of each iteration, the total device utilization is compared to the provided Device Utilization Threshold. Once the obtained device utilization just breaches the threshold, the CDE breaks out of the iteration and finishes the analysis.

For both existing and improved configuration, pie charts (for the software side) and bar charts (for the hardware side) are produced that visually display to the developer cycles used and device utilization consumed, respectively. Examples of these charts will be shown in Section 6.2 which deals with the specific example case-study of the research.

The CDE also outputs the results that are used to generate the pie and bar charts to a single file, cde_output.txt. This file contains ascii tables showing the SW and HW component resource requirements to implement the number of modulators for the baseline and improved implementation. An example portion of this file is shown in Source 3. Note

Figure 6.1: CDE Software Results Message Box.

that while the total number of DUCs has gone down from 16 to 12 from the baseline to improved implementation, the improved implementation has the addition of Resamplers attached to the 12 DUCs, which will be shown in Section 6.2 to provide an increase in total platform capacity of 20%.

```
⋮
*****************************************************
Baseline Implementation (HW side)
*****************************************************
The Baseline Implementation can fit 16 DUCs.
The required utilization for the Baseline Implementation is listed below

                 Slices,SliceFF,LUT,BRAM,DSP48
FPGABaseBuild(%) 8.0,5.3,6.5,3.4,3.1
DUC          (%) 85.3,68.9,29.7,30.0,73.8
-------------------------------------------------
Total Usage  (%) 93.4,74.2,36.3,33.4,77.0

*****************************************************
Improved Implementation (HW Side)
*****************************************************
The Improved Implementation can fit 12 DUCs with Resamplers.
The required utilization for the Improved Implementation is listed below

                 Slices,SliceFF,LUT,BRAM,DSP48
FPGABaseBuild(%) 8.0,5.3,6.5,3.4,3.1
DUC          (%) 64.0,51.7,22.3,22.5,55.4
Resampler    (%) 23.2,14.1,13.4,30.0,2.6
-------------------------------------------------
Total Usage  (%) 95.3,71.1,42.2,55.9,61.1
⋮
```

Source 3: Example portion of cde_output.txt file.

At the end of each SW/HW portion, a message box appears telling the developer the new number of communications channel portion that can fit in either section, as shown in

Figure 6.2: CDE Hardware Results Message Box.

Figures 6.1 and 6.2. As in the example provided by these Figures, the total number of communications channels that can exist on the entire platform is the minimum of these two results, or 12 in this case. As the case-study will show, this resulted in a prediction of 2 additional channels if the Resampler is moved to hardware, given the resource utilizations for the stages in the input utilization files. This gives an increase of $2/10 = 20\%$.

6.2 Case Study, Specific Example: Polyphase Resampler

This section describes the case study on the target platform that was used to prove the CDE method. It follows a step by step analysis using the equations derived from the Software and Hardware Analysis chapters discussed previously. The results of this analysis are verified with true profiling on the target platform running the communications application in real-time, thus proving the validity of the method discussed in this paper for this platform. The general method can then be applied to other platforms, with the inputs changing slightly for different architectures and their models.

6.2.1 Baseline Implementation

One of the first pieces of information a hardware software platform codesigner looks for when trying to add capacity to the system is what the *current* capability is. This section describes an analysis that predicts the maximum number of communications channels that can be instantiated on the target platform. Using the method and equations from previous chapters, and verifying the results with true profiles made directly on the platform running

Table 6.5: Software Cycle Requirements

Task	**Cycles**
Overhead	50000
Duc Hwi Isr	12669
Modulator Frame	648325

in real-time, the results show a correct prediction of platform channel capacity in the baseline implementation.

Software: Max capacity

The first thing that was needed to begin using the CDE described previously was to profile the baseline implementation. For this case study the profile was done on the actual platform, providing cycle benchmarks by recording the CPU internal clock counter values before and after a function was called, accounting for any clock counter overflows that may have occurred during the function execution. The TI 6416 DSP provides easy access to these counters and the ability to monitor overflow counts as well. The results for profiling the software cycle requirements of the target platform architecture are shown in Table 6.5.

Recall that Overhead is really treated as a lump-sum that contains typical architecture overhead (i.e. context switching requirements) and also other tasks running on the CPU on this platform that are not related to the communications channel. They are seen as necessary tasks that will always be running, and unavailable for adjustment by the codesigner.

Since the modulator frame is the portion of the design that is under consideration, we profiled the individual stages in the modulator frame's implementation. These cycle profiles are shown in Table 6.6. They are graphically represented in Figure 6.3, where 100% of the pie chart is the entire modulator frame cycle usage, or the total cycles from Table 6.6.

From Figure 6.3 it becomes obvious that the Resampler portion of the modulator frame is very unbalanced from the other stages in terms of cycle requirements. This portion therefore becomes the first choice candidate for the CDE to attempt to move out of the software portion and into the hardware portion of the target platform.

However, the goal for this portion of the CDE is to predict current capacity. In order

Table 6.6: Modulator Frame Cycle Requirements

Task	**Cycles**
Bookkeeping	2100
CRC Computation	12000
Encoding	28000
Interleaving	55000
Scrambling	23225
Symbol Modulation	19000
Resampling	509000
Total	**648325**

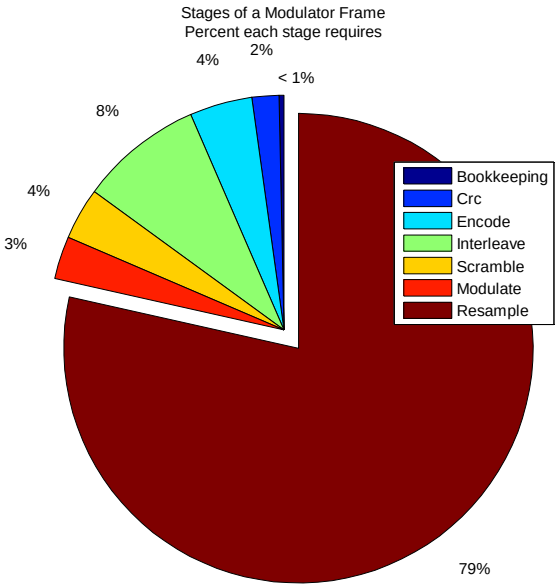

Figure 6.3: Cycle requirements of the individual stages of a modulator frame.

Table 6.7: CDE Software Constraint File Values

Parameter	**Value**
SYS_SW_NumRequestedModulators	11
SYS_SW_OverheadIncrPerMod	50000
SYS_SW_IRAMUsagePerMod	32768
SYS_SW_TotalIRAM	1048576
SYS_SW_IRAMUnavailable	512000
SYS_SW_SamplesPerHWI	512
SYS_SW_SamplesPerMFrame	1953
SYS_SW_SymbolsPerMFrame	234
SYS_SW_CPURate	1000000000
SYS_SW_EmifClockRate	100000000
SYS_SW_EmifCPUCyclesPerWord	3
SYS_SW_SymbolRate	23400
SYS_SW_DucRate	195312.5

to do so, we need to gather necessary variables to solve Equation 4.13. We gather a list of software input constraints that will be used for the computation, listed in Table 6.7. These parameters are *inputs* to the CDE in the software constraint file described in Section 6.1 and will change for different platform architectures, but the CDE method remains the same. We also take the values from Tables 6.5 and 6.6 to generate the software utilization file shown in Table 6.8. With these parameters established, the CDE can now begin to solve for the various components needed in Equation 4.16, repeated here to help the reader follow the text.

$$N_{\substack{max \\ active \\ mods}} = floor\left[\frac{T_{\substack{duc \\ period}}}{\beta}\right]$$

$$\beta = \left(\frac{T_{mf}}{n_{\substack{mf \\ samp}}}\right) + \left(\frac{2\hat{t}_{\substack{hwi \\ isr}} + t_{emif}}{n_{\substack{dma \\ samp}}}\right)$$

Focusing first on β, the first variable we solve for is T_{mf}, the time duration between subsequent calls to the modulator frame. From 4.2, and that our CPU clock is SYS_SW_CPURate 1.0GHz we obtain

Table 6.8: Software Utilization File Values

Parameter	**Value**
CycReq_MFStageNames	{'Bookkeeping','Crc','Encode','Interleave','Scramble','Modulate','Resample'}
CycReq_MFUsed	[2100,12000,28000,55000,23225,19000,509000]
CycReq_Overhead	50000
CycReq_DucHwiIsr	12669

$$\begin{aligned} T_{mf} &= t_{mf} + t_{ov} \\ &= 648.325 + 50.000 \ \mu\text{sec} \\ &= 698.325 \ \mu\text{sec} \end{aligned} \tag{6.1}$$

Refering to Table 6.5 we see that profiling the target platform provided 12669 cycles to execute the DUC HWI ISR. This number, when profiled reflects both the cycle requirements of the ISR and the penalty incurred for setting up the DMA transfer, as this is done inside the ISR. Therefore we can use this number as an input and set

$$\begin{aligned} \hat{t}_{\substack{hwi \\ isr}} &= \frac{12669}{1.0 \text{ GHz}} \\ &= 12.669 \ \mu\text{sec} \end{aligned} \tag{6.2}$$

Next we solve for t_{emif}, the time spent transfering samples across the EMIF that interfaces the software and hardware components on the target platform. From Table 6.7, SYS_SW_SamplesPerHWI = 512 tells us that we have 512 complex samples that will be transfered across the EMIF for every DUC HWI. Since a complex sample is made up of a 16-bit I and 16-bit Q[1], and the EMIF is a 16-bit data bus, transfering one complex sample requires two transfers. On the target platform SYS_SW_EmifClockRate = 100 MHz and SYS_SW_EmifCPUCyclesPerWord = 3, corresponding to f_{emif} and $n_{\substack{cyc \\ word}}$ in 4.3. So for n

[1]I,Q sample pair stands for the Inphase and Quadrature components of the complex sample pair.

= 1024 words (i.e. 512x2 I,Q complex samples) to be transfered, we have

$$\begin{aligned} t_{emif} &= (n[words]) \left(n_{\substack{cyc \\ word}} \right) \left(\frac{1}{f_{emif}} \right) \\ t_{emif} &= (1024)(3) \left(\frac{1}{100e6} \right) \\ &= 30.72\ \mu\text{sec} \end{aligned} \tag{6.3}$$

The remaining variables can be obtained from Table 6.7: $n_{\substack{mf \\ samp}} = 1953$, and $n_{\substack{dma \\ samp}} = 512$. Likewise for the DUC period we have

$$\begin{aligned} T_{\substack{duc \\ period}} &= \frac{1}{\text{SYS\_SW\_DucRate}} \\ &= \frac{1}{195312.5} \\ &= 5.12\ \mu\text{sec} \end{aligned} \tag{6.4}$$

Everything needed has been obtained to solve for the number of active modulators that the existing configuration can support without choking the software side of the platform. First we solve for the β value, and then $N_{\substack{max \\ active \\ mods}}$.

$$\begin{aligned} \beta &= \left(\frac{T_{mf}}{n_{\substack{mf \\ samp}}} \right) + \left(\frac{2\hat{t}_{\substack{hwi \\ isr}} + t_{emif}}{n_{\substack{dma \\ samp}}} \right) \\ &= \left(\frac{698.325\text{e-}6}{1953} \right) + \left(\frac{2 * 12.669\text{e-}6 + 30.72\text{e-}6}{512} \right) \\ &= 0.467\ [\mu\text{sec/samp}] \end{aligned} \tag{6.5}$$

$$
\begin{aligned}
N_{\substack{max \\ active \\ mods}} &= floor\left[\frac{T_{\substack{duc \\ period}}}{\beta}\right] \\
&= floor\left[\frac{\text{5.12e-6}}{\text{0.467e-6}}\right] \\
&= floor\left[10.96\right] \\
&= 10 \qquad (6.6)
\end{aligned}
$$

We now have a predicted maximum number of modulators that can be active simultaneously in the software component, from a cycle requirement perspective. Now we must calculate how many modulators will be allowed from a memory standpoint to ensure that we have enough memory resources to run all of these modulators at the same time. Using SYS_SW_TotalIRAM, SYS_SW_IRAMUnavailable, and SYS_SW_IRAMUsagePerMod from Table 6.7 with 4.17 we obtain

$$
\begin{aligned}
N_{\substack{mods \\ mem \\ limit}} &= floor\left[\frac{MEM_{iram} - MEM_{unavail}}{MEM_{mod}}\right] \\
&= floor\left[\frac{1048576 - 512000}{32768}\right] \\
&= floor\left[16.38\right] \\
&= 16 \qquad (6.7)
\end{aligned}
$$

From these results we see that the Cycle requirements allows 10 modulators, while there is enough room in software memory to host 16. The minimum of these two, as dictated by Equation 4.18, states that the baseline implementation allows for 10 modulators in the software component. In other words, 10 software portions of the full communications

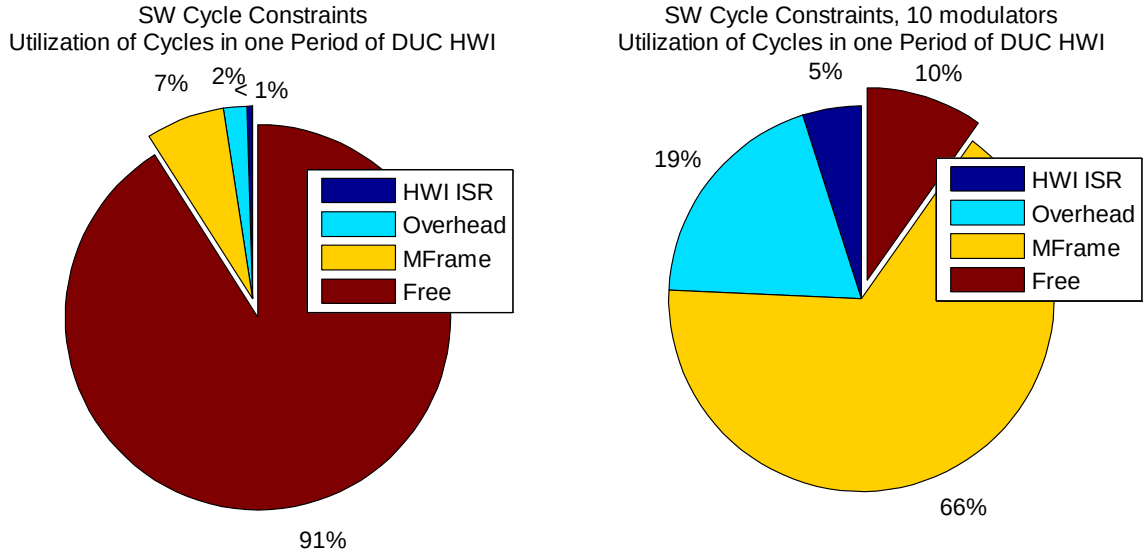

Figure 6.4: Cycle Constraints for both one modulator and multiple modulators during one period of DUC HWI. The modulator frame has been normalized to the amount of the function that needs to complete before the next DUC HWI.

channel can be instantiated on the target platform.

$$\begin{aligned} N_{\substack{SW \\ mods}} &= min\left[N_{\substack{max \\ active \\ mods}}, N_{\substack{mods \\ mem \\ limit}}\right] \\ &= min\left[10, 16\right] \\ &= 10 \text{ SW modulators} \end{aligned} \tag{6.8}$$

Figure 6.4 visually represents this maximum. In both of the charts in the figure, 100% of the pie chart is one full period of the DUC HWI, or $\hat{T}_{\substack{next \\ hwi}}$ as described previously. To clarify, every time one full cycle of the pie chart has been executed (the total number of cycles of the piechart has passed), the DUC will send an interrupt trigger to the software CPU requesting more samples to be transferred across the EMIF (refer to Figure 4.1). Since

the modulator frame generates 1953 samples every time it executes, and the DUC HWI ISR only transfers 512, the modulator frame does not need to execute every time the DUC HWI triggers. However, to visual inspect things, we can normalize the modulator frame by one cycle of $\hat{T}_{\substack{next \\ hwi}}$, visually spreading its execution over multiple DUC HWIs. Then we can take up portions of the pie chart in Figure 6.4 with the portion of the modulator frame that needs to execute every $\hat{T}_{\substack{next \\ hwi}}$ period. The left chart in the figure shows the portion of $\hat{T}_{\substack{next \\ hwi}}$ taken up by a single modulator, while the right chart in the figure shows the portion taken by 10 modulators.

One can see that 10 modulators consumes almost all the free time available in the $\hat{T}_{\substack{next \\ hwi}}$ period. Although there is 10% left, the magnitude of this additional 10% is most likely affected by the precision of the models used for overhead increase, as discussed at the end of the Software Analysis chapter. Therefore, as the results before flooring showed that we could fit 10.96, or almost 11, this free time appears almost large enough to fit one more modulator in, when in actuality it would be a poor design decision to attempt to squeeze in one more modulator at the cost of using almost 100% of the cycles available from the CPU. Doing so would most likely cause unknown behavior from the CPU, as it would be choked for cycles, with no room for flexibility for asynchronous event handling that may arise unexpectedly. The flooring step in Equation 6.6 is used to take into account not only the fact that you can only have an integer number of modulators running, but also plays a conservative roll in the overhead calculation control[2].

SW Verification

Verifying these predicted CDE results was as easy as running the application on the target platform, and increasing the number of active communications channels until the platform reported underflows and CPU stalls. After running this test we did indeed verify that after 10 active modulators, the software CPU began to report CPU stalls and buffer underflows

[2]Or other model assumptions that may have been made during profiling.

Table 6.9: CDE Hardware Constraint File Values

Parameter	Value
SYS_HW_DevUtilThresholdPerc	90
SYS_HW_MultiFactorIncrease	1.1253
SYS_HW_useLinearScaledModel	1

Table 6.10: CDE Hardware Utilization File Values

Parameter	Value
DevUtil_ResourceNames	{'Slices','SliceFF','LUT','BRAM','DSP48'}
DevUtil_ResourcesAvailable	[24576,49152,49152,320,512]
DevUtil_FPGABaseBuild	[1974,2597,3212,11,16]
DevUtil_SingleDUC	[1165,1881,812,6,21]
DevUtil_SingleResampler	[423,513,488,8,1]

as it could not keep the transmit buffer full enough to keep the DUC FIFO from running dry.

Hardware: Max capacity

Now that the maximum number of software portions of the communications channel is known, the designer needs to find out how many hardware portions fit in the baseline implementation of the hardware component of the target platform, an FPGA. The results of this section will not only show how many are currently in the baseline implementation, but also what level of chip utilization the FPGA is currently at, giving the designer the first glimpse at whether or not there is room for more.

Just like in the software portion, we define a table of parameters that are provided as inputs to the CDE. There values are listed in Table 6.9 and make up the hardware constraints file described in Section 6.1. Likewise Table 6.10 is used for the hardware utilization file.

Using these files and Equation 5.2, repeated here to help the reader follow the text, the CDE iteratively increases n in 5.2 until one of the key resources listed in Table 5.1 breaches the provided device utilization threshold. The threshold we used can be seen from Table 6.9 to be 90%.

$$DevUtil_{n-blocks} = DevUtil_{fpga-bb} + (n * \alpha)BlockUtil$$

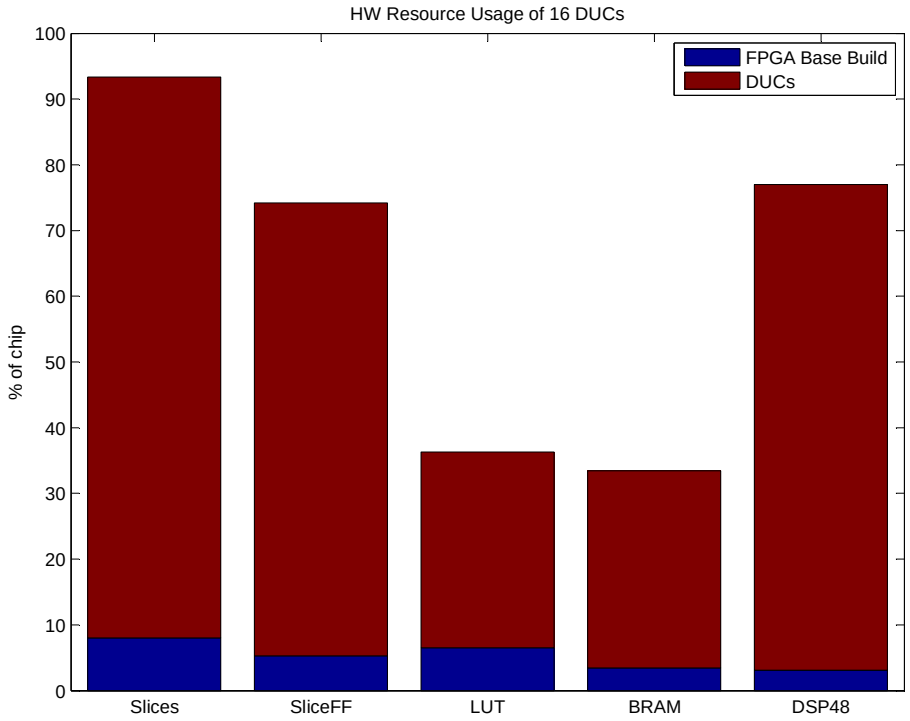

Figure 6.5: HW utilization, baseline configuration

Figure 6.5 shows that the CDE was able to fit 16 DUCs into the original configuration after just breaching the 90% threshold. The figure is a stacked bar for each of the key parameters from Table 5.1. As the implementation requires more of a resource, the bar chart grows upwards increasing device utilization. The FPGA Base Build provided as input to the CDE takes up less than 10% of the FPGA. With 16 instantiations of the DUC, using the linear scaled method with the given α, utilization is at about 93% for slices.

This tells us that with no changes to the current implementation (i.e. no movement of blocks from software to hardware), there is an upper limit on the system of 16 DUCs that could ever be instantiated in the FPGA. Also note that the FPGA Utilization is already at a high level when instantiating 16 DUCs, not leaving much room for moving anything from software to hardware. This might tell the design engineer that if they revisit their DUC implementation, they may be able to free up some space on the FPGA for codesign work.

Table 6.11: Actual Utilization Error: Baseline Configuration, α=1.1253.

12 DUCs	
Resource	**Error**
CLB Slices	-0.1%
Flip Flop Slices	-0.8%
Look Up Tables	-0.2%
Block RAM	0.0%
DSP48	-2.4%

HW Verification

To verify that the predicted utilization results, a configuration with 16 DUCs was built with Xilinx ISE for the Virtex4 SX55 FPGA. The resulting error between the predicted and actual utilization is shown in Table 6.11. This error is small enough to show adequate accuracy verification in the device utilization prediction using the linearly scaled increase model shown in Equation 5.2. Note that this is for an FPGA configuration built with DUCs, which was the same block in the FPGA that was used to solve for the value of $\alpha = 1.1253$ used. It will be seen that for the improved configuration, this α value is not as accurate, but still useful.

Results from SW and HW (Number of channels)

With a software upper limit of 10 modulators, and a hardware upper limit of 16 DUCs, and the fact that a communications channel requires a single software modulator and hardware DUC per channel, the capacity of the baseline implementation is 10 communications channels. The next section describes the CDE improved analysis and solution, and then verifies the results.

$$\begin{aligned} N_{chan} &= min\left[N_{\ mods}^{sw}, N_{ducs}^{hw}\right] \\ &= min\left[10, 16\right] \\ &= 10 \text{ channels} \end{aligned} \tag{6.9}$$

6.2.2 Improved Implementation

The baseline software analysis led the designer to view the resampler as an unbalanced stage in the modulator frame, making it a likely candidate for movement out of the software component and into the hardware side. Likewise, the baseline hardware analysis showed the designer that the FPGA had enough room to instantiate much more than the 10 the software was limited to. Therefore, a middle ground could be met somewhere between 10 and 16 by shifting the Resampler stage from software to hardware.

The approach here is to follow the same path as in the baseline implementation, changing pertinent inputs to the CDE along the way. The parameters that need to be adjusted will be highlighted. To recompute β we note that only the first part of the β equation has changed, the portion involving the modulator frame work. The second portion which deals with the transfer of data across the interface does not change, as the number of samples transferred per HWI does not change in this configuration[3].

Software: Improved max capacity

Since the resampler has been moved into the hardware portion of the platform, the software component no longer has to provide a sample stream for transfer across the EMIF. Now the modulator provides a *symbol stream* because the input rate of the resampler is the symbol rate, in this case 23400 symbols/sec, and so the DUC's input FIFO will drain much slower. Therefore the time between DUC HWI's will increase, providing more cycles for the CPU in the software component to do work. Therefore we compute a new DUC period as

[3]Although the design engineer could definitely adjust the number of samples that are transferred to achieve a balance of work generating data versus transferring it.

$$
\begin{aligned}
T_{\substack{duc \\ period}} &= \frac{1}{\text{SYS\_SW\_SymbolRate}} \\
&= \frac{1}{23400} \\
&= 42.735\ \mu\text{sec} \qquad (6.10)
\end{aligned}
$$

The only difference to T_{mf} is that the resampler now no longer exists as part of the software chain of stages. Therefore the time required for the Resampler is backed out of the total time for the modulator from Table 6.6. The new modulator frame with its existing stages is shown in Figure 6.6. It becomes apparent that the modulator frame's stages are now well balanced, meaning there is not one stage that far outweighs any of the other stages in terms of cycle requirements.

$$
\begin{aligned}
T_{mf} &= t_{mf} + t_{ov} - t_{resamp} \\
&= 648.325 + 50.000 - 509.000 \\
&= 189.325\ \mu\text{sec} \qquad (6.11)
\end{aligned}
$$

We can now recompute the β value using the number of *symbols* generated each modulator frame for the modulator work portion, and then a new $N_{\substack{max \\ active \\ mods}}$.

$$
\begin{aligned}
\beta &= \left(\frac{T_{mf}}{n_{\substack{mf \\ samp}}}\right) + \left(\frac{2\hat{t}_{\substack{hwi \\ isr}} + t_{emif}}{n_{\substack{dma \\ samp}}}\right) \\
&= \left(\frac{189.325\text{e-}6}{234}\right) + \left(\frac{2 * 12.669\text{e-}6 + 30.72\text{e-}6}{512}\right) \\
&= 0.919\ [\mu\text{sec/samp}] \qquad (6.12)
\end{aligned}
$$

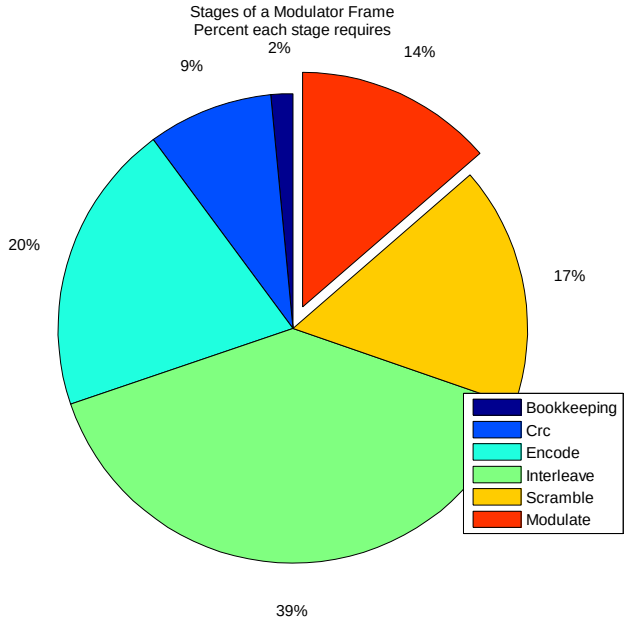

Figure 6.6: Improved modulator frame stages. 100% of the pie is the entire modulator frame cycle usage.

$$
\begin{aligned}
N_{\substack{max \\ active \\ mods}} &= floor\left[\frac{T_{\substack{duc \\ period}}}{\beta}\right] \\
&= floor\left[\frac{42.735\text{e-}6}{0.919\text{e-}6}\right] \\
&= floor\left[46.50\right] \\
&= 46 \qquad (6.13)
\end{aligned}
$$

These results show us that by moving the resampler into hardware, we increased the time between DUC HWIs and also decreased the time required to generate data to feed the DUC. This provided a $36/10 = 360\%$ increase over the baseline version in terms of available cycles for active software modulators. However, the memory of the system has not changed from the baseline to improved version, therefore the result from the previous section regarding the number of modulators that can simultaneously fit into the memory

available to the software component is still 16. Therefore, the minimum of these two still dictates how many software modulators can run.

$$\begin{aligned} N_{\substack{SW \\ mods}} &= min\left[N_{\substack{max \\ active \\ mods}}, N_{\substack{mods \\ mem \\ limit}}\right] \\ &= min\,[46, 16] \\ &= 16 \text{ SW modulators} \end{aligned} \tag{6.14}$$

With this in mind, the CDE predicts a $6/10 = 60\%$ increase in capacity in terms of the software component portion of the communications channel. Figure 6.7 shows the increase visually over Figure 6.4. All the required software tasks take just a small sliver of the time provided in one period of the DUC HWI. This is due not only to the modulator frame taking less samples, but that the DUC front end has been replaced by the resampler, dropping the input rate to the DUC to the symbol rate instead of the sample rate. The plot on the right side of Figure 6.7 tells the designer that the upper limit is not restricted by cycle requirements anymore, but instead by the memory capacity of the software portion of the codesign platform.

SW Verification

By removing the resampler cycles from the modulator frame, cycles were were shown to not be a limiting factor anymore through use of a TI cycle accurate simulator. At compile time, the number of modulators that would fit in memory was limited to 16. This is because the compiler references the memory map and linker file which describes the memory available to the system. The number of modulator transmit buffers had to be limited to 16 for compilation to proceed without resource errors being generated. Therefore the CDE correctly predicted a huge increase in modulators based off of cycles, and a true upper limit of 16 based off of memory resources available.

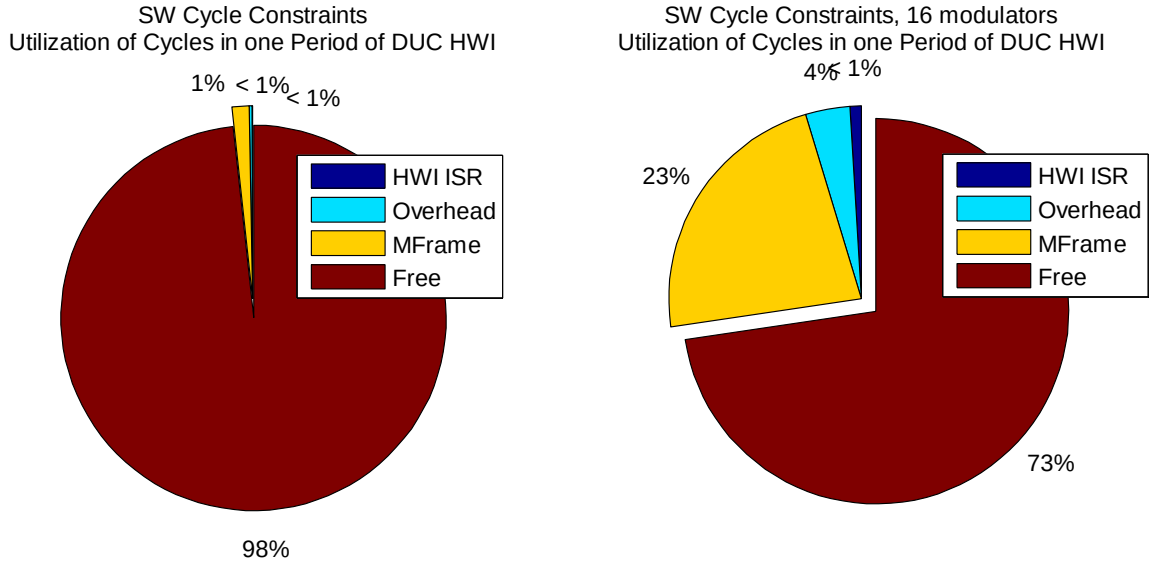

Figure 6.7: Cycle Constraints for both one modulator and multiple modulators during one period of DUC HWI. The modulator frame has been normalized to the amount of the function that needs to complete before the next DUC HWI.

Hardware: Improved max capacity

Following the same order as in the baseline implementation analysis, the next step is to determine how many hardware portions of the communications channel can fit in the FPGA. One of the inputs to the CDE is the utilization requirements of a Resampler block. This is provided in the hardware utilization file and the value for this case-study is shown in Table 6.10. Note that different implementations will provide different utilization numbers. For the purpose of this research, these numbers are seen as inputs to the CDE[4]. For comparison, Figure 6.8 shows the utilization requirements for the components of interest in the FPGA.

Once the utilization requirements of the additional block are provided, the same iterative process is employed as the baseline implementation, just with a larger requirement per each iteration to make room for the resampler as well. The results, shown in Figure 6.9, depict that with the addition of the resampler block, the CDE could only fit 12 full communications

[4]After running through a few iterations of using the CDE tool, the designer may choose to revisit their design to change these numbers, if the desire to manipulate the results of the codesign analysis.

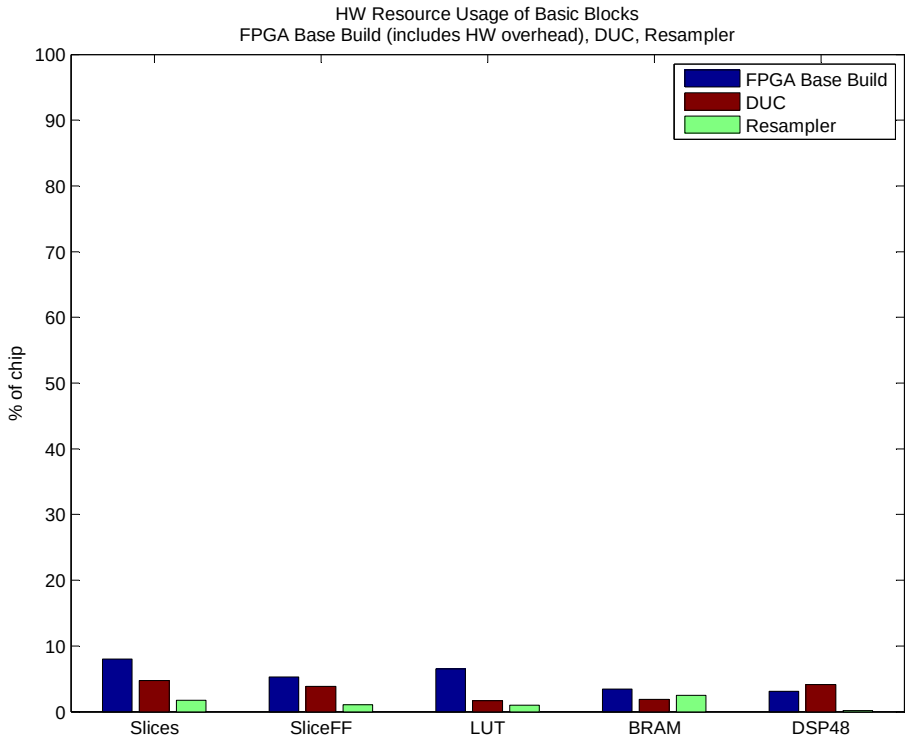

Figure 6.8: Utilization requirements of basic blocks in FPGA.

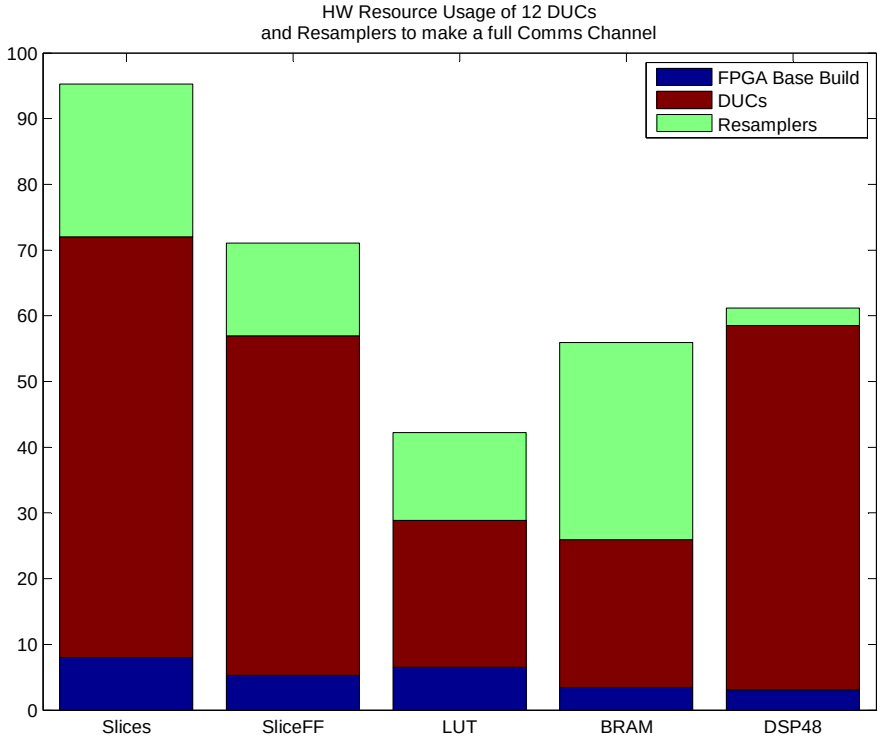

Figure 6.9: HW utilization, improved configuration. The FPGA is getting very close to 100% utilization.

paths in the FPGA versus 16 without the resampler.

HW Verification

The results concerning the number of DUCs and Resamplers that can fit in the FPGA were verified by building a configuration with the 12 instantiations. It was verified that the FPGA did in fact fit 12 DUCs and Resamplers inside it, at a high utilization. However, the accuracy of the utilization was not perfect, bringing the error in prediction back to the order of 10% more than actually used, shown in Table 6.12, questioning the benefit of the scaled value. Therefore, the CDE, using the linearly scaled utilization model, with an α of 1.1253, provided a stricter utilization requirement than what the Xilinx ISE tool-set was able to produce for the DUC/Resampler instantiation pair. This is mainly due to the limits in accuracy of using an α solved for a configuration with only DUCs instantiated, for a

Table 6.12: Predicted Utilization Error: Improved Configuration, α=1.1253.

12 DUCs and Resamplers	
Resource	**Error**
CLB Slices	-11.4%
Flip Flop Slices	-7.7%
Look Up Tables	-4.8%
Block RAM	0.0%
DSP48	-0.6%

prediction with DUCs and a new block (i.e. the Resampler). Also, the Xilinx toolset takes an iterative approach in the FPGA, using circuit reuse to optimize portions of the FPGA design. The creation of a *finer* resolution utilization prediction model is left for follow on efforts to this work.

Improved Results from SW and HW (Number of channels)

Moving the resampler into the hardware component allowed for 6 additional channels to be placed inside the software, while decreasing the total channels in the hardware component by 4. This tradeoff, however, allows the system to include two additional channels total, since previously the software component was limited to 10 simultaneous channels. Therefore, the designer would move forward with an implementation on the codesign platform for a channel capacity of 12, providing a 2/10 = 20% increase from the baseline design, as summarized in Table 6.13.

$$\begin{aligned} N_{chan} &= min\left[N_{\substack{sw \\ mods}}, N_{ducs}^{hw}\right] \\ &= min\left[16, 12\right] \\ &= 12 \text{ channels} \end{aligned} \tag{6.15}$$

Table 6.13: Results, Baseline and Improved Implementations

	SW Limit	MEM Limit	HW Limit	Num Channels
Baseline	10	16	16	10
Improved	46	16	12	12
Channel capacity increased by				*20%*

6.3 Generalized Case-Study

Since the previous case-study was application specific to a pre-existing SDR configuration, we decided to employ a generalized set of inputs to the CDE and observe the analysis. We created some general constraints and utilization files, and modified the CDE to work with the more general case. Since we are not working with a modulator specifically, we call our equivalent modulator frame a *block under test*. This section describes our findings.

6.3.1 Generalizing the CDE

To generalize the CDE so that it can work with a the general case-study, we had to modify the software utilization file by adding two additional arrays. The first array holds the number of data that are generated by each stage of the block under test, while the second array provides the data-rate at the output of each stage. It is useful to have this information because as the CDE moves portions of the software block into hardware, it can adjust automatically for the different data rates that are now moving across the HW/SW interface[5].

```
CycReq_BlockStageNames  = {'STG1','STG2','STG3','STG4'};
CycReq_BlockStagesUsed  = [2100,30000,28000,30000];
CycReq_BlockStageDataOut= [100,100,100,650];
CycReq_BlockStageRateOut= [80000,80000,100000,800000];
CycReq_Overhead         = 50000;
CycReq_HwiIsr           = 12669;
```

Source 4: A generalized software utilization file.

An example change to the source code is shown in Source 6, which shows how the output

[5]Note that a different output data-rate of a stage does not effect the EMIF transfer rate, just the rate between the last SW stage and the first HW stage draining its input FIFO.

data rate is now a function of which stage of the block under test is the last one before the HW/SW interface. Source 4 shows our modified utilization file with the new arrays. The stages of the generic block are labeled 'STG1' through 'STG4.' The number of data and the data-rate values were then deleted from the software constraints file, provided in Source 5.

```
SYS_SW_OverheadIncrPerBlock     = 50000;
SYS_SW_IRAMUsagePerBlock        = 32768;
SYS_SW_TotalIRAM                = 1048576;
SYS_SW_IRAMUnavailable          = 512000;
SYS_SW_SamplesPerHWI            = 512;
SYS_SW_CPURate                  = 1000000000;
SYS_SW_EmifClockRate            = 100000000;
SYS_SW_EmifCPUCyclesPerWord     = 3;
```

Source 5: A generalized software constraints file.

```
% ------------------------------------------------------------
% Get Effective Sample Per HWI
% ------------------------------------------------------------
% Now that we have the EMIF transfer time we can calculate how many samples
% drain from the Block FIFO during the HWI ISR overhead to get an effective
% sample count added to the DUC FIFO on each HWI call.  We convert all
% variables to [sec] to ensure matching units during the calculation.
% This is important because some parts run at different clock rates, e.g,
% the EMIF runs at a different rate than the HWI ISR code.
SW_TimePerHwiIsr = CycReq_HwiIsr/SYS_SW_CPURate;
SW_NumSamplesDrained =
  (SW_TimePerHwiIsr+SW_EmifTransferTime)*CycReq_BlockStageRateOut(idx_outputStage);
SW_NumEffectiveSamplesPerHWI = SYS_SW_SamplesPerHWI - SW_NumSamplesDrained;
% ------------------------------------------------------------
% Get Time between HWI's, T_next_hwi
% ------------------------------------------------------------
% Now that the effecgive sample per HWI has been calculated, since we know
% the Rate at which the DUC drains its DATA FIFO, we can then calculate
% how often the DUC Data FIFO 1/2 empty HWI will trigger the Software DSP
% chip.
SW_TimeNextHWI =
  SW_NumEffectiveSamplesPerHWI / CycReq_BlockStageRateOut(idx_outputStage);
```

Source 6: Portion of CDE source code modified for general case output data-rates.

6.3.2 Difference from Resampler Case-study

The same equations and procedure outlined in Section 6.2 for the Resampler case-study are followed in the general case. In order to produce a distinction between the two case-studies that makes this analysis worth while, a difference was introduced in the proposed system for the general study. After 'STG4' is moved from software to hardware, the number of data generated by 'STG3' will be half as many than the number of data transferred across the EMIF each HWI as in the previous case-study (i.e. 100 versus 234). Therefore, the block under test will need to be called *multiple times* between each HWI to ensure there are enough samples in the transmit buffers, so that the hardware blocks do not run dry. Something to keep in mind is that the overhead must execute between each call to the block under test as that is how the model described it.

Also, as can be seen from Figure 6.10, there is a much better balance of stages in the block under test than in the previous case-study, where the Resampler stuck out as a backend heavy stage to the software modulator.

Since at this time the hardware portion of the CDE is just a prediction of utilization, no changes besides variable names where made to the hardware portion of the CDE code. Because of this, an analysis of the hardware portion of a generic example is not included here, since it would just be a repeat of analysis presented in Section 6.2 with different numbers, but no substantially different scenario. Therefore, the baseline and improved implementation for the software portion of the generic case-study are discussed in the remaining portion of this section, and the hardware is assumed as a non-limiting factor in this example.

6.3.3 Baseline Implementation

With the software component requiring all for stages to execute (i.e. STG1 to STG4), the cycle portions for each stage of the block under test are shown in Figure 6.10. With this configuration, Figure 6.11 shows that CDE predicts a capacity of 3 simultaneous blocks. Note that the right plot appears to show that there is room for 4 blocks. This is due to the *floor* operator in Equation 4.16. Using the floor operator provides safety in prediction

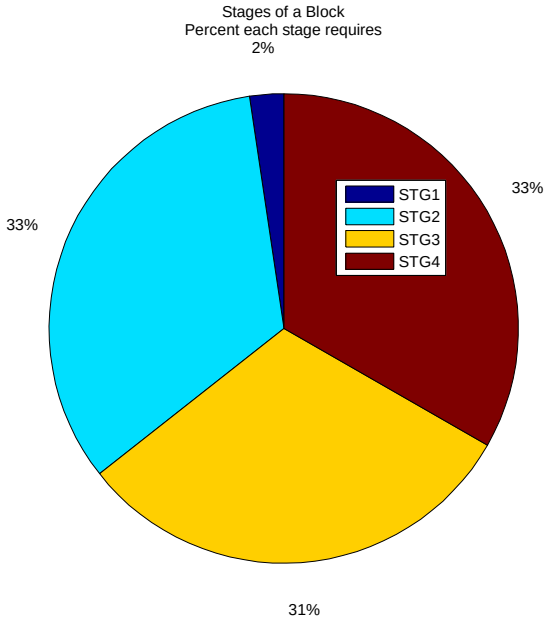

Figure 6.10: Cycle requirements of the individual stages of general block under test.

of cycle requirements. A possible solution would be to use a rounding operator instead. However, at this point the designer could infer a closer estimate by human reasoning.

6.3.4 Improved Implementation

Just as in the previous case-study, the CDE moves the last stage of the software block, the one assumed to be on the HW/SW interface boundary, out of the computation by assuming its implementation in hardware. Once again the same methods are followed that were discussed fully in Section 6.2, and the results are shown in Figure 6.12. After moving 'STG4' into hardware, the CDE predicts a capacity of 8 simultaneous blocks.

Verification and Results

Since this is a generic example, without a hardware platform to verify on, we show a verification by a logical intuitive method. First, we can calculate how much time is required to execute the block under test without 'STG4' using the provided clock frequency of 1 GHz as

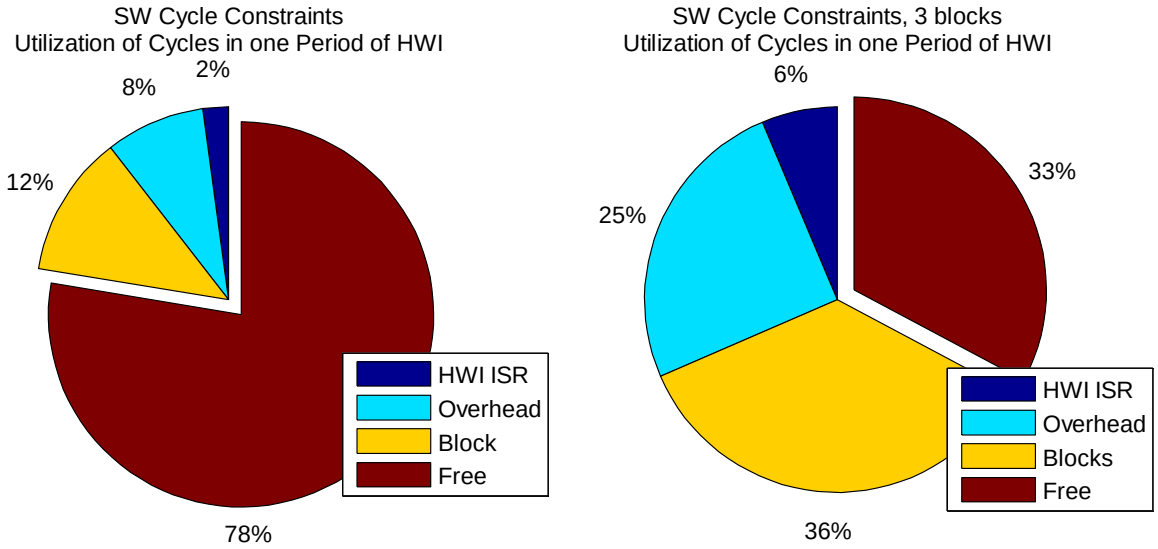

Figure 6.11: Cycle Constraints for both one block and multiple blocks under test during one period of HWI, Baseline Implementation.

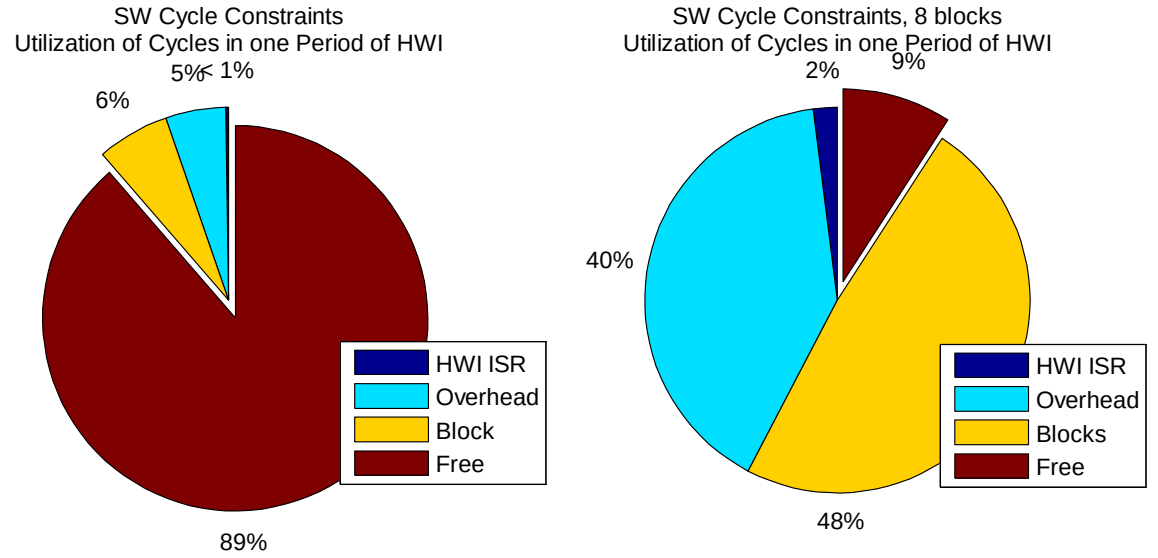

Figure 6.12: Cycle Constraints for both one block and multiple blocks under test during one period of HWI, Improved Implementation.

$$
\begin{aligned}
t_{block} &= \frac{STG1 + STG2 + STG3}{f_{clk}} \\
&= \frac{2100 + 30000 + 28000}{1GHz} \\
&= 60.10\ \mu\text{sec} \qquad (6.16)
\end{aligned}
$$

Next, realizing that STG3 will generate 100 data out, we can determine how many times the block under test must execute between HWIs as

$$
\begin{aligned}
N_{\substack{block \\ calls}} &= ceil\left[\frac{n_{\substack{data \\ hwi}}}{n_{\substack{data \\ block}}}\right] \\
&= ceil\left[\frac{512}{100}\right] \\
&= ceil[5.12] \\
&= 6 \text{ times} \qquad (6.17)
\end{aligned}
$$

Recall that the overhead is all the other cycles that must be spent inbetween subsequent calls to the block under test. Therefore we can add the overhead in to get essential a total time between block calls[6] for our block as

$$
\begin{aligned}
T_{block} &= t_{block} + t_{ov} \\
&= 60.10 + 50.00\ \mu\text{sec} \\
&= 110.10\ \mu\text{sec} \qquad (6.18)
\end{aligned}
$$

[6]This is the equivalent to T_{mf}.

Multiplying this number by the times it must execute gives

$$\begin{aligned} \hat{T}_{block} &= T_{block} * N_{\substack{block \\ calls}} \\ &= (110.10\ \mu\text{sec}) * 6 \\ &= 660.60\ \mu\text{sec} \end{aligned} \tag{6.19}$$

Which is how much time is required for each simultaneous block under test instantiation to execute to ensure that the hardware does not run dry on data. Now we compute a rough time between the HWI that will pull new data from the software component to be transferred across the EMIF to the hardware side.

$$\begin{aligned} t_{hwi} &= n_{\substack{data \\ hwi}} * f_{\substack{stg3 \\ data-rate}} \\ &= 512 * 100000 \\ &= 5.12\ \text{msec} \end{aligned} \tag{6.20}$$

Now we can do an approximation of how many blocks under test can fit in one period of the HWI.

$$\begin{aligned} N_{blocks} &= \frac{t_{hwi}}{\hat{T}_{block}} \\ &= \frac{5.12\ \text{e}^{-3}}{660.60\ \text{e}^{-6}} \\ &= 7.75 \approx 8\ \text{blocks} \end{aligned} \tag{6.21}$$

By using this rough logical approach we were able to arrive at approximately the same figure as the CDE, using some of the same method, but not worrying about all of the

Table 6.14: Results, Generic Baseline and Improved Implementations

	SW Limit	MEM Limit	HW Limit	Num Channels
Baseline	3	16	-	3
Improved	8	16	-	8
		Block capacity increased by		*167%*

specifics, such as the HW/SW interface transfer portion. Because of this, the result we have achieved by our calculations here is even more coarse grained than the CDE results, but has shown a close enough solution to the CDE to provide sufficient verification of the CDE solution.

Results (Number of Blocks)

Table 6.14 summarizes the CDE output for the generic case study. Recall that the hardware was assumed to be a non-limiting factor for this case-study. Also, the memory requirements will change as well depending on the platform. The values used here were chosen to be the same as the polyphase resampler case study. An increase from 3 to 8 blocks provided a block capacity increase of $5/3 = 167\%$. The main focus of this section was to show the small level of effort required to generalize the CDE method presented in this work to other applications.

Chapter 7: Conclusion

The purpose of this research was to apply a new focus to SDR HW/SW codesign, that of increasing the channel capacity of a codesign SDR platform versus minimizing the latency of a single channel. In other words, maximizing the number of channels simultaneously active on an SDR board. By following the method outlined in Section 1.2, we arrived at models that predicted constraint requirements and resource utilization for the SW and HW components, respectively. These models were used to create a CDE that ran in MATLAB which predicted the channel capacity of the ArgonST ITC platform for a baseline configuration, and then an improved configuration which involved moving portions of the physical layer implementation from SW into HW. The CDE results were verified by observing actual performance of the platform, proving the reliability of the method, and general derived equations.

Future Work

This work focused on predicting and increasing channel capacity based on key resource utilization on the codesign platform. Future work could include additional factors such as power requirements, flexibility and maintainability of design, and also cost and schedule required for design implementation as inputs into the codesign process. This would allow the optimization techniques to be spread across multiple criteria.

Additional follow on research could include investigating models that provide greater prediction accuracy in hardware device resource utilization, building tables of general resource requirements for basic components that are used in communications applications to provide a database that the CDE can access for suggestions, and testing the generalized equations derived in the analysis chapters against a matrix of different platforms and

applications including more than just SDRs. Future versions of the CDE tool should autonomously be able to determine where to place the HW/SW interface barrier based off of a chosen algorithm and platform, while still focusing on maximizing the channel capacity.

Bibliography

Bibliography

[1] Theerayod Wiangton, Peter Y.K. Cheung, and Wayne Luk, *Hardware/Software codesign: A systematic approach targeting data-intensive applications*. IEEE Signal Processing Magazine, May 2005, pp.14-22.

[2] T. Wiangton, P.Y.K. Cheung, W. Luk *Multitasking in hardware-software codesign for reconfigurable computer*. Proceedings of the 2003 International Symposium on Circuits and Systems, ISCAS '03, Volume 5, May 2003, pp.V621-V624.

[3] Asawaree Kalavade, Edward A. Lee, *A hardware-software codesign methodology for DSP applications*. IEEE Design and Test of Computers, September 1993, pp.16-28.

[4] Christopher Hylands, et al., *Overview of the Ptolemy project*. Department of Electrical and Computer Science, University of California, Berkeley, California, July 2003.

[5] U.C. Regents, *POLIS: A framework for hardware/software co-design of embedded systems*. Center for Electronic Systems Design, Department of Electrical and Computer Science, University of California, Berkeley, California, 2000.

[6] Roel Meeuws, Y. Yankova, K. Bertels, G. Gaydadjiev, S. Vassiliadis, *A quantitative prediction model for hardware/software partitioning*. IEEE Field Programmable Logic and Applications Conference August 2007, pp.735-739.

[7] Stamatis Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, E.M. Panainte *The MOLEN polymorphic processor*. IEEE Transactions on Computers, Volume 53-11, November 2004, pp.1363-1375.

[8] Soonhoi Ha, C. Lee, Y. Yi, S. Kwon, Y.P. Joo, *Hardware-software codesign of multimedia embedded systems: The PeaCE approach*. 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2006, pp.207-214.

[9] Ahmed Jerraya, W. Wolf, *Hardware/Software interface codesign for embedded systems*. IEEE Computer Society, Computer, Volume 38-2 February 2005 pp.63-69.

[10] Patrice Gerin, H. Shen, A. Chureau, A. Bouchhima, A. Jerraya, *Flexible and executable hardware/software interface modeling for multiprocessor SoC design using SystemC*. IEEE Computer Society, ASP-DAC '07: Proceedings of the 2007 conference on Asia South Pacific design automation, 2007, pp.390-395.

[11] Pierre-Andre Mudry, G. Zufferey, G. Tempesti, *A dynamically constrained genetic algorithm for hardware-software partitioning*. ACM, GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation, 2006, pp.769-776.

[12] Jurgen Helmschmidt, E. Schuler, P. Rao, S. Rossi, S. di Matteo, R. Bonitz, *Reconfigurable signal processing in wireless terminals*. Proceedings of The Conference on Design, Automation and Test in Europe, Designer's Forum Volume 2, 2003, pp.20244-20249.

[13] Tim Hentschel and Gerhard Fettweis, *Sample rate conversion for software radio*. IEEE Communications Magazine, August 2000, pp.142-150.

[14] John L. Hennessy and David A. Patterson, *Computer architecture: A quantitative approach*. Morgan Kaufmann, California, 4th Edition, 2007, pp.19-25.

[15] Fei Xie, X. Song, H. Chung, R. Nandi, *Translation-based co-verification*. Proceedings of the 2nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design, 2005, pp.111-120.

[16] Fei Xie, G. Yang, X. Song, *Component-based hardware/software co-verification*. Proceedings of the Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006, pp.27-36.

[17] Gabriela Nicolescu, S. Yoo, A. Bouchhima, A. A. Jerraya, *Validation in a component-based design flow for multicore SoCs*. Proceedings of the 15th international ACM symposium on System Synthesis, 2002, pp.162-167.

[18] Massimiliano Chiodo, P. Giusto, A. Jurecska, A. Hsieh, A. S. Vincentelli, *Hardware-software codesign of embedded systems*. IEEE Micro, Volume 14, Issue 4, 1994, pp.26-36.

[19] Hyunok Oh, S. Ha, *Hardware-software cosynthesis of multi-mode multi-task embedded systems with real-time constraints*. Proceedings of the 10th international ACM symposium on Hardware/software codesign, 2002, pp.133-138.

[20] Yanbing Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, J. Stockwood, *Hardware-software co-design of embedded reconfigurable architectures*. Proceedings of the 37th ACM conference on Design automation, 2000, pp.507-512.

[21] Mateusz Majer, J. Teich, A. Ahmadinia, C. Bobda, *The Erlangen slot machine: A dynamically reconfigurable FPGA-based computer*. The Journal of VLSI Signal Processing, Volume 47, Number 1, April 2007, pp.15-31.

[22] Lesley Shannon, P. Chow, *Using reconfigurability to achieve real-time profiling for hardware/software codesign*. Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays, 2004, pp.190-199.

[23] Benot Miramond, J.M. Delosme, *Design space exploration for dynamically reconfigurable architectures*. Proceedings of the Design, Automation and Test in Europe, 2005, pp.366-371.

[24] Jiang Xu, W. Wolf, J. Henkel, S. Chakradhar, *A design methodology for application-specific networks-on-chip*. ACM Transactions on Embedded Computing Systems, Volume 5, Issue 2, May 2006, pp.263-280.

[25] C.Y. Jung, M.H. Sunwoo, S.K. Oh, *Design of reconfigurable coprocessor for communication systems*. IEEE Workshop on Signal Processing Systems, October 2004, pp.142-147.

[26] John Glossner, D. Iancu, J. Lu, E. Hokenek, M. Moudgill, *A software defined communications baseband design*. IEEE Communications Magazine, Volume 41, Number 1, January 2003, pp.120-128.

[27] Michael Schulte, J. Glossner, S. Jinturkar, M. Moudgill, S. Mamidi, S. Vassiliadis, *A low-power multithreaded processor for software defined radio*. The Journal of VLSI Signal Processing, Volume 43, Numbers 2-3, June 2006, pp.143-159.

[28] S. Rajagopal, S. Rixner, J. R. Cavallaro, *A programmable baseband processor design for software defined radios*. The 45th Midwest Symposium on Circuits and Systems, Volume 3, August 2002, pp.413-416.

[29] Yuan Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, K. Flautner, *SODA: A low-power architecture for software radio*. IEEE Proceedings of the 33rd annual international symposium on Computer Architecture, 2006, pp.89-101.

[30] Hans-Martin Bluethgen, C. Grassmann, W. Raab, U. Ramacher, J. Hausner, *A programmable baseband platform for software-defined radio*. Proceedings of the SDR 2004 Techincal Conference and Product Exposition, SDR Forum, 2004.

[31] Joseph Mitola III, G. Q. Maguire Jr., *Cognitive radio: making software radios more personal*. IEEE Personal Communications, Volume 6, Issue 4, August 1999, pp.13-18.

[32] Hyunseok Lee, Y. Lin, Y. Harel, M. Woh, S. Mahlke, T. Mudge, K. Flautner, *Software defined radio - A high performance embedded challenge*. High Performance Embedded Arhcitectures and Compilers: Lecture Notes in Computer Science, Volume 3793 2005, pp.6-26.

[33] J. P. Delahaye, H. Gogniat, C. roland, P. Bomel, *Software radio and dynamic reconfiguration on a DSP/FPGA platform*. Proceedings of the 3rd Karlsruhe Workshop on Software Radios, March 2004, pp.143-151.

[34] M. Cummings, S. Haruyama, *FPGA in the software radio*. IEEE Communications Magazine, Volume 37, Issue 2, February 1999, pp.108-112.

[35] J. Mitola III, *Software radio architecture: a mathematical perspective*. IEEE Journal on Selected Areas in Communications, Volume 17, Issue 4, April 1999, pp.514-538.

[36] T. Hentschel, M. Henker, G. Fettweis, *The digital front-end of software radio terminals*. IEEE Personal Communications, Volume 6, Issue 4, August 1999, pp.40-46.

[37] F. J. Harris, M. Rice, *Multirate digital filters for symbol timing synchronization in software defined radios*. IEEE Journal on Selected Areas in Communications, Volume 19, Issue 12, December 2001, pp.2346-2357.

[38] Texas Instruments, *TMS320C6000 DSP external memory interface (EMIF) : Reference guide*. TI SPRU266E, April 2008.

[39] GMR-1 01.202, *GMR-1: Part1: General specifications; Sub-part 3: General system description*. European Telecommunications Standards Institute, 2005.

[40] GMR-1 05.001, *GMR-1: Part5: Radio interface physical layer specifications; Sub-part 1: General description*. European Telecommunications Standards Institute, 2005.

Curriculum Vitae

Jason M. Bales graduated from West Springfield High School, Springfield, Virginia, in 1997. He received his Bachelor of Science in Electrical Engineering from George Mason University in 2004. After graduating, he was employed with the US Army Night Vision and Electronic Sensors Directorate at Fort Belvoir, Virginia for two years as a Research engineer working on small, wireless disposable embedded sensor systems. During those two years he also lectured an undergraduate course on microcontroller programming and applications at George Mason University. In 2006 he accepted a position with ArgonST, Inc. in Newington, Virginia where he works as a Digital Signal Processing engineer. His work includes design and testing of satellite and terrestrial based software radios.