NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

DISSERTATION

A SYSTEMATIC SOFTWARE, FIRMWARE, AND HARDWARE CODESIGN METHODOLOGY FOR DIGITAL SIGNAL PROCESSING

by

Daniel Y. Chang

March 2014

Dissertation Supervisor: Neil C. Rowe

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2014	**3. REPORT TYPE AND DATES COVERED** Dissertation

4. TITLE AND SUBTITLE: A SYSTEMATIC SOFTWARE, FIRMWARE, AND HARDWARE CODESIGN METHODOLOGY FOR DIGITAL SIGNAL PROCESSING	**5. FUNDING NUMBERS**
6. AUTHOR(S) Daniel Y. Chang	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) ECSEL/JEWEL, NAWCWD 575 I Avenue, Suite 1 Point Mugu, CA 93042-5049	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB protocol number N/A

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Creating an embedded system that meets its functional, performance, cost, and schedule goals is a software-and-hardware codesign problem, since the design of the software and hardware components influence each other. The traditional design methodology is sequential, with hardware designed first and then software. The lack of a unified and unbiased approach can lead to suboptimal design and incompatibilities across the software and hardware boundary.

To solve these problems, we propose a new software/firmware/hardware codesign methodology to systematically build correct designs efficiently. This codesign methodology includes requirements development, architecture forming, software/ firmware/hardware partitioning, design-pattern mapping, new-design pattern synthesis, integration, and testing.

We tested our methods on three application areas. One was a digitizer-filter architecture for ultra-high frequency signals for which we synthesized design patterns in firmware to meet high-frequency requirements. Another was a digitizer-filter architecture for low-frequency signals. A third was a hidden Markov model using dynamic programming. We implemented and tested the first application on a Tektronix/Synopsys embedded system and the second on a Pentek embedded system based on the requirements provided by the stakeholders

14. SUBJECT TERMS A*, AND/OR graph, AO*, codesign, concurrent design, data alignment, digital signal processing, design pattern, embedded systems, firmware/software/hardware codesign, FPGA, OR tree, hidden Markov model, polyphase DFT filter banks, post-deserialization bits remapping, pre-serialization bits remapping, switch-and-filter architecture, reconfigurable computing	**15. NUMBER OF PAGES** 211
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

A SYSTEMATIC SOFTWARE, FIRMWARE, AND HARDWARE CODESIGN METHODOLOGY FOR DIGITAL SIGNAL PROCESSING

Daniel Y. Chang
Civilian, Naval Air Warfare Center, Weapons Division
B.S., Fu-Jen Catholic University, 1979
M.S.E.E, California State University, Northridge, 1992

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN SOFTWARE ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 2014

Author: ______________________________
Daniel Y. Chang

Approved by:

____
Neil C. Rowe
Professor of Computer Science
Dissertation Supervisor, Dissertation Committee Chair

______________________	______________________
Mikhail Auguston	Man-Tak Shing
Associate Professor	Associate Professor
of Computer Science	of Computer Science
______________________	______________________
Roberto Cristi	Melissa Midzor
Professor of Electrical Engineering	JEWEL, NAWCWD

Approved by: ______________________________
Peter J. Denning, Chair, Department of Computer Science

Approved by: ______________________________
Douglas Moses, Vice Provost for Academic Affairs

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Creating an embedded system that meets its functional, performance, cost, and schedule goals is a software-and-hardware codesign problem, since the design of the software and hardware components influence each other. The traditional design methodology is sequential, with hardware designed first and then software. The lack of a unified and unbiased approach can lead to suboptimal design and incompatibilities across the software and hardware boundary.

To solve these problems, we propose a new software/firmware/hardware codesign methodology to systematically build correct designs efficiently. This codesign methodology includes requirements development, architecture forming, software/ firmware/hardware partitioning, design-pattern mapping, new-design pattern synthesis, integration, and testing.

We tested our methods on three application areas. One was a digitizer-filter architecture for ultra-high frequency signals for which we synthesized design patterns in firmware to meet high-frequency requirements. Another was a digitizer-filter architecture for low-frequency signals. A third was a hidden Markov model using dynamic programming. We implemented and tested the first application on a Tektronix/Synopsys embedded system and the second on a Pentek embedded system based on the requirements provided by the stakeholders.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

A*	A star
ADC	analog-to-digital converter
ADTS	air data test set
AIRL	Airborne Interceptor Research Laboratory
AO*	AO star
ASIC	application-specific integrated circuit
CAD	computer aided design
CFSM	Codesign Finite-State Machine
COTS	Commercial off-the-shelf
CPLD	complex programmable logic device
DAC	digital to analog converter
DCM	data converter module
DFT	discrete Fourier transform
DOD	Department of Defense
DSL	domain specific language
DSM	domain specific modeling
DSP	digital signal processing
EMI	electromagnetic interference
ENOB	effective number of bits
EP	extreme programming
EW	electronic warfare
FCC	Federal Communications Commission
FFT	fast Fourier transform
FIR	finite impulse response
FPGA	field programmable gate array
FSM	finite state machine
GPU	graphical processing unit
GSPS	giga samples per second

GUI	graphical user interface
HDL	hardware description language
HMM	hidden Markov model
HWIL	hardware in the loop
IBW	instantaneous bandwidth
IC	integrated circuit
IF	intermediate frequency
IQ	in-phase quadrature
ISE	Integrated Software Environment
JEWEL	Joint Electronic Warfare Effects Laboratory
LOC	logic of constraints
LTL	linear temporal logic
MBD	model-based design
MDA	model-driven architecture
MDD	model-driven development
MDRE	model-driven requirements engineering
MMM	METROPOLIS meta-model
MoC	model of computation
MSPS	mega samples per second
OBP	object based programming
OJT	on-the-job-training
OMG	Object Management Group
OOP	object oriented programming
PBD	platform-based design
PIM	platform-independent model
PRF	pulse repetition frequency
PSM	platform-specific model
PSoC	programmable system on a chip
RA	requirements analysis
RAM	random access memory

RE	requirements engineering
RTL	register transfer language
RUP	rational unified process
SDRF	software defined radiofrequency
SIS	a system for sequential circuit synthesis
SLD	system level design
SoC	system on a chip
SoPC	system on a programmable chip
SRAM	serial random access memory
SWAP	size weight and power
UAV	unmanned aerial vehicle
UHF	ultra-high frequency
UML	universal modeling language
Verilog	verify logic
VHDL	VHSIC hardware description language
VHSIC	very high speed integrated circuit
VIS	verification interacting with synthesis

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Creating an embedded system which meets its functional, performance, cost, and schedule goals is a software-and-hardware codesign problem, since the design of the software and hardware components influence each other. The traditional design methodology is sequential, with hardware designed first and then software. The lack of a unified and unbiased approach can lead to suboptimal design and incompatibilities across the software and hardware boundary.

To solve these problems, we develop a new codesign methodology to partition software/firmware/hardware, and then map functional components to design patterns if existing. This methodology includes first building a tree with conjunctions and disjunctions of possible mappings from functional components to the options of software, firmware, and hardware following requirements and constraints; second, rating the cost of each mapping; third, searching the tree to find a minimum weighted sum of the costs; and fourth, identifying existing design patterns once design is selected, and otherwise synthesizing new design patterns.

We tested our methods on three application areas. The first was a digitizer-filter architecture for ultra-high frequency signals. The major challenge was how to move ultra-fast data from a faster sensor (including a digitizer) to a slower processor and then perform useful tasks. We implemented and tested this application on a Tektronix/Synopsys demo embedded system based on the test specifications established by the vendor and Joint Electronic Warfare Effects Laboratory (JEWEL) at Point Mugu, California.

The second application was a digitizer-filter architecture for low-frequency requirements. The challenge was how to partition software/firmware/hardware and then map design to the existing vendor products to save development time and cost. We implemented and tested this application on a Pentek embedded system based on the test specifications established by the vendor and Airborne Interceptor Research Laboratory (AIRL) at Point Mugu, California. By way of contrast, for a period of more than 12

months from 2011 to 2012, with 10 engineers, we spent \$3.16M on an airborne interceptor project (including software, firmware and hardware designs) but failed to produce any software deliverable. In 2013, for a period of five months, starting from ground zero, with two engineers, we only spent \$90K on the same project with the help of our new software/firmware/hardware codesign methodology; we were able to map 86 percent of our project unto vendor’s existing products and delivered Doppler range gating software successfully.

The third application was a hidden Markov model using dynamic programming. The challenge was how to partition software/firmware/hardware for better processing speed performance. We discussed the advantages and disadvantages of mapping hidden Markov models unto software and firmware in terms of cost, speed and design complexity.

ACKNOWLEDGMENTS

First I would like to thank my wife, Jennifer, for her unconditional love and encouragement for my PhD studies. Throughout this journey, she has never complained about my long hours of studies each day, including weekends and holidays. To show her support, she even accompanied me for my written examination and final oral defense at Naval Postgraduate School, Monterey, California.

This research is based on Dr. Melissa Midzor's vision in digitizing radiofrequency signals in the gigahertz range used for electronic warfare signal simulation. Without her vision and support in funding, equipment, and personnel, this research cannot be made possible.

Through weekly phone conversations, Professor Neil C. Rowe tirelessly provided strategic advice ensuring our work was on track. Even though, he is methodical in guiding this project, he is extremely flexible and open-minded, which allowed me to have much freedom in accomplishing the task. In addition, Professor Rowe suggests using an artificial intelligence A* search with embedded AND nodes methodology to partition software/firmware/hardware, and then match the optimal leaf-nodes to some design patterns for embedded system design. Software and hardware partitioning is a very difficult challenge in the field of hardware-and-software codesign.

Special thanks to Professor Roberto Cristi for his enthusiasm and knowledge in digital signal processing. He gave directions in dividing ultra-wide bandwidth signals into multiple parallel subbands using polyphase discrete Fourier transform filter banks. Equally important, he has been enormously generous in sharing his lecture videos and class notes with me on various topics.

Professor Shing thoroughly reviewed my dissertation and pointed out the most important weakness in software testing. Without his meticulousness and diligence, this dissertation could not have been completed. In addition, I audited his Requirements Engineering course and much of the material in Chapter II reflects his knowledge.

Personally, I have taken Formal Methods and audited Software Testing from Professor Auguston; I was very impressed with his knowledge in fundamental software engineering and up-to-date tools. His influence can be seen clearly in Chapter V (Case Study One). Last but not the least, while Professor Auguston was experiencing a very difficult health condition, he was still able to promptly review my dissertation and give me valuable feedback.

I. INTRODUCTION AND PROBLEM ADDRESSED

A. ADDRESSED PROBLEM

Creating an embedded system which meets its functional, performance, cost, and schedule goals is a software-and-hardware codesign problem, since the design of the software and hardware components influence each other [1]. Traditionally, when designing an embedded system, hardware is designed first by a group of hardware engineers, and then software is designed by a group of software engineers. Once a design is completed, both software and hardware engineers strive to make every effort to implement changes in software to avoid expensive hardware redesign. The problems with this approach are:

- A presumptive definition of software-and-hardware partitions can cause suboptimal designs.
- Lack of a unified software-and-hardware design methodology can cause incompatibilities across the software and hardware boundary.

To solve these problems, the codesign group at U.C. Berkeley in 1997 developed a framework, POLIS, with a unified software-and-hardware representation for unbiased specification, automatic synthesis, and validation of the embedded systems. The most difficult challenge in POLIS according to the group is software-and-hardware partitioning because the partitioning decisions are heavily based on designer's expertise and are very difficult to automate [2].

For embedded systems, software can be divided into three categories: computer-based software, central-processing-unit (CPU) firmware, and reconfigurable computing firmware. Software and firmware both contain programming instructions and necessary documentations, except that software runs on a computer and firmware runs on a hardware device. As a result, it is helpful to replace the term software/hardware codesign with software/firmware/hardware codesign.

Our goal for this dissertation is to provide a new software/firmware/hardware codesign methodology for seamless integration and design of embedded systems. There are several problems to resolve, in particular functional decomposition, what should be classified as software or firmware or hardware, design-pattern mapping, and new design-pattern synthesis.

B. MOTIVATION

A motivating problem is how to push the upper limit of the capability in moving very fast digitized data from a sensor and digitizer to a slower processor, and then usefully process it in real-time. The conventional data rate for a sensor is in the range of megasamples-per-second; the data rate for our research is in the range of gigasamples-per-second and above. This is too fast for an all-software design.

The solution to this problem is in software-and-hardware codesign because the final system must be flexible enough to accommodate different data rates and perform various useful tasks, and must not only function properly but also meet critical timing constraints due to its ultra-high data speed [1,3]. There are many details we need to keep straight; a software/firmware/hardware codesign methodology will help us build designs correctly and efficiently.

There are many similar problems in design of systems for use in electronic warfare, in which proper design decisions are critical because of the signal frequencies involved and the processing time required. Electronic warfare tries to dominate the electromagnetic spectrum, or to protect our use of the electromagnetic spectrum and to exploit the enemy’s spectrum. This involves minimizing mutual interference among friendly systems, minimizing detection by enemy sensors, and minimizing enemy interference with the ability to execute a military deception plan. Techniques often used to prevent or reduce an enemy’s effective use of the electromagnetic spectrum are jamming and electromagnetic deception [4]. An advantage can be gained in the domain of the electromagnetic spectrum by being able to handle higher frequencies than an adversary can handle.

A wide variety of equipment has been designed and used in electronic warfare, such as Northrop Grumman EA-6B Prowlers, Boeing EA-18G Growlers, unmanned aerial vehicle systems, and ground jamming vehicles. To ensure the readiness of this equipment, tests and evaluations are required. The environment for tests and evaluations is simulated in software (that is, the aircraft, vehicles, terrain and weather), but the hardware is real.

C. CLAIM

Our claim is that rather than the trial-and-error approach being currently practiced for embedded system design, a new software/firmware/hardware codesign methodology based in software engineering has the potential to systematically build correct designs efficiently to satisfy the requirements provided by the stakeholders.

D. TRADITIONAL METHODOLOGY FOR DSP DESIGN

For many real-time applications, a specialized field-programmable gate array (FPGA) embedded system, instead of a general-purpose computer, should be used [5]. The reason is that an FPGA can process hundreds of times more operations per clock cycle than a processor. The speed of a state-of-the-art multicore processor is in gigahertz and the speed of a state-of-the-art FPGA is in hundreds of megahertz. Also, a typical high-end FPGA has thousands of times more parallel channels than a multicore processor.

Traditional FPGA-based embedded software is written manually from text-based specification and requirements. This approach is time-consuming and error-prone, and there is little tracking to ensure that changes are correctly implemented [6]. A more systematic approach provided by software engineering could reduce these problems.

E. OUR SOFTWARE ENGINEERING METHODOLOGY

To develop specifications, we start with gathering high-level requirements in the form of novel ideas and questions. With proper domain knowledge, we can derive subrequirements from them. In requirements analysis, we use feature models and decision trees to explore design concepts and possible implementation technologies for the feasibility check. These design concepts and implementation technologies are in the form of models, and they can also be used for fine-tuning the requirements in the next requirements development iteration. During this phase, we treat software, firmware, and hardware together, since a software component and a (reconfigurable) hardware component can often both achieve a design, although software tends to be more flexible and hardware tends to be faster [7]. The final products at the end of this stage should include requirements and design models as well as dataflow and control-flow architectures.

We can generate models by using specialized FPGA embedded software design tools. Non-specialized tools such as documentation, reports, tables, diagrams, and algorithms can also assist model building without the benefits of automatic code generation [8].

Next, we must decide what should be classified as software, what should be classified as firmware, and what should be classified as hardware. Software-and-hardware partitioning involves a diversity of applications, design styles and implementation technologies; ultimately it depends on human expert knowledge [7]. In this dissertation we propose using a tree of options to find possible mappings from functional components to the set of modalities {software, firmware, hardware}. When an optimal node (solution) is chosen, we can expand any component within a node into subcomponents, and then use the same methods to assign the subcomponents. When we have found the best assignment for the subcomponent search, we embed it in the original tree.

During the construction phase, we apply software and FPGA programming methodology, and perhaps additional hardware design, to implement the design. The process flow includes designing (using a hardware description language or a high-level graphical modeling tool), functional and timing simulations, doing synthesis, implementation and programming.

F. TOPICS COVERED FROM CHAPTERS II TO VIII

In Chapter II, we survey some important software-engineering methodologies in the fields of requirements engineering, embedded-system design, and concurrent-system design. We also discuss FPGA programming languages and FPGA design methodology. In Chapter III, we use an ***OR*** tree with embedded ***ANDs*** to partition software/firmware/hardware, and then use the same methodology to map the optimal leaf-node to a design pattern if existing; otherwise, we synthesize a new design pattern. In Chapter IV, we present five example design patterns for reconfigurable computing based embedded systems: data alignment, post-de-serialization bits remapping, pre-serialization bits remapping, polyphase DFT filter banks and switch-and-filter. These patterns were used in implementing our case studies. In Chapter V, we present a case study using a Tektronix Digitizer/FPGA/DAC demo unit to digitize and process radiofrequency signals up to 6 gigahertz and then discuss the test results for this case study. In Chapter VI, we present a case study fora conventional Doppler radar receiver. In Chapter VII, we present a case study involving a Hidden Markov model (HMM). In Chapter VIII, we conclude the dissertation by stating our major contributions and suggest directions for future research.

THIS PAGE INTENTIONALLY LEFT BLANK

II. PREVIOUS WORK

In this chapter we will survey current software engineering methodologies related to requirements engineering (RE), embedded-system design, hardware-and-software codesign (or concurrent system design), and field-programmable gate array (FPGA) design to lay the foundation for our research.

A. REQUIREMENTS ENGINEERING

1. Requirements Development

Requirements development includes six key activities—elicitation, analysis, validation, negotiation, documentation and management [9]. Elicitation is to discover system requirements through consultation with stakeholders, and to establish a scope and boundary for the project. Analysis is to analyze requirements in detail and to identify possible conflicts and overlaps. Validation is to review or validate requirements for clarity, consistency, and completeness with stakeholders. Negotiation with stakeholders establishes which requirements are to be considered. Documentation is to write down agreed requirements at a certain level of detail for review, evaluation, and approval. Requirements management is an ongoing activity that starts from the moment the first requirement is elicited and ends only when the system is finally decommissioned [10]. Requirements management includes software-baseline definition, change control, and approval and status tracking. The baseline can be defined as a set of features agreed to be delivered to customers in a specific software version [11].

Some requirements engineering researchers use the term "specification" for "documentation" [12], combine analysis and negotiation into analysis activity, and treat management as a different topic. Based on this, requirements development includes four key activities—elicitation, analysis, specification and validation [10].

The scheduling for requirements engineering should not be less than one third of the entire project time, since much time is required to include research or exploration of new techniques [13].

2. Rational Unified Process

To manage requirements effectively, we must have a well-defined software lifecycle development process. The waterfall model, introduced by Winston Royce in 1970 [14], requires complete and fully elaborated requirements before design, coding, testing, operation and maintenance. This is not practical since requirements change throughout the entire software lifecycle. The spiral model, introduced by Barry Boehm in 1988 [15], requires multiple risk analyses, validations and prototypes before a rigorous waterfall methodology is followed, and also has proven to be expensive and time-consuming.

In 1995, Philippe Kruchten [16] introduced the *iterative* approach. This divides a software project into multiple time-boxed iterations. An iteration is a sequence of activities, such as requirements, design, implementation, test, and integration, resulting in an executable of some type. Each iteration is based on prior iterations. Some benefits of the iterative process are early risk mitigation, early feedback, and analysis-paralysis avoidance.

In 2003, Rational Software Corporation (a division of IBM) proposed the "rational unified process" for software lifecycle development. It is a sequence of inception, elaboration, construction, and transition. Each phase consists of one or more executable iterations of the software at that stage of development. Inception is to define the scope of project, and its milestone is the identification of actors and use cases. Elaboration is to plan project and specify features, and its milestone is the establishment of baseline architecture. Construction is to build the product, and its milestone is the building of initial operational capability. Transition is to deliver products to customers, and its milestone is the release of product. Each phase has multiple iterations. The number of iterations depends on the project size and agreement among stakeholders. The relationship between lifecycle phases, milestones and iterations are shown in Figure 1 [17].

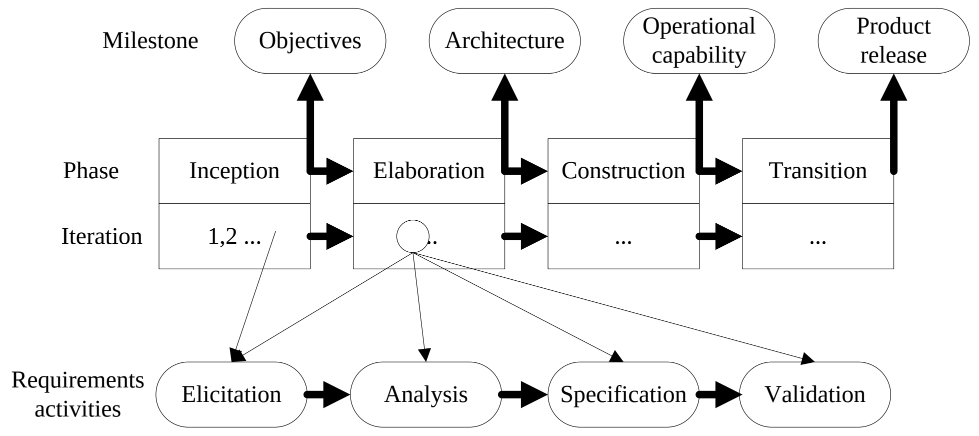


Figure 1 Rational unified process lifecycle phases, milestones and iterations

3. Design-Based Requirements

The requirements and design activities must be iterative. Gestalt round-trip design, introduced by Grady Booch [18] in 1994, emphasizes the human characteristic of learning by completing [19]. In other words, the requirements at one iteration cause us to select certain design options, and the selected design options may in turn initiate new requirements. This is due to the fact that requirements are always changing and cannot be correctly defined until some design work is developed.

4. Test-Based Requirements

In 2008, Martin and Melnik proposed a Möebius strip approach for requirements-development [20]. A Möebius strip means that writing requirements and testing are closely interrelated. Writing requirements in the form of acceptance tests can reduce the number of pointless features and code and handle changes more efficiently. Every requirement must be testable with this approach.

5. Agile Software Development

In 2001, the Agile Alliance stated that software development should be focusing on (1) individuals and interactions over processes and tools, (2) working software over comprehensive documentation, (3) customer collaboration over contract negotiation, and (4) responding to change over following a plan [21]. With this approach, emphasis is placed on the end result of working software rather than comprehensive documentation. In addition, the client is taken on-board as a member of the development team, so that missing requirements are discovered in the early software development stage.

6. Model-Driven Requirements Engineering

A model is a representation of a system that allows for investigation of the properties of the system and, in some cases, prediction of future outcomes. Software models come in many forms, such as use cases, diagrams, and statecharts [22].

Requirements can also be modeled, and the benefits of requirements modeling are: (1) allowing us to understand the product requirements precisely, (2) showing generalizations, (3) simplifying complex relationships between requirements, and (4) describing the context and background in which the product will be used.

There are different types of requirements models, such as business models, feature/goal models, analysis (use case) models, design models, implementation models and test models. A business model describes why a product is needed. A feature model describes the features of a product. A requirements analysis model explains the features in sufficient detail to define product specifications. A design model illustrates the architecture for the product. An implementation model describes the construction of the product. A test model describes how the product would be tested [23].

7. Model-Driven Development

Model-driven development (MDD) is a software engineering approach which uses models of high-level abstraction to create and evolve software. The goals of model-driven development are to simplify and formalize the various activities in software lifecycle [24]. According to Object Management Group (OMG), model-driven design can be realized by using model-driven architecture (MDA). Model-driven architecture specification consists of a definitive platform-independent model (PIM), plus one or more platform-specific models (PSMs) and sets of interface definitions, each describing how the base model is implemented on a different middleware platform.

8. Domain-Specific Modeling and Language

Domain-specific modeling (DSM) is a software engineering methodology for designing and developing software systems by using domain-specific modeling languages (DSL) to represent various features of the system [25]. Domain-specific languages support higher-level abstraction than general-purpose modeling languages, so they require less effort and fewer low-level details to specify a given system. Most domain-specific models and languages are created for a particular domain by a particular vendor with domain expertise; as a result, automatic quality code generation is made possible. As Booch pointed out in 2004, to achieve the full value of model-driven architecture, modeling concepts must map directly to domain concepts rather than computer technology concepts [26].

One example is hidden Markov model "toolbox" for MATLAB from MathWorks that supports inference and learning for hidden Markov models. This toolbox is designed for a specific domain (statistical inference) and the language is domain-specific with a higher-level of abstraction (e.g., TRANS representing transmission matrix and EMIS representing emission matrix); this model cannot be easily described by using a general-purpose Universal Modeling Language (UML).

The best practices put forth by the rational unified process (incremental and iterative), extreme programming (test-driven), agile development (client-oriented and design-based), and model-driven and domain-specific methodologies are used throughout the software-development lifecycle of our three case studies.

B. SOFTWARE ENGINEERING OF EMBEDDED SYSTEMS

An embedded system is hardware and software which forms a component of some larger system and which is expected to function without human intervention. Firmware is a software program or a set of instructions programmed on a reconfigurable hardware device. Software is associated with a computer system, firmware is associated with an embedded system, and both of them contain programming instructions and necessary documentation.

Microprocessors and FPGAs are two major kinds of programmable integrated circuits (ICs) in an embedded system. In this research, our embedded system contains FPGAs instead of microprocessors, so we will only survey the issues with FPGAs. Our rationale is explained in Section D, Chapter I.

Much embedded software is based on the traditional waterfall model, using "emphasis on fully elaborated documents as completion criteria for early requirements and design phases" [27]. Since code is written manually from text-based specifications and requirements, and fully elaborated documents are not possible for most projects, this approach is time-consuming and error-prone, and there is little tracking to ensure that changes are correctly implemented [6].

Model-based design allows concise representation of behavior at a high level of abstraction. It is a better choice for embedded software development since the entire system can be visualized graphically, which leads to easy comprehension of the system; its models can be validated and verified; and it is easier to refine models and track model changes than with text-based documents. Model-based design also creates a structure allowing for software reuse; options and performance can be evaluated and the outcome can be predicted; and code can be automatically generated from the fully tested specification for software development and rapid prototyping [28].

Some major specialized model-based embedded software design tools are Mathworks MATLAB/SIMULINK®, Synopsys Synphony Model Compiler®, Annapolis CoreFire®, and National Instruments LabView®. More general tools (such as documentations, reports, tables, diagrams, and algorithms) can also be used to assist model building without the benefits of automatic code generation [8].

C. CONCURRENT SOFTWARE ENGINEERING OF EMBEDDED SYSTEMS

1. A Top-down Concurrent Design Process for an Embedded System

Creating an embedded system which meets its functional, performance, cost, and schedule goals is a hardware-and-software codesign problem, since the design of the hardware and software components influence each other [1]. We can define hardware-

and-software codesign as “meeting system level objectives by exploiting the synergism of hardware and software through their concurrent design” [29].

An FPGA or any reconfigurable computing can be reconfigured to perform different functions without changing the underlying hardware. From a user perspective, reconfigurable computing can function equivalently to software running on a processor [29].

To address hardware-and-software codesign problems, Wolf in 1994 [1] suggested using a top-down design process for embedded systems design as shown in Figure 2. A brief explanation for each step is provided below:

- Specification: Process models are used to represent both the hardware and software elements without biased implementation. A requirements model is produced and it includes a dataflow diagram, a control -flow diagram, response-time specifications, and a requirements dictionary; this model can be tested and validated. A dataflow diagram (DFD) is a graphical representation of the flow of data through an information system. A control flow diagram (CFD) is a diagram to describe the control flow of a process or program.

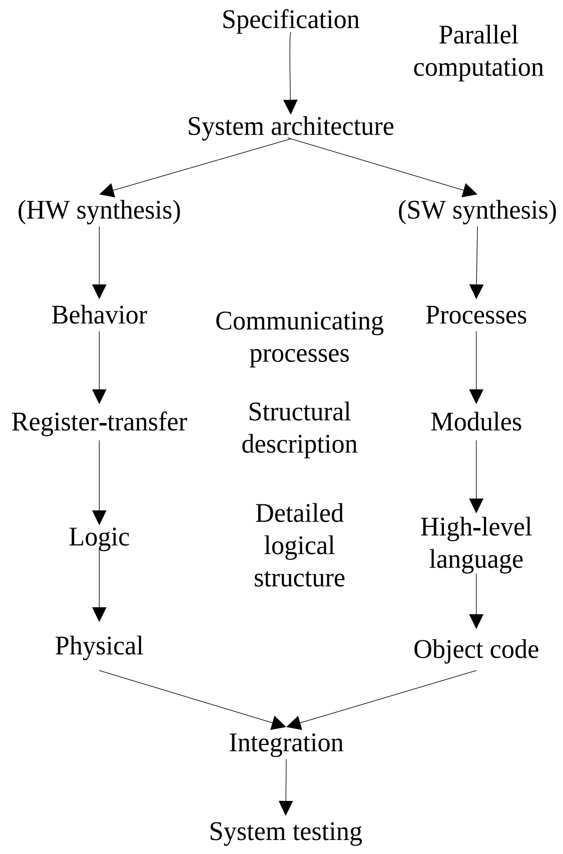


Figure 2 A top-down concurrent design process for an embedded system

- System Architecture: The architecture model includes an architecture flow diagram which allocates functional elements of the requirements model to physical units in the architecture, and an architecture interconnect diagram (a block diagram). The first component in a system architecture to be considered should be the hardware engine (a processor), since it provides the raw computing power for the system instruction execution and peripheral operations.
- Synthesis: Once hardware and software components are partitioned and selected, software can be compiled and tested for a certain processor; hardware can be synthesized, simulated, and implemented onto a particular hardware device.
- Integration: We can integrate implementations of hardware and software components together after they are synthesized and tested individually, with their interfaces.
- System testing: After integration, we can perform system testing to verify and validate the entire system.

2. POLIS

In 1997, the codesign group at U.C. Berkeley, California [30] developed POLIS, a software tool for hardware-and-software codesign. A POLIS **s**ystem is represented by a codesign finite-state machine (CFSM), a model unbiased towards a hardware or software implementation. Each element of a network of codesign finite-state machines describes a component of the system to be modeled. A codesign finite-state machine is asynchronous since hardware and software exhibit different delay characteristics. It is synthesizable and verifiable, because many existing theories and tools for the finite-state machine model can be easily adapted for codesign finite-state machine [30].

POLIS is the realization of the hardware-and-software codesign methodology proposed by Wolf in section C.1. It takes advantage of the existing software tools developed by U.C. Berkeley, such as PTOLEMY, SIS and VIS. VIS (Verification Interacting with Synthesis) is a system for formal verification, synthesis, and simulation of finite-state systems. However, in terms of hardware-and-software partitioning, POLIS only gives designers feedback on their design choices; it is still based on trial-and-error approach that largely depends on designer’s expertise and knowledge.

The major steps are briefly described below.

- High-level language translation: Designers write their specifications in a high-level language that can be translated into codesign finite-state machines.
- Formal verification: POLIS translates the codesign finite-state machine into a formalism which can be verified by verification systems.
- System co-simulation: POLIS uses PTOLEMY as a simulation engine. PTOLEMY (developed by Lee at U.C. Berkeley) focuses on assembly of concurrent components.
- Design partitioning: Following Vahid [31], hardware-and-software partitioning is the problem of dividing an application’s computations into a part that executes as sequential instructions on a microprocessor (the software) and a part that runs as parallel circuits on some integrated circuit (the hardware). Making system-level design decisions such as hardware and software partitioning, target architecture selection, and scheduler selection is based heavily on design experience; therefore, it is very difficult to automate this process.

- Hardware synthesis: Codesign finite-state machine subnetworks chosen for hardware implementation by POLIS are implemented and optimized using logic synthesis techniques from SIS (a system for sequential circuit synthesis). SIS is an interactive program for the synthesis of both synchronous and asynchronous sequential circuits. The input can be given in state -table format or as logical equations (for synchronous circuits) or as a signal-transition graph (for asynchronous circuits). The output is a netlist of gates in the target technology. A netlist represents the connectivity of an electronic design. Figure 3 shows the POLIS design flow.

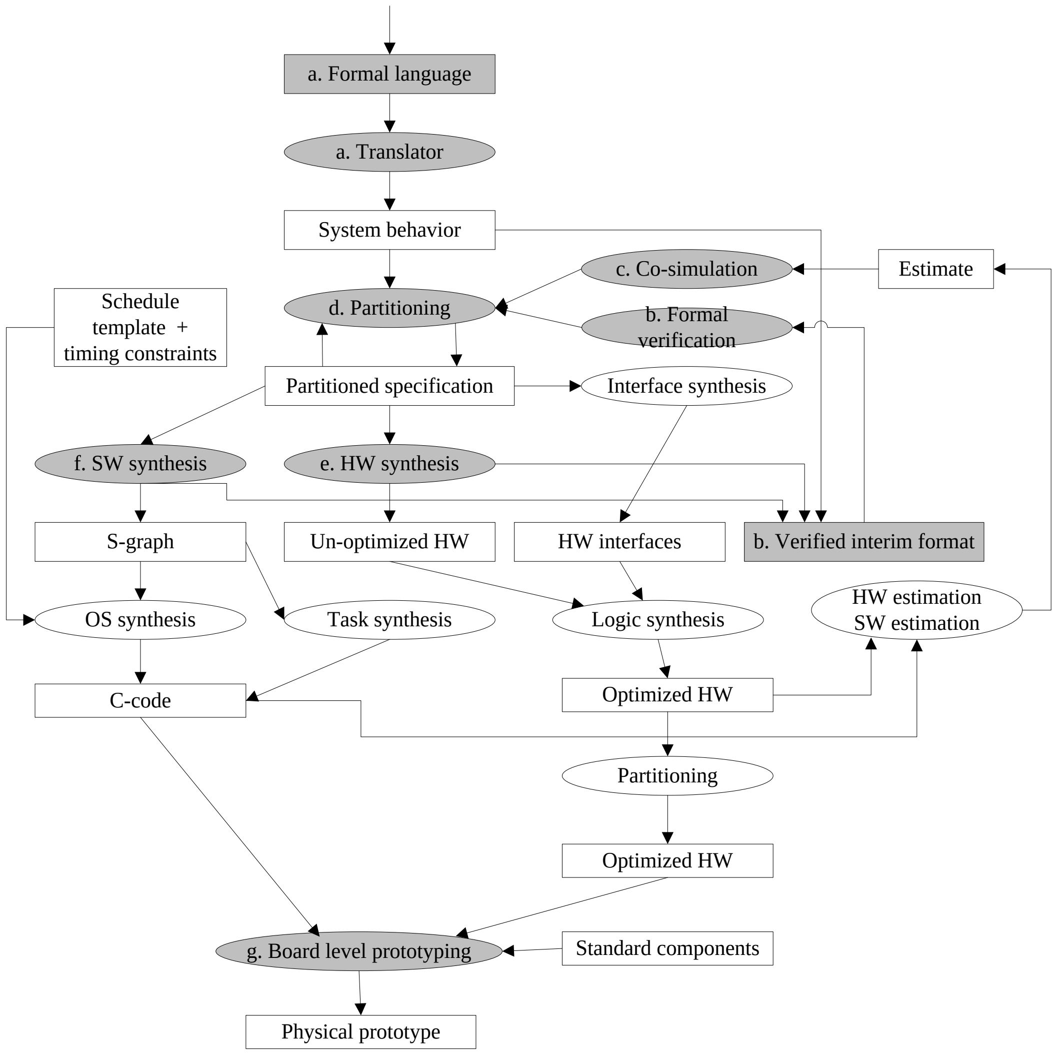


Figure 3 POLIS process

- Software synthesis: A codesign finite-state machine subnetwork chosen for software implementation is mapped into a software structure that includes a procedure for each codesign finite-state machine and a simple real-time operating system.
- Interfacing domains: Interfaces between different implementation domains (hardware and software) are automatically synthesized within POLIS. These interfaces come in the form of cooperating circuits and software procedures (I/O drivers) embedded in the synthesized implementation. Communication can be through I/O ports available on the microcontroller, or by general memory mapped I/O.

3. Orthogonalization of Concerns and Platform-Based Design

In 2004, the codesign group took a different approach for hardware-and-software codesign with the intention of increasing the reusability of software and hardware as well as applying the new methodology to heterogeneous systems (different domains) other than hardware and software. In this section, we briefly describe two concepts (orthogonalization of concerns and platform-based design) and the frameworks (METROPOLIS and METRO II) used for this approach.

Orthogonalization of concerns is the separation of the various aspects of design allowing more effective exploration of alternative solutions. Platform-based design is a unified design approach for hardware-and-software codesign. It summarizes the important parameters of the implementation in an abstract model, and carries out the design as a sequence of refinement steps that go from the initial specification to the final implementation using platforms at various level of abstraction [32,33,34].

A platform is defined as a library of components that can be assembled to generate a design at that level of abstraction [35]. The METROPOLIS design is a meet-in-the-middle process. A top-down process maps an instance of the functionality of the design into an instance of the platform; a bottom-up process builds a platform by choosing the components from a library (see Figure 4.)

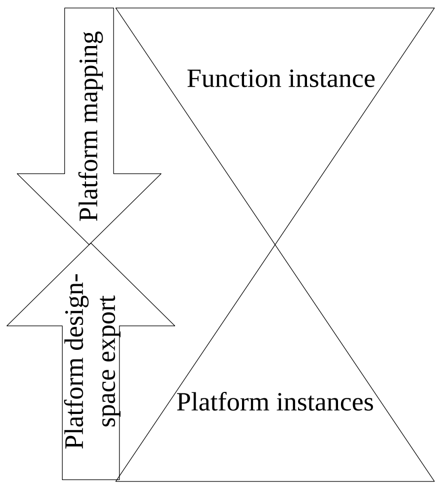


Figure 4 Platform-based design process

METROPOLIS, a platform-based design tool developed by Alessandro Pinto in 2004, provides a recursive paradigm where the action of mapping a function onto an architecture generates a new function described at a lower level of abstraction and more detailed than the original one. A design process should start with a denotational description of the function to implement plus a set of constraints that the implementation must satisfy. Constraints specified at this level of abstraction are propagated down to all subsequent levels until the implementation level is reached. While constraints are propagated in a top-down fashion, performances (such as speed and power) are abstracted in a bottom-up manner (see Figure 5.) Performance abstraction is the process of hiding details that are not relevant for the level of abstraction under consideration. In fact, each level of abstraction focuses on a particular design choice on which only few quantities have impact; this is essential for speeding up the design space exploration [36].

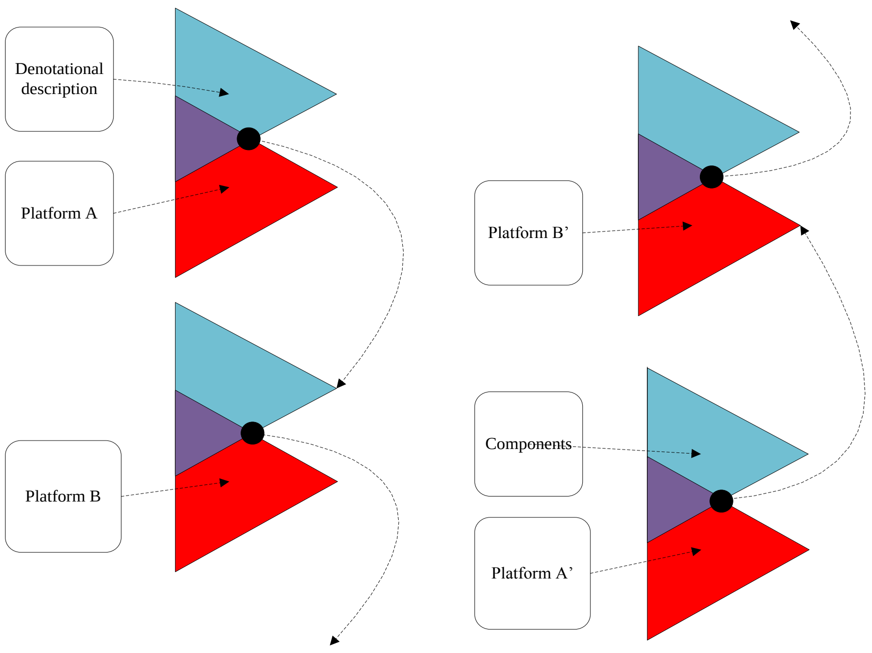


Figure 5   Platform-based design is iterative

The METRO II framework is an enhanced version of METROPOLIS. The improvements are:

1. The ability to import pre-designed intellectual properties (IPs).
2. The ability to separate cost from behavior when carrying out design.
3. The ability to explore the design space in a structured manner.

Platform-based design methodology provides an efficient way to map a functional design to an architecture in a library (top-down). At the same time, components in a library can be mapped to an architecture for implementation (bottom-up). The disadvantage of this method is that it depends on the trial-and-error approach and the availability of architectures and components in the libraries. Our methodology is similar but it provides a systematic way to partition hardware and software, and then map components to design patterns.

4. The Double Roof Model of Codesign, a System Level Design

Another view of codesign [37, 38, 39] identifies three challenges in synthesizing hardware and software:

1. Allocation: Select a set of system resources including processors, hardware intellectual property blocks, and their interconnection network to compose the system architecture in terms of resources. The design should be synthesizable.
2. Binding: Map functionality (e.g., tasks, processes, functions, or basic blocks) onto processing resources, variables and data structures onto memories, and communications to routes between corresponding resources.
3. Scheduling: Determine when functions are executed on the proper resources including function execution, memory accesses, and communication.

A feasible solution is the one satisfying the above triplets along with a certain number of additional nonfunctional constraints such as cost, performance, power, temperature, etc. To help reason about these three challenges of hardware and software design, a "double roof" model was proposed (see Figure 6.)

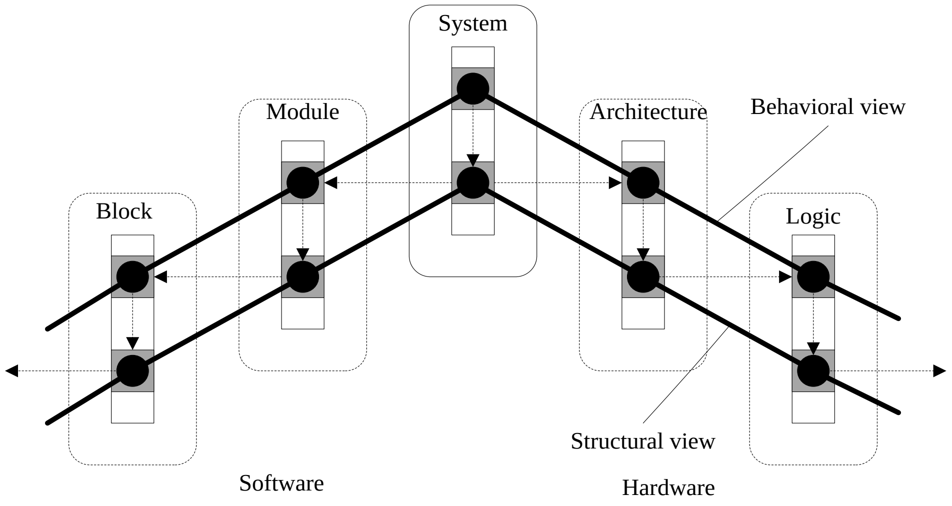


Figure 6 Double-roof model of codesign

The double-roof model defines the typical top-down design process for embedded hardware and software systems [37,38]. In Figure 6, the left-hand side of the roof shows a typical software design process, such as module (task) and block (instruction); the right-hand side shows a typical hardware design process, such as architecture and logic. At the highest system level of abstraction, one cannot distinguish between hardware and software.

The upper roof describes the functional or specification view of the system at the abstraction level, whereas the lower roof describes its structural implementation, including resources allocation, scheduling, binding, and coding. The vertical arrows represent synthesis steps, and the horizontal arrows indicate the step of passing information about the implementation at a certain level directly to the next lower level of abstraction as additional specification or constraints [39]. There is no fully automated design flow for all shown abstraction levels available today.

Comparing to the codesign process in section C.2, the double-roof model gives a better view of the relationship between hardware and software at different levels of abstraction. However, this model does not provide a systematic way to partition hardware and software.

5. Integrated Chip Codesign

To achieve efficient hardware and software integration, system-on-a-chip (SoC), programmable-system-on-a-chip (PSoC) and system-on-a-programmable-chip (PSoC) builder technologies are becoming available. An SoC is an integrated circuit that puts all components of a computer or other electronic system into a single chip; in essence, it is an advanced and powerful embedded system. A PSoC is an SoC with built-in programmable logic. SoPC Builder is a piece of software created by Altera that automates connecting reconfigurable computing (or soft-hardware) components to create a complete computer system that runs on any of its various FPGA chips.

Concurrent design is not a mature discipline due to the complex nature of the embedded systems. This leads to a lack of available computer aided design (CAD) tools and support [39]. The future of concurrent design methodology probably will be shaped by major SoC and PSoC manufacturers (such as Altera and Xilinx) and computer-aided design tools providers (such as Synopsys and MathWorks.)

We can apply software/firmware/hardware codesign methodology to system-on-a-chip and programmable-system-on-a-chip, since component integration is a step of implementation after our methodology.

D. FPGA DESIGN METHODOLOGY

The history of FPGA-based embedded system design goes back to 1985 when the first commercial FPGA was invented by Ross Freeman and Bernard Vonderschmitt, co-founders of Xilinx incorporation. In this section, we will briefly discuss FPGA's programming languages and design methodologies as well as coding examples and development tools.

1. FPGA and Central Processing Unit

The structure of an FPGA is not predefined as it can be programmed according to the user applications. The access to the internal resource of an FPGA is through I/O and clock pins; as a result, parallel processing can be easily achieved by custom programming. For non-timing-critical, complex, and dedicated tasks, a central processing unit is a better choice over an FPGA; for timing-critical, less complex, and heavily parallel tasks, an FPGA is a better choice over a central processing unit, even a multicore processor (explained in section D, Chapter I).

2. FPGA and Hardware Description Language

An FPGA is an integrated circuit designed to be configured by a customer or a designer after manufacturing in the field. A modern FPGA chip contains a combination of processors, embedded memory, programmable interconnects, dedicated digital signal processing (DSP) elements, and conventional lookup tables, multiple clock domains, high-speed serial I/Os connections, and a large number of pins [40]. To program the interconnects inside an FPGA, a programming tool is required.

Hardware description languages (HDL) are programming tools (or languages) for formal description and design of electronic circuits. They describe the circuit's operation (behavior), organization (architecture), and tests to verify its operation by means of simulation. With hardware description languages, the design can be verified before implementation by using simulation software tools; reuse is part of the language paradigm in dealing with complex designs; changes can be made easily; and hardware description languages can be used for documentations.

Hardware-description language is a specification language, not a traditional programming language. There are two primary real programming languages: Very High Speed Integrated Circuit (VHSIC) hardware description language (VHDL) and Verilog (Verify Logic). The relationship between HDL and VHDL/Verilog is analogous to the relationship between a class (e.g., human language) and two instances (e.g., English and French).

An FPGA can also be programmed by using model-based tools, such as Mathworks MATLAB/SIMULINK, Synopsys Synphony Model Compiler, Annapolis CoreFire and National Instruments LabView. These proprietary tools are not as widely used as hardware-description languages.

3. FPGA Programming

Figure 7 shows the process-flow in programming an FPGA. This process includes developing a design in hardware description language, synthesizing the design to a netlist, translating all designs into a single file, mapping the design to the resource in a targeted device, placing and routing the design on the device, creating a bit stream file and then programming an FPGA.

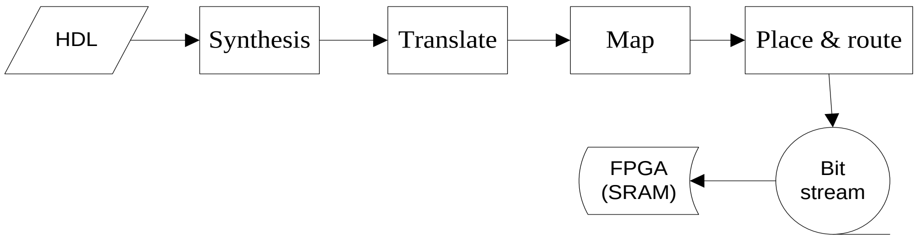


Figure 7 FPGA programming process

Each step of the process flow is further described below:

1. Synthesize: Send the hardware-description language code through a synthesis engine which translates the high-level hardware-description language code into a low-level netlist. A netlist represents the connectivity of an electronic design.
2. Translate: Merge all netlists and constraints (such as timing and area) into a single design file.

3. Map: Fit the design (generic logics) into the available resources on the target device.

4. Place and route: Place and route the design to the constraints on the target device.

5. Create bit stream: Turn the results from the place-and-route engine into a bitstream file. A bitstream file contains a series of bits for configuring (programming) the entire FPGA resource. This bitstream file is loaded into the FPGA's internal serial random-access memory. These memory cells are connected to various logic entities, multiplexers, lookup tables, Random Access Memory (RAM) blocks, and routing matrices, and constitute "configuration (programming)." Once the bitstream is loaded, the FPGA begins to operate. The bits in the configuration-memory instruct each piece of FPGA how to operate.

4. HDL is Object-Based not Object-Oriented

A computer language is object-oriented if it supports the four specific properties called data abstraction, encapsulation, polymorphism, and inheritance. Data abstraction is the process of recognizing and focusing on important characteristics of an object and leaving out the un-wanted characteristics of that object. Encapsulation is achieved by making the attributes private while creating public methods that can be used to access those attributes. Inheritance allows the user to extend classes (called subclasses) from other classes (called superclasses). Polymorphism allows the programmer to substitute an object of a class in place of an object of its superclass.

Object-based hardware description languages are similar to object-oriented programming languages, except that they do not have inheritance [41] and run-time polymorphism as shown in Table 1.

	Object Oriented feature	HDL feature	Explanation
Data abstraction	Yes	Generic map Port map	The top-level design is a generic module in a hierarchical structure. Instantiations occur at the lowest level. The top-level design has the highest level of data abstraction.
Encapsulation	Yes	Entity, architecture	Entity defines external view (I/Os) of a model; architecture defines the function (behavior) of a model as a black box. Both are private and used by a particular module (design).
Compile-time polymorphism	Yes	Operator overloading	Operators are overloaded (given multiple functionality) by defining a function whose name is the same as the operator itself. This is a static compile-time feature versus dynamic run-time feature.
Run-time Polymorphism	No	None	None
Inheritance	No	None	None

Table 1. Hardware description language features

Generic components and instantiation are typical for object-based languages. Generics allow the components to be customized upon instantiation. Examples of generic uses are customized timing and alteration of array size. The value of a generic component specified for an instance is constant for that instance.

5. Two Primary Hardware Description Languages—VHDL and Verilog

Very high speed integrated circuit (VHSIC) hardware description languages VHDL and Verilog (Verify Logic) are two hardware description languages for coding models of a digital system which possibly will be implemented on an FPGA chip. VHDL and Verilog are not only used for FPGA software design, but are also used for simulation, synthesis, documentation, and requirements. Simulation is to check the behavior of the design for certain input conditions before implementation; synthesis is to turn the high-level code to a low-level gate netlist (a netlist represents the connectivity of an electronic design) for programming a particular chip; documentation and requirements are to guide the FPGA software design, and provide a common platform for communication among all stakeholders [42].

Each VHDL program contains two major language constructs—entity and architecture. An entity section describes the interface of the component (inputs/outputs).

An architecture section describes the operation (behavior) of the component. The basic building block of Verilog is the module statement. It is somewhat analogous to defining a function in C language. Each module has a name, ports list (inputs/outputs), and components (operations). Table 2 shows some basic construct differences between VHDL and Verilog programming languages [43].

	HDL feature	Explanation
Structure	entity-architecture	module ( )
Physical interconnect that communicate between processes	signal	wire or register
Variables	variable	wire or register
assignment	a <= b;	assign a = b;
Basic unit of execution	process(sensitivity lists)	always@(sensitivity lists)

Table 2. Some VHDL and Verilog construct differences

There are three methods in programming an FPGA with VHDL or Verilog: data flow, behavioral, and structural. The data-flow method uses statements to define the actual flow of data from one component (register) to another in concurrency. Table 3 shows a data-flow method in Verilog.

```
// declare and name a module (design blocks); list its ports.

module mux_2_to_1(a, b, out,outbar, sel);

input a, b, sel;                 //Specify each port as input, output, or inout

output out, outbar;

// Express the module's behavior. Each statement executes in parallel

assign out = sel ? a : b;     // a data flow statement, out = sel • a + ‾sel • b

assign outbar = ~out;         // a data flow statement, outbar = ‾out

endmodule                        // Conclude the module code.
```

Table 3. Data flow method in Verilog

"***Assign out=sel? a:b***" and "***assign outbar = ~out***" are data-flow statements to be executed concurrently. "***Assign out=sel? a:b***" means that if ***sel***=1 then ***out***=a, and if

sel=0 then ***out***=b. Basically this is a multiplexer with two inputs and one output; signal ***sel*** determines which input (***a*** or ***b***) connects to the output (***out***). "***Assign outbar = ~out***" means that ***outbar*** is the inverted ***out***.

The behavioral method uses statements to describe a sequential algorithm if an event is triggered. Table 4 shows a behavioral method in Verilog. When a signal (***a***, ***b*** or ***sel***) changes, the statements inside the "always" block will be executed sequentially.

```
module mux_2_to_1(a, b, out, outbar, sel);
input a, b, sel;                        // see last example for explanation
output out, outbar;
reg out, outbar;
// the always block runs once whenever a signal in the sensitivity list changes value
always @ (a or b or sel)
// Statements within the always block are executed sequentially.
begin
   if (sel) out = a;          // if sel is true, then out = a, else out = b
  else out = b;
  outbar = ~out;              // outbar = ¬out
end
endmodule
```

Table 4. Behavior method in Verilog

Structural methods express the design as an arrangement of interconnected pre-defined components designed by data-flow and/or structural methods. Figure 8 shows the relationship among data-flow, behavioral and structural methodologies.

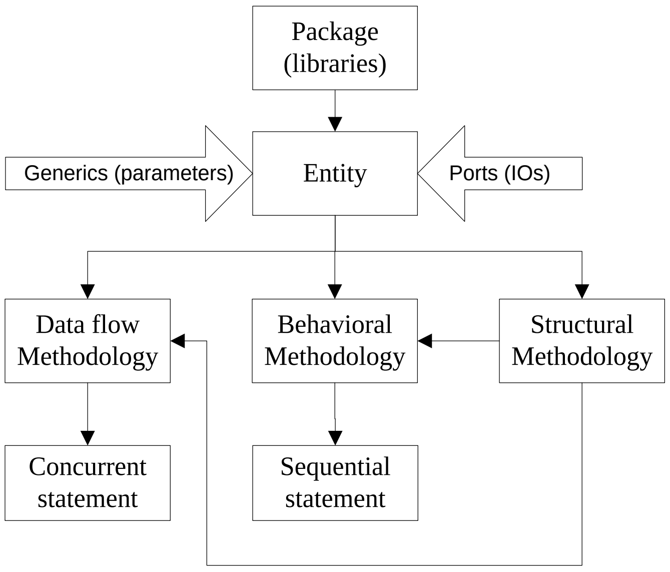


Figure 8 HDL hierarchy

6. FPGA Design Methodology—HDL Approach

FPGA design by using hardware-description languages adopts top-down methodology as shown in Figure 9 with a hierarchical and modular approach defined at different levels of abstraction [44,45].The design flow has four stages [46].Simulation and validation can be performed at all four.

1. System level: Specifications are given.
2. Behavior level: Design is described in texted algorithms.
3. Register transfer level (RTL): Design is described in components. RTL stands for register transfer language, a language for describing the behavior of computers in terms of stepwise register contents.
4. Physical level: Design is described in target hardware.

Traditional standard FPGA design flows are shown on the right hand side in Figure 9. The most important disadvantage of this individual approach is that each stage is addressed separately. This often involves the use of different computer aided design tools, software platforms, and environments at various stages.

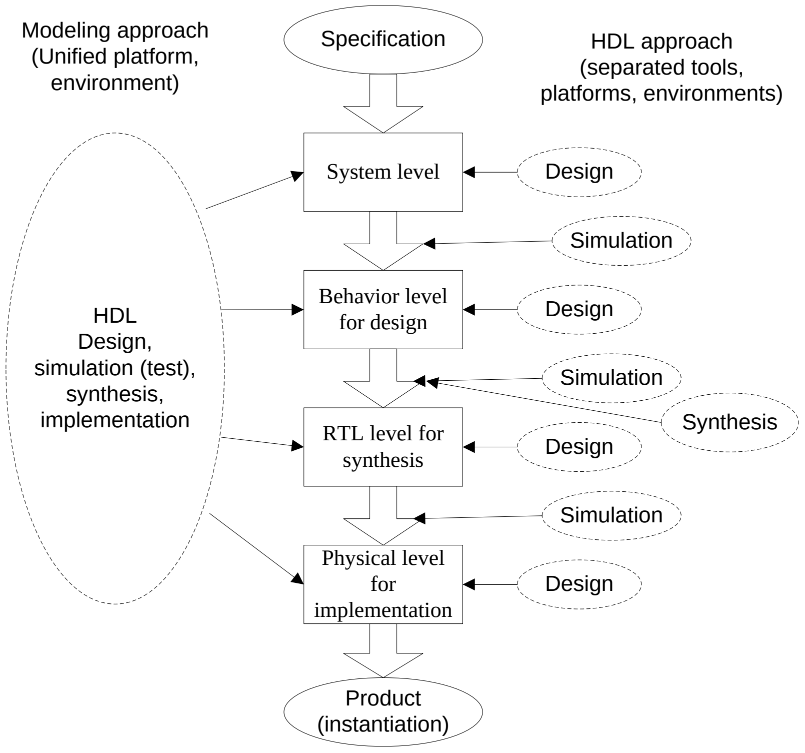

Modeling approach (Unified platform, environment)
Specification
HDL approach (separated tools, platforms, environments)
System level
Design
Simulation
Behavior level for design
Design
HDL Design, simulation (test), synthesis, implementation
Simulation
Synthesis
RTL level for synthesis
Design
Simulation
Physical level for implementation
Design
Product (instantiation)

Figure 9 Hierarchic flow of the top–down design method.

A holistic system-level approach to the FPGA design and development enables a top-down design methodology in a single unique environment as shown on the left hand side in Figure 9. It starts with modeling an idea at an abstract level, and proceeds through the iterative steps to refine this idea into a detailed system. A test environment is developed simultaneously to check if the design is in compliance with the original specifications. Concepts are tested before final physical implementation [46].

Table 5 describes the steps from design to FPGA implementation and some available software tools in today's market [47].

FPGA design flow	Xilinx tools	Synopsys tools (Synplicity tools)	Others
Design entry (create source files)	VHDL or Verilog (text-based) Core generator for reuse	Synphony®	CoreFire® (model-based) by Annapolis Micro Systems Simulink®/MATLAB® (model-based)
Functional simulation (test design functionality)	ISIM®	VCS	ModelSim® by Mentor Graphics NC – Verilog by Cadence
Synthesis (convert VHDL to netlist)	XST®	Synplify Synplify Pro Synplify Premier	
Timing simulation (check timing)	ISIM®	Synplify Synplify Pro Synplify Premier	
Constraint (provide timing and placement requirements)	FPGA editor PlanAhead	Synplify Synplify Pro Synplify Premier	
Implementation Use netlist and constraints to create programming bit files	FPGA editor ISE place-and-route		
Implementation analysis (debugging)	ChipScope	Synplify Synplify Pro Synplify Premier Identify	Debussy by Novas
Implementation improvement (change design and/or constraints)	FPGA editor PlanAhead	Synplify Synplify Pro Synplify Premier	
Programming (program design on a chip)	iMPACT	confprosh	

Table 5. FPGA design tools survey

E. DESIGN PATTERNS

There are design patterns for software, firmware, and hardware of embedded systems. In this section, we list some major patterns for each type. Design patterns are more important for firmware and hardware because of the cost of making modifications.

1. Software Design Patterns

To have good modularity and reusability in software design, an object-oriented language itself is not sufficient; higher-level building blocks (design patterns) are highly desirable. A design pattern is a typical solution to a recurring problem in a software system. According to Gang of Four (GoF), there are 23 major software design patterns generally considered as the foundation for all other patterns in object-oriented design [48].These patterns are categorized as creational, structural, and behavioral as shown in Table 6.

Groups	Patterns	
Creational Patterns	1	Abstract Factory
	2	Builder
	3	Factory Method
	4	Prototype
	5	Singleton
Structural Patterns	6	Adapter
	7	Bridge
	8	Composite
	9	Decorator
	10	Facade
	11	Flyweight
	12	Proxy
Behavioral Patterns	13	Chain of Resp.
	14	Command
	15	Interpreter
	16	Iterator
	17	Mediator
	18	Memento
	19	Observer
	20	State
	21	Strategy
	22	Template
	23	Visitor

Table 6. Software design patterns

2. Embedded Software Design Patterns

In general, embedded software (or firmware) is based either on a CPU or on reconfigurable computing. The key components in a CPU-based embedded system are microprocessors and microcontrollers. The key components in a reconfigurable-computing embedded system are FPGAs and complex programmable logic devices (CPLDs).

a. CPU-Based Firmware Design Patterns

Table 7 contains 28 typical firmware design patterns for CPU-based embedded systems [49].

b. Reconfigurable Computing Based Design Patterns

Table 8 contains 89 firmware design patterns grouped into five classes and twelve subclasses for reconfigurable computing systems. These patterns are based on the existing literature collected by André DeHon et al. in 2004 [50].

One example is the coarse-grained time multiplexing design pattern, number 3 in Table 8 which allows a large design to be run on a smaller or fixed-capacity reconfigurable-computing platform. Another example is template specialization design pattern (number 41 in the Table 8) which implements a specialized computation instead of a generic computation in the reconfigurable computing to reduce space and/or time to meet special requirements for a task [50].

Class	Subclass	
Object	1	Half call
	2	Manager
	3	Resource manager
	4	Message factory and message interface
	5	Publish-subscribe
State	6	Hierarchical state machine
	7	State machine inheritance
	8	Collector state pattern
	9	Parallel wait state pattern
	10	Serial wait state pattern
Hardware interface	11	Serial port
	12	High speed serial port
	13	Hardware device
	14	Synchronizer
Protocol	15	Transmit protocol handler
	16	Receive protocol handler
	17	Protocol packet
	18	Protocol layer
	19	Protocol stack
Architecture	20	Processor architecture patterns
	21	Processor architecture patterns II
	22	Feature coordination patterns
	23	Task design patterns
	24	Resource allocation patterns
	25	Timer management
Implementation	26	C++ header file include patterns
	27	STL design patterns
	28	STL design patterns II

Table 7. Design patterns for embedded systems

Class	Subclass	Expression		Implementation	
Area-time tradeoffs	Basic			1	Sequential vs. Parallel
				2	Fine-grain Time-Multiplexing
				3	Coarse-grain Time-Multiplexing
				4	Element Share Regular Graphs
				5	Operator Share General Graphs
				6	Synthesis Objective
				7	Scheduled Operator Sharing
				8	Data path Sizing and Serialization
	Parallel	9	Extract Control Flow	17	If-Conversion/Predication
		10	Dataflow	18	Parallel Prefix, Reduce, Scan
		11	Synchronous Dataflow	19	SIMD
		12	Acrylic Dataflow Graph	20	Vector
		13	Functional	21	Data path Duplication
		14	Data Parallel	22	Communicating FSMDs
		15	Multithreaded	23	Direct Implementation of Graph
		16	Futures		
	Processor-FPGA			24	Interfacing/IO
				25	Co-processor
				26	Streaming Co-processor
				27	Instruction Augmentation
				28	Sequencer/Controller
	Common-Case			29	Caching
				30	Simple Hardware with Escape
				31	Exception
				32	Trace-Schedule/Exceptional Exit
				33	Prediction
				34	Speculation
				35	Parallel Verifier
Reducing area or time	Reuse hardware			36	Pipelining
				37	Wave Pipelining
				38	Retiming
				39	C-Slow
				40	Software Pipelining
	Specialization	46	Constructor	41	Template
				42	Worst-Case Footprint
				43	Constructive Instance Generator
				44	Instance Generator
				45	Partial Evaluation
	Partial reconfiguration			47	Isolate Fixed/Varying
				48	Constant Fill-in
				49	Unify Data path Variants

Class	Subclass	Expression		Implementation	
				50	1D Function Space
				51	Fixed-Size and Std. IO Page
				52	Bus Interface
Communications	Basic	53	Streaming Data	67	Shared Bus
		54	Message Passing	68	Token Ring
		55	Remote-Procedure Call	69	Reconfigurable Interconnect
		56	Shared Memory	70	Pipelined Interconnect
				71	Serialized Communications
				72	Time-Switched Routing
				73	Circuit-Switched Routing
				74	Packet-Switched Routing
	Layout	62	Cellular Automata	62	Cellular Automata
		63	Systolic, Semi-systolic	63	Systolic, Semi-systolic
				64	Fixed-Radius Communication
				65	Folded/Interleaved Torus
				66	Tree-of-Meshes and Fold-and-Squash
	Synchronization			57	Synchronization Synchronous Clock
				58	Asynchronous Handshaking
				59	Tagged Data Presence
				60	Queues with Back Pressure
				61	H-Tree
Memory	Value-added			75	Address Generator
				76	Content-Addressable Memory
				77	Read-Modify-Write
				78	Data Filter
				79	Indirection/Redirection
				80	Scan-Select-Reorganize
				81	Data Compression/Digest
				82	Stack, Queue
				83	Data Structure
Numbers and functions	Representation	87	Abstract operators	84	Parameterize Data path Operators
				85	Redundant Number System
				86	Distributed Arithmetic
				88	Stochastic Bit-Serial Computation
				89	Bit-Slice Data path

Table 8. Design patterns for reconfigurable computing

3. Hardware Components Selection

Component selection is a process of selecting a suitable component or a set of similar components from different suppliers for the designed circuit to perform its intended operation [51]. Some important generic factors for consideration are:

- Availability (lifecycle)
- Affordability (cost)
- Traceability (component's history)
- Reliability of the component's performance over a period of time

The important technical parameters in selecting an electronic component can be categorized into electrical, mechanical, and environmental. Table 9 lists these three categories and their associated major parameters.

Categories	Parameters	Example
Electrical parameters	Component category	Microcontroller
	Key attribute 1	Speed
	Key attribute 2	Peripherals
	Key attribute 3	Program memory size
	…	
	Key attribute N	Core size
Mechanical parameters	Mounting type	Surface mount
	Number of terminals or pin count	8
	Package dimensions	(2.2~3.2) × (2.7~3.1) mm
	Package style	SOT-23
	Pin diameter	0.2~0.51 mm
	Pin style	Gull wing
	Seated height	0.9~1.45 mm
	Weight	
	Terminal spacing	0.95 mm
Environmental parameters	Standards conformance	ASME Y14.5M (see below)
	Derating temperature	
	Material composition	Silicon
	Moisture sensitivity levels (MSL)	
	Operating and storage temperature ranges	0° ~70°C
	Peak reflow temperature	
	Radiation effect	RoHS compliant (see below)
	Temperature grades	Commercial

Table 9. Electronics component selection

A gull wing device is a surface mount component that has its pins leads folded out from its body in the shape of an "L." ASME Y14.5 standard is considered the authoritative guideline for the design language of geometric dimensioning and tolerancing [52]. RoHS stands for Restriction of Hazardous Substances Directive 2002/95/EC; it was adopted in February 2003 by the European Union to restrict the use of certain hazardous substances in electrical and electronic equipment [53].

F. A* AND AO* ALGORITHMS

Created by Peter Hart, Nils Nilsson and Bertram Raphael in 1968, the A* algorithm finds the least-cost path from an initial node to the goal node. This can be accomplished by using a best-first search using the estimated total path cost. Best-first always chooses the path with lowest estimate until the goal is reached. The estimated total path cost is defined as the sum of the actual cost from the initial node to the current node and the estimated (heuristic) cost from the current node to the goal node. An A* tree only contains ***OR*** nodes (disjunctions).

If the heuristic cost is always less than the subsequent actual cost to the goal (that is, we never overestimate or we always underestimate the heuristic cost), then the solution is guaranteed to be optimal. This is the definition of admissibility of the heuristic cost. If the heuristic cost at node n is always less than the sum of the heuristic cost at node n' and the actual cost from node n to n' (suggesting the triangle inequality that the sum of the lengths of any two sides must be greater that the length of the remaining side in a triangle), then the first path found to the goal is guaranteed to be the best.. This is the definition of consistency. If b is the maximum branching factor (the number of children at each node) and d is the depth of goal (solution), an upper bound on the number of nodes visited by an A* search is b^d. We summarize the definition and properties of A* algorithm in Table 10.

AO* is similar to A* algorithm except that it has conjunctions as well as disjunctions for branches. The nodes in conjunction must be all true; as a result, the estimated cost for a conjunction is the sum of all nodes in this conjunction, or $f(n) = f(n_1)$

$+ f(n_2) + \ldots + f(n_k)$ where n_1, n_2, ….. , n_k are the nodes in conjunction; $f(n_i)$ is the estimated cost for node i in this conjunction, where i is between 1 and k.

Term	Expression	Definition
A* search	$f(n) = g(n) + h(n)$	$f(n)$ = estimated cost from node n to goal, $g(n)$ = actual cost from root to node n, $h(n)$ = heuristic cost from node n to goal
Admissibility	$h(n) \leq h^*(n)$	$h(n)$ = heuristic cost, $h^*(n)$ = actual cost.
Consistency	$h(n) \leq c(n,n') + h(n')$	$h(n)$ = heuristic cost at node n, $h(n')$ = heuristic cost at node n', $c(n,n')$ = actual cost from node n to n'

Worst case time and space complexity	b^d	b = branching factor,
		d = depth of solution

Table 10. A* algorithm

III. METHODOLOGY

A. SW/FW/HW PARTITIONING METHODOLOGY

Software/firmware/hardware partitioning is a difficult task; oftentimes it depends on an expert's knowledge [7, 30]. In this chapter, we propose a procedure to systematically partition components into software, firmware or hardware, and then map partitioned components unto appropriate design patterns for implementation. The procedure is:

1. Develop Requirements

Write down the requirements.

2. Define Constraints

Define constraints on components. Constraints generally come from requirements and they include (but are not limited to):

- Constraints on input and output signals that cannot be changed
- Minimum speeds of processing
- Synchronization necessary within specified windows
- Constraints on use of proprietary software/firmware/hardware
- Others

3. Form an Architecture

This step can be divided into three sub-steps below.

a. Consider Design Options

First, list all design options and associated design problems; second, link design problems to requirements and constraints in steps one and two; third, disqualify the designs with problems and select the optimal design among qualified ones. Domain knowledge is helpful in qualifying and disqualifying design options.

b. Form an Architecture for the Optimal Option

Draw a block diagram of functional components and their data connections without presumptive software/firmware/hardware partitioning.

c. Simplify Architecture for Partitioning Analysis

A component can be eliminated from further software/firmware/hardware partitioning analysis if it has only one feasible choice from the set of {software, firmware, hardware}.

4. Build a Tree to Assign Modalities to Functional Components

a. Software, Firmware and Hardware Partitioning

Build a tree of possible mappings from the remaining functional components to the set of {software, firmware, hardware} for each mapping that fulfills the requirements and constraints. Note that interface components may be needed between connected components.

Starting at the root, expand it into $3 \times n$ child nodes at the first level of expansion, where n is the number of functional components in the architecture without being assigned to certain modalities. At this level, (1) each node has an embedded AND of n functional components; (2) each component could be mapped to three modalities in the set of {software, firmware, hardware}; (3) only one functional component is assigned to a modality and others are don't-cares. Therefore, there are $3 \times n$ child nodes (successors) from the root. Only the child nodes satisfying requirements and constraints are considered for further expansion; unqualified nodes are terminated by being assigned infinite costs. Among the qualified nodes, an optimal one is selected for further expansion to the second level; this optimal node inherits all requirements and constraints from the parent node (root) [54]. But we must allow for the possibility of backtracking to the other choices if the expansion of the original node is disappointing.

At the second level, since one component is already assigned to a modality, there are $n-1$ functional components must be mapped in the node selected at level one. For the same reason as the first level, this selected node (now a parent node) can be expanded into $3×(n\text{-}1)$ child nodes. Use the same process as level one to select an optimal node among these $3×(n\text{-}1)$ child nodes. If the estimated cost of this selected node at level two is not the least estimated cost among all qualified nodes, we must move our search to the node with the least estimated cost anywhere in the tree, consistent with the A* algorithm. Repeat this process until all n components are assigned to certain modalities. Because there are *n* functional components, there are *n* levels of expansion from the root to the final solution.

Our methodology specializes the A* algorithm in the following ways.

- Nodes are disqualified if they violate requirements or constraints. Infinite costs are assigned to disqualify nodes; we do not want to revisit disqualified nodes.
- Among the qualified nodes, we use cost estimation to find the least-cost node for further expansion. If there are ties, we will select a node for expansion by alphanumeric order.
- We are interested in feasible solutions, not necessarily the best solutions. Feasible solutions meet our requirements and constraints at reasonable costs. Ultimately, the stakeholders will select the best solution among the feasible solutions.
- The tree is an ***OR*** tree with embedded ***AND***. The advantage of having an ***OR*** tree with embedded ***AND*** is that we can simplify the tree and use ***A**** search instead of more complicated ***AO**** search algorithm.
- The branching factor is a constant which is the number of modalities.
- The number of levels of expansion is the number of components in the architecture without being assigned to certain modalities.

The least number of nodes in the tree using this methodology occurs when there is no need to backtrack. That is $3×n×(n+1)/2$, because there are $3×n$ mappings at the first level, at least $3×(n\text{-}1)$ mappings at the second level, at least $3×(n\text{-}2)$ mappings at the third level, and so on. The sum of this arithmetic series is $3×(1+2+3+... + n)$. In general, *3* can be replaced by the number of modalities in a set, so that the least number of mapping can be expressed as (#modalities) × (#components) × (#components+1) */ 2*. Using brute-force exhaustion methodology, the most number of mappings is 3^n, because each functional component has *3* possible modalities in the set of {software, firmware, hardware} and there are *n* functional components; $3×3×3×3×…×3=3^n$. Table 11 shows some comparisons between these two methodologies. It is expected that many problems will have close to the least number of mappings since in many real-world problems the costs of alternatives are quite different and revisiting other qualified nodes is not required.

#functional components	Least number of mappings using our methodology when backtracking is unnecessary	Most number of mappings (exhaustion methodology)	Ratio of Most/Least
1	3	3	1
5	45	243	5
10	165	59,049	358
100	15,150	5.E+47	3.E+43

Table 11. Least and most numbers of mappings

b. Design Patterns Mapping

Once all components are partitioned as software, firmware or hardware, we can map each component to a design pattern if existing, and otherwise synthesize a new one. The procedure of mapping design patterns is the same as step 4a; simply replace modalities {software, firmware and hardware} with the existing design patterns.

c. Rate the Cost

Rate the cost of each mapping, including costs of the interfaces, and including a weighted sum of the following factors:

- Monetary costs of the equipment
- Execution time
- Power required
- Space required
- Design complexity
- Monetary costs of necessary further development
- Cost of debugging the implementation
- Degree of lack of satisfaction of the ultimate user needs
- Others

The cost estimation is not critical for our methodology in this dissertation since (1) requirements and constraints are used to qualify or disqualify options, and an infinite cost is assigned to a disqualified option; (2) among the qualified options, the implementation costs for software, firmware and hardware vary significantly and the cost comparisons among them are apparent. Typically, the cost for hardware implementation is in hundreds of thousands of dollars, a firmware implementation is in thousands of dollars, and a software implementation is in hundreds of dollars (explained in Table 15); (3) in terms of design patterns mapping, each design pattern is unique and multiple successful mappings are not likely.

5. Repeat Steps 3 through 5

If the optimal solution is not detailed enough, return to step 3 and refine it further.

6. Repeat Steps 4 through 6

If more feasible solutions are required, return to step 3.

7. Repeat Steps 1 through 7

If there are no solutions, return to step 1 and modify requirements, constraints and/or modalities until one or several feasible solutions are found. Ultimately, stakeholders will decide which solution is optimal. Figure 10 shows the process flow.

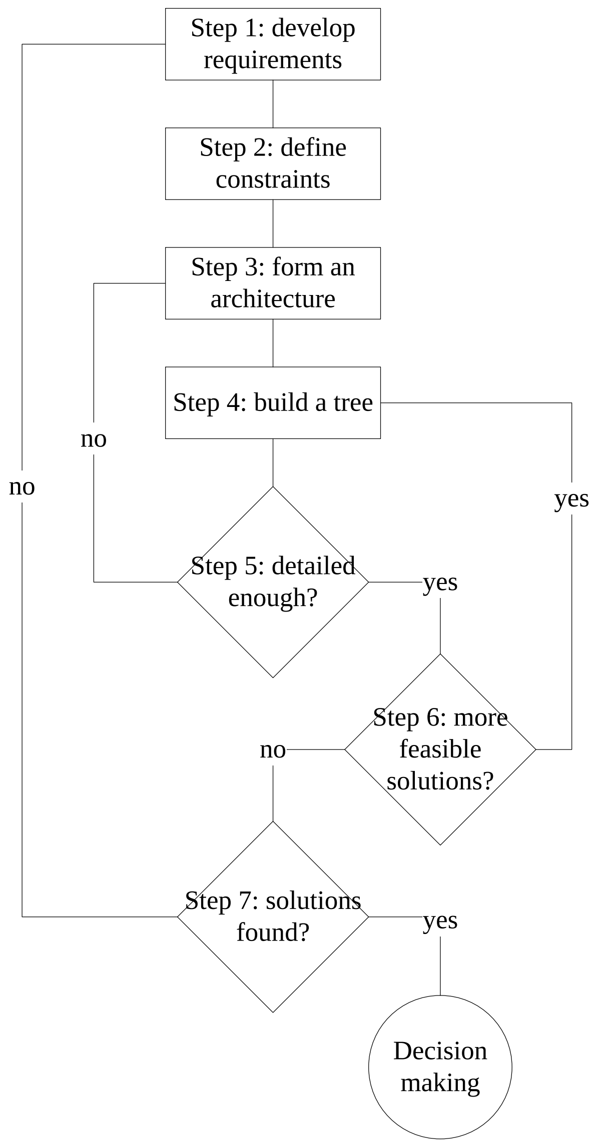

Step 1: develop requirements
Step 2: define constraints
Step 3: form an architecture
Step 4: build a tree
Step 5: detailed enough?
Step 6: more feasible solutions?
Step 7: solutions found?
Decision making
no
no
no
yes
yes
yes

Figure 10 Software/firmware/hardware codesign methodology process flow

B. EXAMPLE—FILTERING FOR ULTRA-HIGH FREQUENCY SIGNALS

1. Develop Requirements

As an example, consider a filtering task for ultra-high radiofrequency signals. We summarize the requirements in Table 12.

	Description	Comment
R1	Input signal frequency range	0-6 gigahertz analog format
R2	Output signal frequency range	0-6 gigahertz analog format

R3	Filtering	Apply specified filtering;
		frequency dependent preferable;
		must be reconfigurable for future use

Table 12. Functional requirements

2. Define Constraints

Non-functional constraints are summarized in Table 13.

	Description	Comment
C1	Latency (real-time)	Less than 5 microseconds for signals going through the filter.
C2	Throughput	Up to 120 gigabits-per-second.
C3	Flexibility	Must be adapted for various data rates (up to 12 gigabytes-per-second).
C4	Synchronization	Parallel data bits must be synchronized (aligned) before digitally filtering the signal they represent as a whole (if digital filtering is used.)
C5	Environment	Not important since the application is in a laboratory (equipment is used to test electronics to be put in planes, but is not in the planes physically).
C6	Development time	Less than a year.
C7	Material budget	Depending on the availability of capital investment property funding ($250K typically).
C8	Quantity	One prototype for feasibility test.
C9	Temperature	Room temperature (70º±5º F).
C10	SWaP (size, weight, and power)	Not important since the application is in a laboratory (see C5 for explanation).
C11	Degree of consistency	Only one data alignment per day.
C12	Information completeness	Information cannot be lost.

Table 13. Non-functional constraints

3. Form an Architecture

a. Consider Design Options

Table 14 lists five design options for filtering and their associated design problems. The first option, using a single analog filter, violates the third requirement and the third constraint, since an analog filter cannot be easily reconfigured (being not flexible). The second option, using only an analog-to-digital converter, a digital-to-analog converter and a digital filter, violates the first and second requirements, since the digital filter cannot handle signals in the gigasamples-per-second range. The third option, subsampling the digitized stream at different frequencies, violates the first constraint, since it takes too much time to switch from one frequency to another to cover a wide spectrum. The fourth option, sampling for a fixed number of samples and transferring them into a buffer for processing, violates the first and twelfth constraints, since information will be lost while processing data in the buffer; this will cause aliasing. Aliasing means frequency ambiguity due to insufficient sampling [55]. The fifth option, using an analog-to-digital converter, a digital-to-analog converter, a digital filter, a multiplexer and a demultiplexer, has no violations; as a result, we select option five for our design as shown in Figure 11

	Design options for filtering	Design problems	
O1	A single analog filter	The filtering task is inflexible. It requires hardware redesign for certain type of filtering.	R3, C3
O2	ADC, DAC, digital filter without demultiplexing and multiplexing	This only works for low-frequency signals, since the processing speed is limited by the digital filter	R1, R2
O3	subsample the digitized stream at different frequencies rather than different phases	The overall frequency bandwidth is too narrow. With the help of down-converters, the switching time from one frequency to another might violate the minimum latency requirements.	C1
O4	Sampling for a fixed number of samples, transferring them to a buffer, waiting a while, then sampling another fixed number of samples	Not real-time, data are missing while processing data in the buffer which could cause aliasing	C1, C12
O5	ADC, demultiplexer, digital filter, multiplexer, DAC	No	

Table 14. Five options for filtering

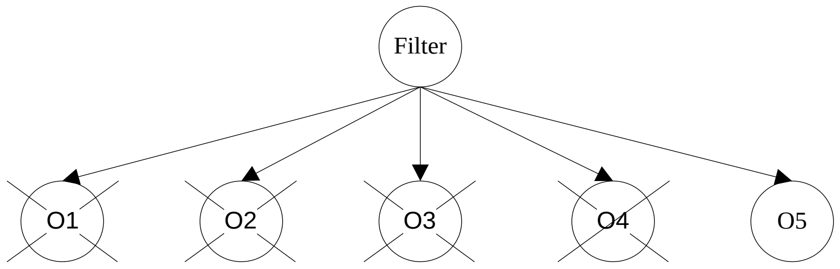


Figure 11 Five options for filtering

b. Form an Architecture for the Optimal Option

Option five is the only architecture meeting the requirements and constraints. For this architecture, we need five components, and they are an analog-to-digital converter, a demultiplexer, a filter, a multiplexer and a digital-to-analog converter. Based on their definitions, the only logical arrangement is shown in Figure 12.

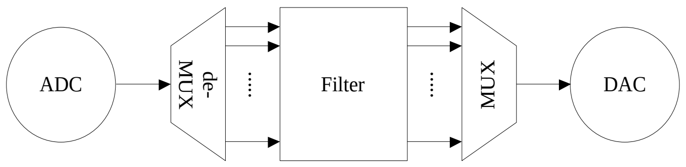


Figure 12 An architecture satisfying our requirements and constraints

c. Simplify Architecture for Partitioning Analysis

An analog-to-digital converter is a device that converts analog signals to digital signals, and a digital-to-analog converter is a device that converts digital signals to analog signals; both of them act like bridges between physical real-world and man-made computer world. As a result, we can exclude the analog-to-digital converter and digital-to-analog converter from our analysis because they must be hardware (see Figure 13.)

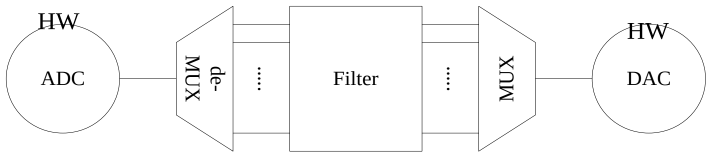


Figure 13 Exclude ADC and DAC from analysis

4. Build a Tree to Map Functions to Modalities

a. Map to Software/Firmware/Hardware

By our methodology there are 18 (3×3×4÷2) mappings to consider. Now we will build an OR tree with embedded ANDs. They are 9 possible branches from the root, and each node is an embedded AND as listed in Table 16. In Table 16, D stands for demultiplexer, F stands for filter, M stands for multiplexer and * stands for don't-care.

Due to non-recurring engineering (NRE) effort, the cost for hardware filtering (hardware implementation) is very high if the number of units used is low [56]. The unit cost for an 8-multicore processor (software implementation) from Texas Instruments is about $300 [57, 58]. The unit cost for a Xilinx Virtex-6 FPGA (firmware implementation) is about $3,200 [59]. The cost comparisons among software, firmware and hardware implementations are listed in Table 15. Node A6, filter being hardware, can be eliminated form our analysis due to its high cost.

Modality	Cost for 5 units	Rating
Hardware (ASIC)	$ 350,150	High-cost
Software (CPU)	$ 1,500	Low-cost
Firmware (FPGA)	$ 16,000	Medium-cost

Table 15. Filter cost rating

We can also eliminate A1, A2, A7 and A8 from our analysis because software and firmware implementations are too slow for ultra-high data rate applications and too costly compared to commercial-off-the-shelf hardware high-speed multiplexers and demultiplexers. The cost for a multiplexer or demultiplexer is less than $50. The estimated costs for other nodes (A3, A4, A5 and A9) are within our budget constraints and considered equal at this stage.

Filter	D	F	M	Violation	Set representation
A1	SW	*	*	A dedicated CPU is required; too slow for high-speed applications	{D=SW,F=*,M=*}
A2	FW	*	*	Speed too slow, cost too high	{D=FW,F=*,M=*}
A3	HW	*	*	OK	{D=HW,F=*,M=*}
A4	*	SW	*	OK	{D=*,F=SW,M=*}
A5	*	FW	*	OK	{D=*,F=FW,M=*}
A6	*	HW	*	Cost is too high for a small quantity of chips	{D=*,F=HW,M=*}
A7	*	*	SW	A dedicated CPU is required; too slow for high-speed applications	{D=*,F=*,M=SW}
A8	*	*	FW	Speed too slow, cost too high	{D=*,F=*,M=FW}
A9	*	*	HW	OK	{D=*,F=*,M=HW}

Table 16. Nine options from the root

We can express Table 16 in a tree structure as shown in Figure 14.

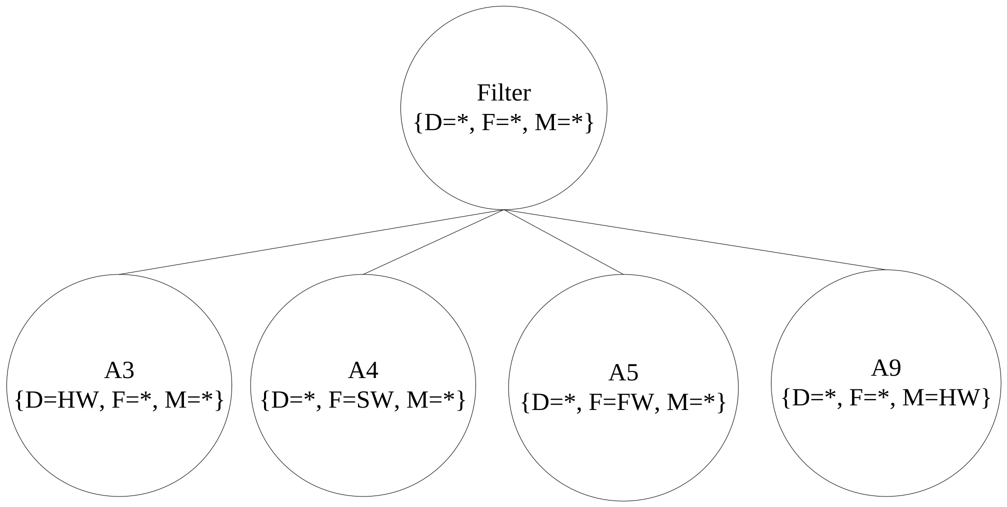


Figure 14 Four possible branches from the root

For the rest of this dissertation, we will simplify the tree representations without including the set symbols as shown in Figure 15.

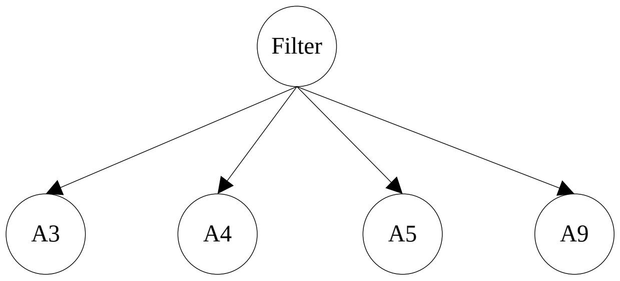


Figure 15 Four possible branches from the root in a simplified form

To satisfy ultra-high frequency, flexible processing, and reasonable cost requirements/constraints, the demultiplexer and multiplexer should be hardware (fast and low-cost), and the filter should be firmware (fast, flexible, and medium-cost) based on Table 15. Searching the tree in Figure 15 from left to right for an optimal solution, A3 is the first node satisfying the requirements/constraints, so we will expand node A3. Because HW is assigned to D in node A3, there are only two components (F and M) to be mapped; this leads to six possible mappings (3×2 = 6). Among these six mappings, 2 nodes, A3-2 and A3-6, meet the requirements/constraints as shown in Table 17. The estimated costs for A3-2 and A3-6 are within our budget constraints and considered equal at this stage.

A3	D	F	M	Violation
A3-1	HW	SW	*	Too slow - violates real-time constraint
A3-2	HW	FW	*	OK
A3-3	HW	HW	*	Cost is too high
A3-4	HW	*	SW	Too slow - violates real-time constraint
A3-5	HW	*	FW	Too slow - violates real-time constraint
A3-6	HW	*	HW	OK

Table 17. Six mappings under A3

Searching the tree in Figure 16 from left to right at the second level, we expand node A3-2. Now there is only one component (M) to be mapped to three modalities because HW and FW are assigned to D and F. A3-2-3 is the only qualified node among these three mappings as shown in Table 18.

A3-2	D	F	M	Violation
A3-2-1	HW	FW	SW	Too slow - violates real-time constraint
A3-2-2	HW	FW	FW	Too slow - violates real-time constraint
A3-2-3	HW	FW	HW	Optimal (only) mapping

Table 18. Three mappings under A32

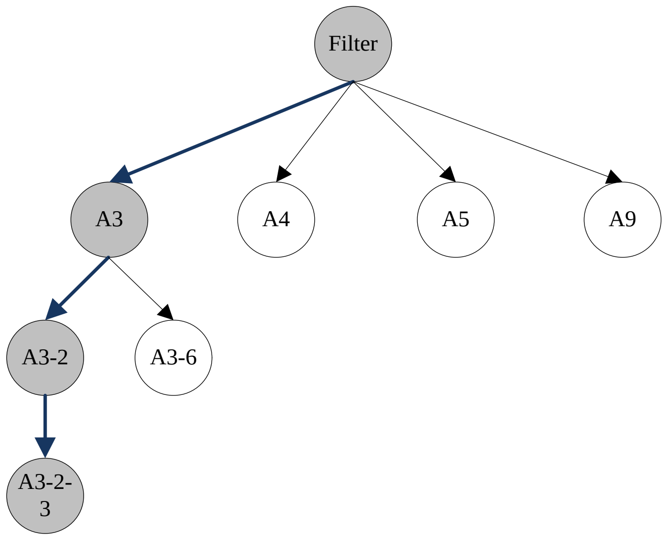


Figure 16 An OR tree for filtering of ultra-high frequency signals

b. Map to Design Patterns

Node A3-2-3 is the optimal solution {D=HW, F=FW, M=HW}. Now we would like to form an architecture for the firmware in node A3-2-3. According to the fourth constraint in Table 13 and the architecture in Figure 12, we need alignment firmware to meet synchronization constraint. Since a demultiplexer and a multiplexer are used, we have to have firmware to re-arrange bits after demultiplexing and before multiplexing. Regarding filtering, we decide to use polyphase DFT filter banks to meet the third requirement in Table 12. The only logical arrangement of alignment, post-demultiplexing, filtering and pre-multiplexing is shown in Figure 17. We use the same methodology to match each component to an optimal existing firmware design pattern; if none existing, synthesize a new one.

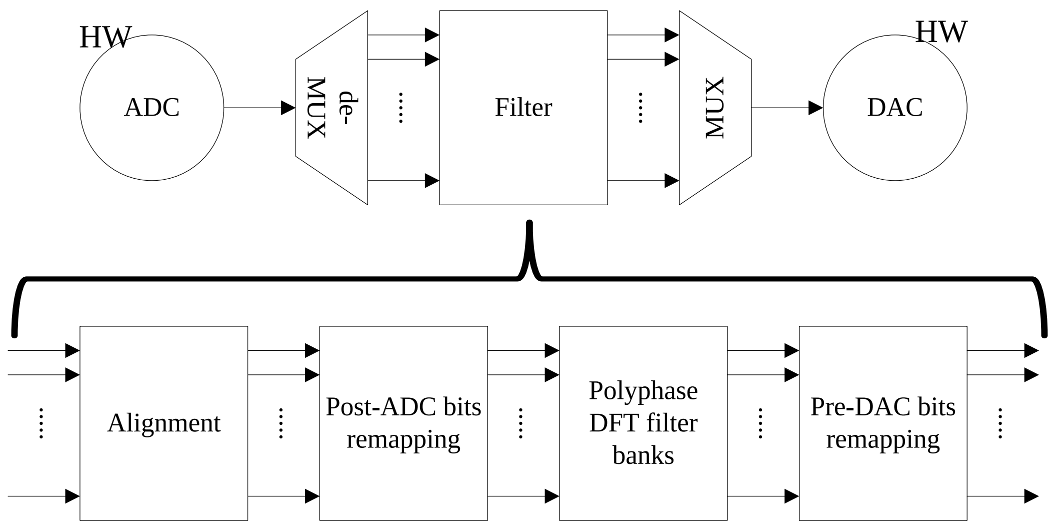


Figure 17 Firmware components

The parallel data bit streams from the demultiplexer must go through alignment, Post-ADC bits remapping, polyphase DFT filter banks and pre-DAC bits remapping algorithms in firmware (shown as four boxes above) to meet our requirements and constraints. Further details of these boxes in Figure 17 are:

- Alignment firmware (AL): When multiple parallel bit-streams arriving at the reconfigurable computing from a demultiplexer, they are misaligned due to different propagation path delays and narrow data windows.

- Post-ADC bits remapping firmware (PA): Bits are not in a proper order for data processing after demultiplexing data from an analog-to-digital converter; consequently, bits must be remapped after demultiplexing.
- Polyphase DFT filter banks (UT): This is application dependent. The polyphase DFT filter banks are used for our particular application.
- Pre-DAC bits remapping firmware (PD): Bits are not in a sequential order after multiplexing multiple parallel data bit-streams for digital to analog conversion; consequently, bits must be remapped before multiplexing.

We will build an OR tree with embedded ANDs (AL•PA•UT•PD) to find the optimal design patterns. According to Table 8 in section E.2.b, Chapter II, there are 89 major design patterns for reconfigurable computing based embedded systems, and they are organized at three levels: (1) class, (2) subclass and (3) purpose.

(1) Level-one (class): After mapping the functional components to firmware in Figure 16 and creating firmware architecture in Figure 17, we would like to further map the firmware algorithms (AL, PA, UT and PD) to the existing 89 design patterns collected by André DeHon [50] if possible. These patterns are organized at three levels; the first level has five classes {ATT, RAT, C, M, NAF} as listed in Table 19. ATT stands for “area-time tradeoffs,” RAT stands for “reducing area or time,” C stands for “communications,” M stands for “memory,” and NAF stands for “numbers and functions.” Among these five classes, ATT and C possibly match our design requirements.

Classes	Descriptions
Area-time tradeoffs class (ATT)	Fitting the logical design to the hardware; Parallelism patterns are considered a subset of ATT
Reducing area or time class (RAT)	Hardware is efficient when it can be reused rapidly.
Communications class (C)	Parallel implementation will be data communication between portions of the computation.
Memory class (M)	Memory bandwidth
Numbers and functions class (NAF)	Allow us to use just as little or as much precision and representation as necessary for the problem

Table 19. Five classes

After four levels of expansion as shown in Table 20 and Figure 18, the first possible solution is B1-1-1-1 = {ATT, ATT, ATT, ATT}. The cost estimates for all possible qualified nodes are within our budget constraints, and considered equal at this stage.

B	AL	PA	UT	PD	Possible match
B1	ATT	*	*	*	Yes
B2	RAT	*	*	*	No
B3	C	*	*	*	Yes
B4	M	*	*	*	No
B5	NAF	*	*	*	No
B6	*	ATT	*	*	Yes
B7	*	RAT	*	*	No
B8	*	C	*	*	Yes
B9	*	M	*	*	No
B10	*	NAF	*	*	No
B11	*	*	ATT	*	Yes
B12	*	*	RAT	*	No
B13	*	*	C	*	Yes
B14	*	*	M	*	No
B15	*	*	NAF	*	No
B16	*	*	*	ATT	Yes
B17	*	*	*	RAT	No
B18	*	*	*	C	Yes
B19	*	*	*	M	No
B20	*	*	*	NAF	No

B1	AL	PA	UT	PD	Possible match
B1-1	ATT	ATT	*	*	Yes
B1-2	ATT	RAT	*	*	No
B1-3	ATT	C	*	*	Yes
B1-4	ATT	M	*	*	No
B1-5	ATT	NAF	*	*	No
B1-6	ATT	*	ATT	*	Yes
B1-7	ATT	*	RAT	*	No
B1-8	ATT	*	C	*	Yes
B1-9	ATT	*	M	*	No
B1-10	ATT	*	NAF	*	No
B1-11	ATT	*	*	ATT	Yes
B1-12	ATT	*	*	RAT	No
B1-13	ATT	*	*	C	Yes
B1-14	ATT	*	*	M	No
B1-15	ATT	*	*	NAF	No

B1-1	AL	PA	UT	PD	Possible match
B1-1-1	ATT	ATT	ATT	*	Yes
B1-1-2	ATT	ATT	RAT	*	No
B1-1-3	ATT	ATT	C	*	Yes
B1-1-4	ATT	ATT	M	*	No
B1-1-5	ATT	ATT	NAF	*	No
B1-1-6	ATT	ATT	*	ATT	Yes
B1-1-7	ATT	ATT	*	RAT	No
B1-1-8	ATT	ATT	*	C	Yes
B1-1-9	ATT	ATT	*	M	No
B1-1-10	ATT	ATT	*	NAF	No

B1-1-1	AL	PA	UT	PD	Possible match
B1-1-1-1	ATT	ATT	ATT	ATT	Yes
B1-1-1-2	ATT	ATT	ATT	RAT	No
B1-1-1-3	ATT	ATT	ATT	C	Yes
B1-1-1-4	ATT	ATT	ATT	M	No
B1-1-1-5	ATT	ATT	ATT	NAF	No

Table 20. Only ATT and C are possible mappings

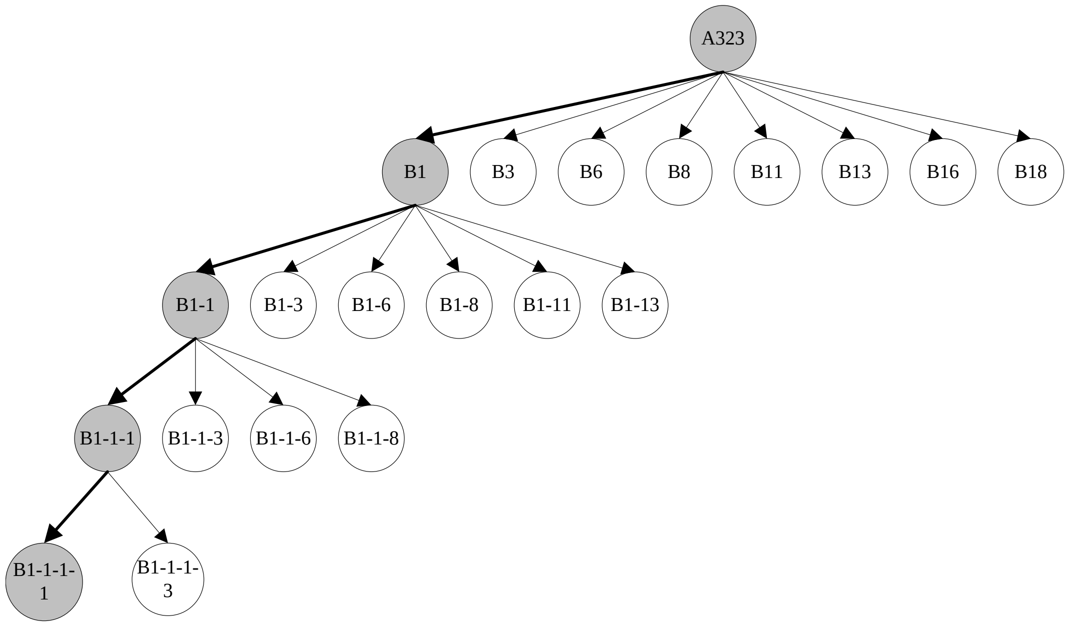


Figure 18 Only ATT and C are possible mappings

(2) Level-two (subclass): Under class area-time tradeoffs (ATT), there are 4 subclasses {B, P, F, C}; B stands for "basic," P stands for "parallel," F stands for "processor-FPGA," and C stands for "common-case." Table 21 lists these four subclasses and their descriptions. Among these four subclasses, only P possibly matches our design requirements.

Subclasses	Descriptions
Basic (B)	fitting the logical design to the hardware
Parallel (P)	use parallelism to increase performance
FPGA processor (F)	use FPGAs and processors together
Common-Case (C)	implement the common-case spatially in minimal hardware and have an escape mechanism to handle the less common cases

Table 21. Four subclasses

After four levels of expansion as shown in Figure 19 and Table 22, the first possible solution is C2-2-2-2 = {P, P, P, P}. The cost estimates for all possible qualified nodes are within our budget constraints, and considered equal at this stage.

C=B1111	AL	PA	UT	PD	Possible match
C1	B	*	*	*	No
C2	P	*	*	*	Yes
C3	F	*	*	*	No
C4	C	*	*	*	No
C5	*	B	*	*	No
C6	*	P	*	*	Yes
C7	*	F	*	*	No
C8	*	C	*	*	No
C9	*	*	B	*	No
C10	*	*	P	*	Yes
C11	*	*	F	*	No
C12	*	*	C	*	No
C13	*	*	*	B	No
C14	*	*	*	P	Yes
C15	*	*	*	F	No
C16	*	*	*	C	No

C2	AL	PA	UT	PD	Possible match
C2-1	P	B	*	*	No
C2-2	P	P	*	*	Yes
C2-3	P	F	*	*	No
C2-4	P	C	*	*	No
C2-5	P	*	B	*	No
C2-6	P	*	P	*	Yes
C2-7	P	*	F	*	No
C2-8	P	*	C	*	No
C2-9	P	*	*	B	No
C2-10	P	*	*	P	Yes
C2-11	P	*	*	F	No
C2-12	P	*	*	C	No

C2-2	AL	PA	UT	PD	Possible match
C2-2-1	P	P	B	*	No
C2-2-2	P	P	P	*	Yes
C2-2-3	P	P	F	*	No
C2-2-4	P	P	C	*	No
C2-2-5	P	P	*	B	No
C2-2-6	P	P	*	P	Yes
C2-2-7	P	P	*	F	No
C2-2-8	P	P	*	C	No

C2-2-2	AL	PA	UT	PD	Possible match
C2-2-2-1	P	P	P	B	No
C2-2-2-2	P	P	P	P	Yes
C2-2-2-3	P	P	P	F	No
C2-2-2-4	P	P	P	C	No

Table 22. Only P is a possible mapping

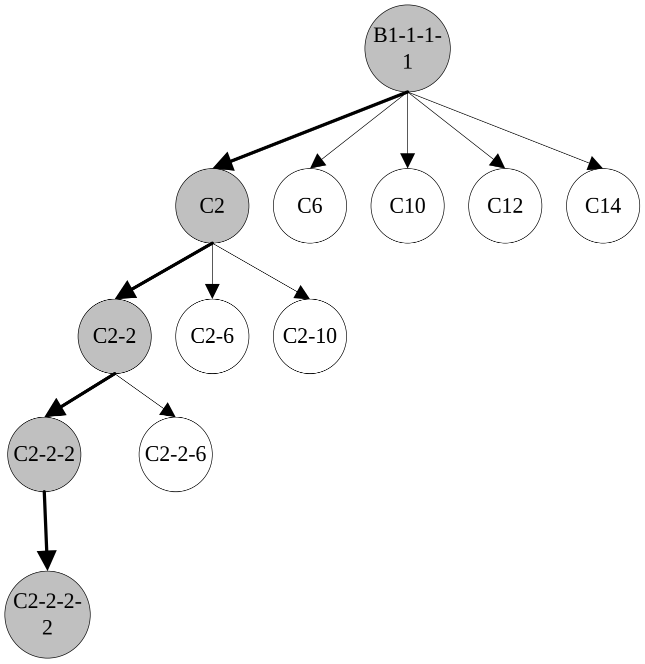


Figure 19 Only P is a possible mapping.

(3) Level-three (purpose): Under subclass parallel (P), there are 6 design patterns {EC, DF, SD, DP, MT, FU}; EC stands for “extract control flow,” DF stands for “dataflow,” SD stands for “synchronous dataflow,” DP stands for “data parallel,” MT stands for “multithreaded,” and FU stands for “futures.” Table 23 lists these six purposes and their descriptions. Among these six purposes, DF, SD and DP possibly match our design requirements.

Purposes	Descriptions
Extract Implicit Parallelism from Control Flow (EC) [Callahan, 2000]	Connecting FPGAs
Dataflow (DF) [Rinker, 2001]	The system consists of an optimizing compiler which produces dataflow graphs, and a dataflow graph to VHDL translator.
Synchronous dataflow (SD) [Lee, 1987]	A method of partitioning of a signal processing task into multiple programs that execute concurrently.
Data parallel (DP) [Hillis et al., 1986]	A series of algorithms appropriate for fine-grained parallel computers with general communications.
Multithreaded (MT) [Caspi et al., 2002]	Dividing a computation up into fixed-size "pages" and time-multiplexing the virtual pages on available physical hardware.
Futures (FU) [Halstead, 1985]	Multilisp is a version of the Lisp dialect Scheme extended with constructs for parallel execution.

Table 23. Eight purposes

After four levels of expansion as shown in Table 24 and Figure 20, the first possible solution is D2-2-2-2 = {DF, DF, DF, DF}. The cost estimates for all possible qualified nodes are within our budget constraints, and considered equal at this stage.

D	AL	PA	UT	PD	Possible match
D1	EC	*	*	*	No
D2	DF	*	*	*	Yes
D3	SD	*	*	*	Yes
D4	AD	*	*	*	No
D5	FN	*	*	*	No
D6	DP	*	*	*	Yes
D7	MT	*	*	*	No
D8	FU	*	*	*	No
D9	*	EC	*	*	No
D10	*	DF	*	*	Yes
D11	*	SD	*	*	Yes
D12	*	AD	*	*	No
D13	*	FN	*	*	No
D14	*	DP	*	*	Yes
D15	*	MT	*	*	No
D16	*	FU	*	*	No
D17	*	*	EC	*	No
D18	*	*	DF	*	Yes
D19	*	*	SD	*	Yes
D20	*	*	AD	*	No
D21	*	*	FN	*	No
D22	*	*	DP	*	Yes
D23	*	*	MT	*	No
D24	*	*	FU	*	No
D25	*	*	*	EC	No
D26	*	*	*	DF	Yes
D27	*	*	*	SD	Yes
D28	*	*	*	AD	No
D29	*	*	*	FN	No
D30	*	*	*	DP	Yes
D31	*	*	*	MT	No
D32	*	*	*	FU	No

D2	AL	PA	UT	PD	Possible match
D2-1	DF	EC	*	*	No
D2-2	DF	DF	*	*	Yes
D2-3	DF	SD	*	*	Yes
D2-4	DF	AD	*	*	No
D2-5	DF	FN	*	*	No
D2-6	DF	DP	*	*	Yes
D2-7	DF	MT	*	*	No
D2-8	DF	FU	*	*	No
D2-9	DF	*	EC	*	No
D2-10	DF	*	DF	*	Yes
D2-11	DF	*	SD	*	Yes
D2-12	DF	*	AD	*	No
D2-13	DF	*	FN	*	No
D2-14	DF	*	DP	*	Yes
D2-15	DF	*	MT	*	No
D2-16	DF	*	FU	*	No
D2-17	DF	*	*	EC	No
D2-18	DF	*	*	DF	Yes
D2-19	DF	*	*	SD	Yes
D2-20	DF	*	*	AD	No
D2-21	DF	*	*	FN	No
D2-22	DF	*	*	DP	Yes
D2-23	DF	*	*	MT	No
D2-24	DF	*	*	FU	No

D2-2	AL	PA	UT	PD	Possible match
D2-2-1	DF	DF	EC	*	No
D2-2-2	DF	DF	DF	*	Yes
D2-2-3	DF	DF	SD	*	Yes
D2-2-4	DF	DF	AD	*	No
D2-2-5	DF	DF	FN	*	No
D2-2-6	DF	DF	DP	*	Yes
D2-2-7	DF	DF	MT	*	No
D2-2-8	DF	DF	FU	*	No
D2-2-9	DF	DF	*	EC	No
D2-2-10	DF	DF	*	DF	Yes
D2-2-11	DF	DF	*	SD	Yes
D2-2-12	DF	DF	*	AD	No
D2-2-13	DF	DF	*	FN	No
D2-2-14	DF	DF	*	DP	Yes
D2-2-15	DF	DF	*	MT	No
D2-2-16	DF	DF	*	FU	No

D2-2-2	AL	PA	UT	PD	Possible match
D2-2-2-1	DF	DF	DF	EC	No
D2-2-2-2	DF	DF	DF	DF	Yes
D2-2-2-3	DF	DF	DF	SD	Yes
D2-2-2-4	DF	DF	DF	AD	No
D2-2-2-5	DF	DF	DF	FN	No
D2-2-2-6	DF	DF	DF	DP	Yes
D2-2-2-7	DF	DF	DF	MT	No
D2-2-2-8	DF	DF	DF	FU	No

Table 24. Only DF, SD and DP are possible mappings

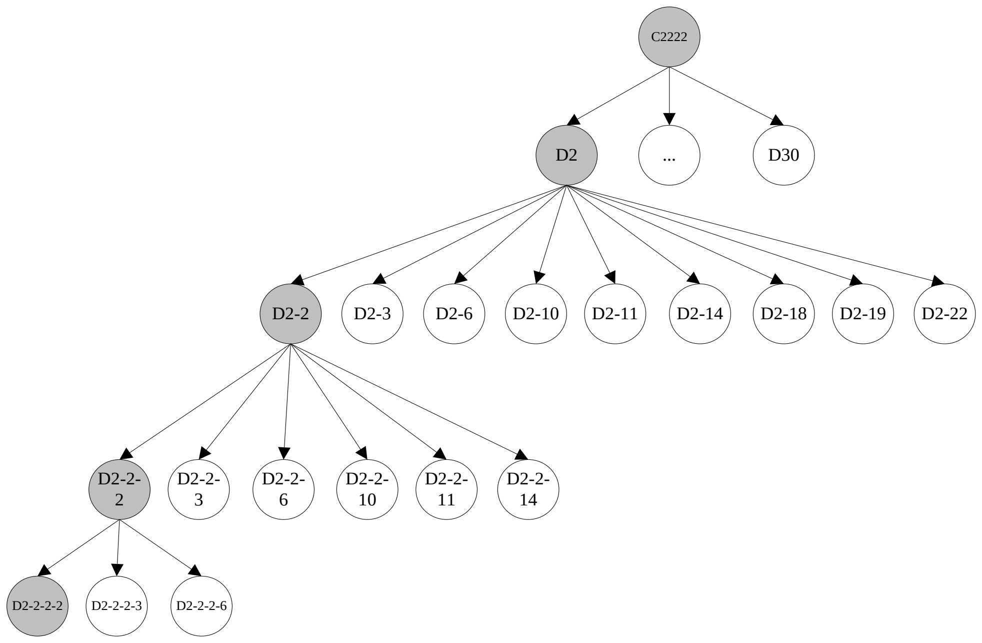


Figure 20 Only DF, SD, and DP are possible mappings

Once a possible optimal leaf-node D2-2-2-2 is identified, we study the design pattern in the published literature carefully [60], and find out that it does not meet our requirements/constraints (see the purpose of dataflow in Table 25.) We will have to use A* algorithm to search all other possible matches. In summary, after exhausting all possible searches, there are only seven patterns which might meet our requirements as listed in Table 25.

Design pattern	Published literature	Purpose
Dataflow (DF)	[60]	The system consists of an optimizing compiler which produces dataflow graphs and a dataflow graph to VHDL translator.
Synchronous Dataflow (SD)	[61]	A method of partitioning of a signal processing task into multiple programs that execute concurrently.
Data parallel (DP)	[62, 63, 64]	A series of algorithms appropriate for fine-grained parallel computers with general communications.
Streaming data (SD)	[65, 66, 67]	Cheops abstracts out a set of basic, computationally intensive stream operations that may be performed in parallel and embodies them in specialized hardware.
Message passing (MP)	[68, 69, 70, 71]	This “Cosmic Cube” computer is a hardware simulation of a future VLSI implementation that will consist of single-chip nodes.
Synchronous clocking (SC)	[72]	VLSI system timing
Tagged Data Presence (TDP)	[73, 74]	The processors are pipelined to support many concurrent processes.

Table 25. Possible optimal leaf-nodes

After reading these seven papers carefully, we conclude that none of the patterns in Table 25 matches our requirements. As a result, we have to synthesize new design patterns; these new design patterns are briefly described in Table 26 (detailed descriptions are in Chapter IV.) We use the term “new design patterns” because they are not among the 89 reconfigurable computing design patterns collected by André DeHon in Table 8 [50]; however, it does not mean that we are the first group invented these patterns.

Design patterns	Description	Among 89 patterns
Application specific: polyphase Discrete Fourier transform filter banks [Vaidyanathan, 1993]	Separate an ultra-wide bandwidth input signal into multiple subbands, process each subband independently and differently, and then combine all subbands into one serial output in an efficient way (Chapter IV)	No
Data bit-streams alignment	Multiple parallel data bit-streams must be aligned with respect to the source-synchronous sampling clocks to ensure correct data sampling at the filter (Chapter IV)	No
Switch-and-filter architecture	Bring an ultra-fast signal from an ADC to a slower device (filter) through a demultiplexer, process the signal, and then output the processed signal to a DAC through a multiplexer (Chapter IV)	No
Post-ADC bits remapping	Data bits must be remapped after demultiplexing for proper data processing (Chapter IV)	No
Pre-DAC bits remapping	Data bits must be remapped before multiplexing for digital to analog conversion (Chapter IV)	No

Table 26. New design patterns descriptions

C. EXAMPLE—FILTERING FOR LOW FREQUENCY SIGNALS

This example is the continuation of the prior example except that the input frequency is low (in megahertz range). To satisfy low-frequency, flexible-processing, and reasonable-cost requirements/constraints, the demultiplexer and multiplexer are most likely unnecessary (parallelism is not required), and filtering can be implemented either by software or firmware. There are two solutions as shown in Figure 21, Figure 22 and Table 27.

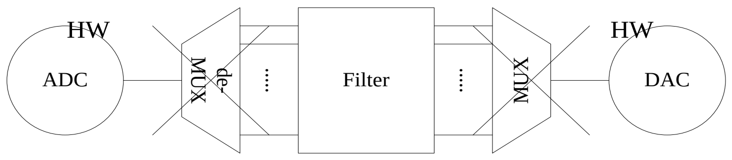


Figure 21 An architecture without parallelism

Filter	F	Violation	Solution
A1	SW	Low-cost	Optimal
A2	FW	Medium-cost	Yes
A3	HW	Cost is too high	No

Table 27. Eliminate A3 from analysis

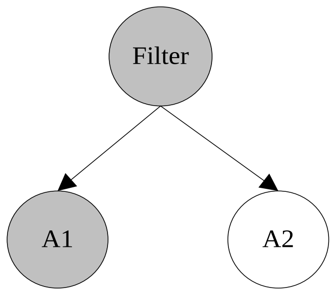


Figure 22 OR tree for low frequency signals

In average, the cost for a Texas Instruments multi-core processor is $300, and the cost for a Xilinx Virtex-6 FPGA is $3,200 with the assumption of using the same analog-to-digital converter and digital-to-analog converter for both solutions. Following decision analysis, a software filter is the optimal solution as shown in Table 28.

	ADC	Filter					DAC	Solution
	HW	SW	FW	HW	Characteristics	Cost	HW	
A1	HW	SW			1s cores, 1s GHz	$300	HW	Optimal
					$300			
A2	HW		FW		100s cores, 100s MHz	$3,200	HW	Yes
					$3,200			

Table 28. Cost association

Apply the same methodology to find an optimal CPU-based embedded system design pattern in Table 7 (section E.2.a, Chapter II) for the problem. We can expand node A1 into one of the six groups {object, state, hardware interface, protocol, architecture, implementation} as shown in Table 29, and then further expand the selected group into a particular subclass as shown in Table 30. The leaf-node B3-1 is the optimal possible solution as shown in Figure 23.

B=A1	Class	Intent
B1	Object	Similar to object oriented language (no match)
B2	State	State machine (no match)
B3	Hardware interface	Hardware device (possible match)
B3	Protocol	Protocol layers management (no match)
B4	Architecture	Processor, timer, resource and task management (no match)
B6	implementation	Standard template library (STL) and header files (no match)

Table 29. Six major groups for CPU-based embedded system design patterns

B3	Subclass	Intent	Possible Match
B3-1	Serial port	The Serial Port design pattern defines a generic interface with a serial port device. The main intention here is to completely encapsulate the interface with the serial port hardware device. All classes interfacing with the serial port will not be impacted by change in the hardware device. (possible match)	Yes
B3-2	High speed serial port	This design pattern covers interfacing techniques with high speed serial communication devices. The main objective is to encapsulate the interface with the device and provide a hardware independent interface to the high speed serial port. (no match)	No
B3-3	Hardware device	The Hardware Device Design Pattern encapsulates the actual hardware device being programmed. The main idea is to encapsulate device register programming and bit manipulation into a single class dealing with the device. (no match)	No
B3-4	Synchronizer	The Synchronizer Design Pattern is used to look at the raw incoming bit or byte stream and detect and align to the frame structure. The frame structure is detected by searching for a sync pattern in the frame. Once the synchronization is achieved, the Synchronizer confirms the presence of the sync pattern in every frame. If the sync pattern in missed a certain number of times, loss of sync is declared. (no match)	No

Table 30. Four subclasses design patterns for hardware interface

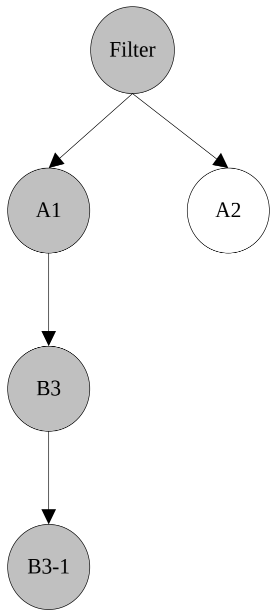


Figure 23 Optimal design pattern mapping

D. EXAMPLE—AIR DATA TEST SET

Consider implementing an air data test set (ADTS) to monitor and simulate air data for altitude and air speed. In each aircraft, there is a static tube and a Pitot tube. The static tube reads the static air pressure (PS), and then translates it into altitude. The Pitot tube reads the moving air pressure (PT), and then calculate the air speed by using the equation: QC = PT–PS. To simulate air data, a PS valve is used to generate static air pressure, and a PT valve (along with the PS valve) is used to generate air speed.

The functional components are listed in Figure 24 and Table 31. There are 16 components, and the number of possible mappings to {SW, FW, HW} is 3^{16}=43,046,721 if we use the brute-force (exhaustion) methodology.

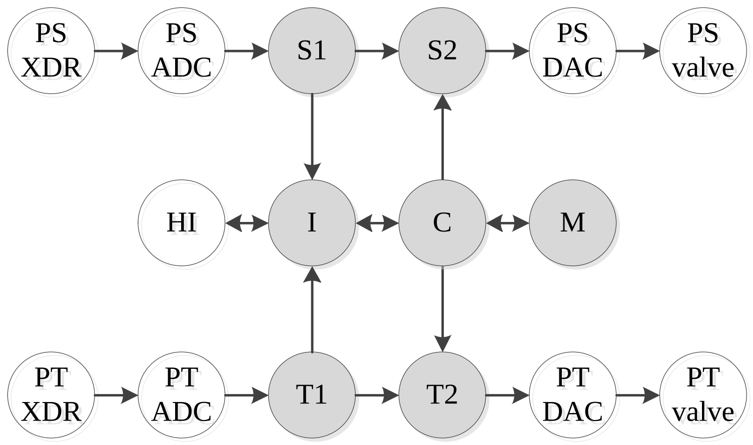


Figure 24 Air data test set architecture

We can reduce the number of possible mappings by imposing requirements and constraints as listed in Table 31.

Component	Symbol	HW	FW	SW	Reasoning
PS transducer		HW			must be hardware
PS ADC		HW			must be hardware
PS unit converter	S1		FW	SW	not hardware due to high cost
PS pressure control	S2		FW	SW	not hardware due to high cost
PS DAC		HW			must be hardware
PS valve		HW			must be hardware
Human interface	HI	HW			Keyboard, display, etc.
Interface	I		FW	SW	not hardware due to high cost
Air data calculator	C		FW	SW	not hardware due to high cost
Math co-processor	M	HW		SW	too complex for FW
PT transducer		HW			must be hardware
PT ADC					must be hardware
PT unit converter	T1		FW	SW	not hardware due to high cost
PT pressure control	T2		FW	SW	not hardware due to high cost
PT DAC		HW			must be hardware
PT valve		HW			must be hardware

Table 31. Components descriptions

For software/hardware/firmware partitioning analysis, we can first assign the components that can be only mapped to a single modality without other options.

Furthermore, we can group S1 and S2 into a single component (S1-S2) to reduce the number of components for analysis. Similarly, we can group T1 and T2 into (T1-T2), and I and C into (I-C). We redraw Figure 24 as Figure 25.

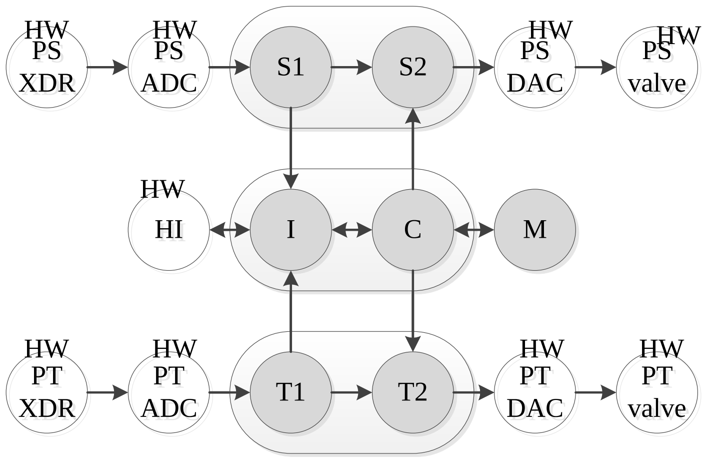


Figure 25 A simplified architecture for analysis

Based on Figure 25 and Table 32, the number of possible mappings is reduced from 3^{16}=43,046,721 to 3×4×5÷2=30 (from 16 to 4 components).

Component	Symbol	HW	FW	SW	Reasoning
PS unit converter PS pressure control	(S1-S2)		FW	SW	not hardware due to high cost
Interface Air data calculator	(I-C)		FW	SW	not hardware due to high cost
Math co-processor	M	HW		SW	too complex for FW
PT unit converter PT pressure control	(T1-T2)		FW	SW	not hardware due to high cost

Table 32. A simplified table for analysis

An air data calculator involves with complex mathematical calculations, and the timing for component (I-C) is not critical, so that {SW} is a better mapping than {FW}; as a result, the number of possible mappings is further reduced to 3×3×4÷2=18 (from 4 to 3 components). The modalities for (S1-S2) and (T1-T2) should be identical, since they are similar in functionality. Now, 18 mappings are reduced to 9 (from 3 to 2 components) as shown in Table 33, Table 34, Table 35 and Figure 26.

ADTS	(S1-S2), (T1-T2)	M	
A1	SW	*	
A2	FW	*	
A3	HW	*	No, cost is too high
A4	*	SW	
A5	*	FW	No, too complicated
A6	*	HW	

Table 33. Possible mappings for 2 components

A1	(S1-S2), (T1-T2)	M	
A1-1	SW	SW	Option 1
A1-2	SW	FW	No, too complicated
A1-3	SW	HW	Option 2

A2	(S1-S2), (T1-T2)	M	
A2-1	FW	SW	Option 3
A2-2	FW	FW	No, too complicated
A2-3	FW	HW	Option 4

Table 34. Expand nodes A1 and A2

Expanding nodes A4 and A6 does not provide any additional benefits as shown in Table 35, so that they are terminated from the tree.

A4	(S1-S2), (T1-T2)	M	
A4-1	SW	SW	Same as A1-1
A4-2	FW	SW	No, too complicated
A4-3	HW	SW	No, cost is too high

A6	(S1-S2), (T1-T2)	M	
A6-1	SW	HW	Same as A1-3
A6-2	FW	HW	No, too complicated
A6-3	HW	HW	No, cost is too high

Table 35. Expand nodes A4 and A6

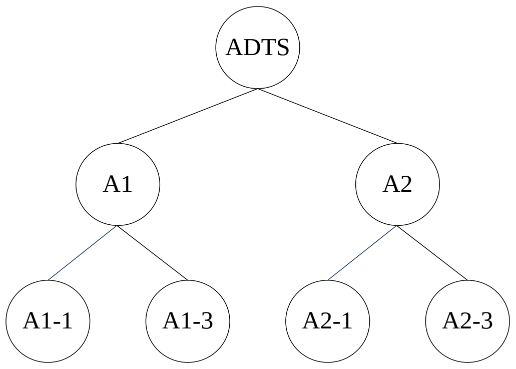


Figure 26 Four possible mappings

After reducing the possible components mappings to four as shown in Figure 26, we are able to propose reasonable options for design decision-making in Table 36, Table 37, Table 38 and Table 39.

1. Option 1 (A1-1)

A1-1	Symbol	Partition/ Implementation	Description
Interface and air data calculator	(I-C)	SW/CPU	Altitude and air -speed calculations
PS unit converter and valve control	(S1-S2)	SW/CPU	(1) Convert frequency to in-hg; (2) Pressure control
PT unit converter and valve control	(T1-T2)	SW/CPU	(1) Convert frequency to in-hg; (2) Pressure control
Mathematical coprocessor	M	SW/CPU	Mathematics library
Advantages	Low-cost; simple programming and design		
Disadvantages	Slow in valve control; slow in display		

Table 36. Option 1

2. Option 2 (A1-3)

A1-3	Symbol	Partition/ Implementation	Description
Interface and air data calculator	(I-C)	SW/CPU	Altitude and air-speed calculations
PS unit converter and valve control	(S1-S2)	SW/CPU	(1) Convert frequency to in-hg; (2) Pressure control
PT unit converter and valve control	(T1-T2)	SW/CPU	(1) Convert frequency to in-hg; (2) Pressure control
Mathematical coprocessor	M	HW/chip	Mathematics library
Advantages	Low-cost; simple programming and design; fast in display		
Disadvantages	Slow in valve control		

Table 37. Option 2

3. Option 3 (A2-1)

A2-1	Symbol	Partition/ Implementation	Description
Interface and air data calculator	(I-C)	SW/CPU	Altitude and air-speed calculations
PS unit converter and valve control	(S1-S2)	FW/FPGA	(1) Convert frequency to in-hg; (2) Pressure control
PT unit converter and valve control	(T1-T2)	FW/FPGA	(1) Convert frequency to in-hg; (2) Pressure control
Mathematical coprocessor	M	SW/CPU	Mathematics library
Advantages	Fast in valve control		
Disadvantages	Medium cost; complex programming and design; slow in display		

Table 38. Option 3

4. Option 4 (A2-3)

A2-3	Symbol	Partition/ Implementation	Description
Interface and air data calculator	(I-C)	SW/CPU	Altitude and air-speed calculations
PS unit converter and valve control	(S1-S2)	FW/FPGA	(1) Convert frequency to in-hg; (2) Pressure control
PT unit converter and valve control	(T1-T2)	FW/FPGA	(1) Convert frequency to in-hg; (2) Pressure control
Mathematical coprocessor	M	HW/chip	mathematics library
Advantages	Fast in valve control; fast in display		
Disadvantages	Medium-cost; complex programming and design		

Table 39. Option 4

E. A TOOL FOR SW/FW/HW CODESIGN

Though we did not build a tool for our software/firmware/hardware codesign methodology in this dissertation, we present a procedure for building this tool. This procedure is similar to Figure 10 in section A but with more details.

1. List and enumerate design requirements in a table.
2. List and enumerate design constraints in a table.
3. List and enumerate design options in a table.

4. Disqualify design options which cost too much or violate requirements/ constraints. Link disqualified design options to the requirements/ constraints for traceability.
5. Select the best design from the qualified options by cost estimation, and then form an architecture from this option. The architectural components are described in the selected design option; the connections among these components can be easily constructed if inputs and outputs are clearly defined in each component.
6. Define modalities in a set {modality #1, modality #2…modality #L}. The number of modalities is L=|{modality #1, modality #2… modality #L}|.
7. Simplify the analysis by excluding the components which must be assigned to certain modalities.
8. Form an N×M table. N is the number of columns; each column represents an unassigned component in the architecture. M is the number of rows; each row represents a condition with one component being assigned to a modality in the set of modalities and other components being don't cares (unassigned). There are N×L rows with N being the number of components and L being the number of modalities.
9. Disqualify the condition (row) if it costs too much or violates any requirement/constraint by tagging it with "FALSE," and then link it to the requirements/constraints for traceability. Qualify the condition if it satisfies all requirements/constraints by tagging it with "TRUE."
10. Use A* algorithm to find a feasible solution. For tie-breaker nodes, expand the node in an alphanumerical order.
11. Continue to step 12 if we have enough design details; otherwise, return to step 5.
12. If the leaf-node is not a solution or more leaf-nodes are required, return to step 10; otherwise, continue to step 13.
13. If no solutions are found, return to step 1 to modify requirements, constraints or modalities; otherwise, present solutions to decision makers.

Figure 27 shows this procedure in a diagram.

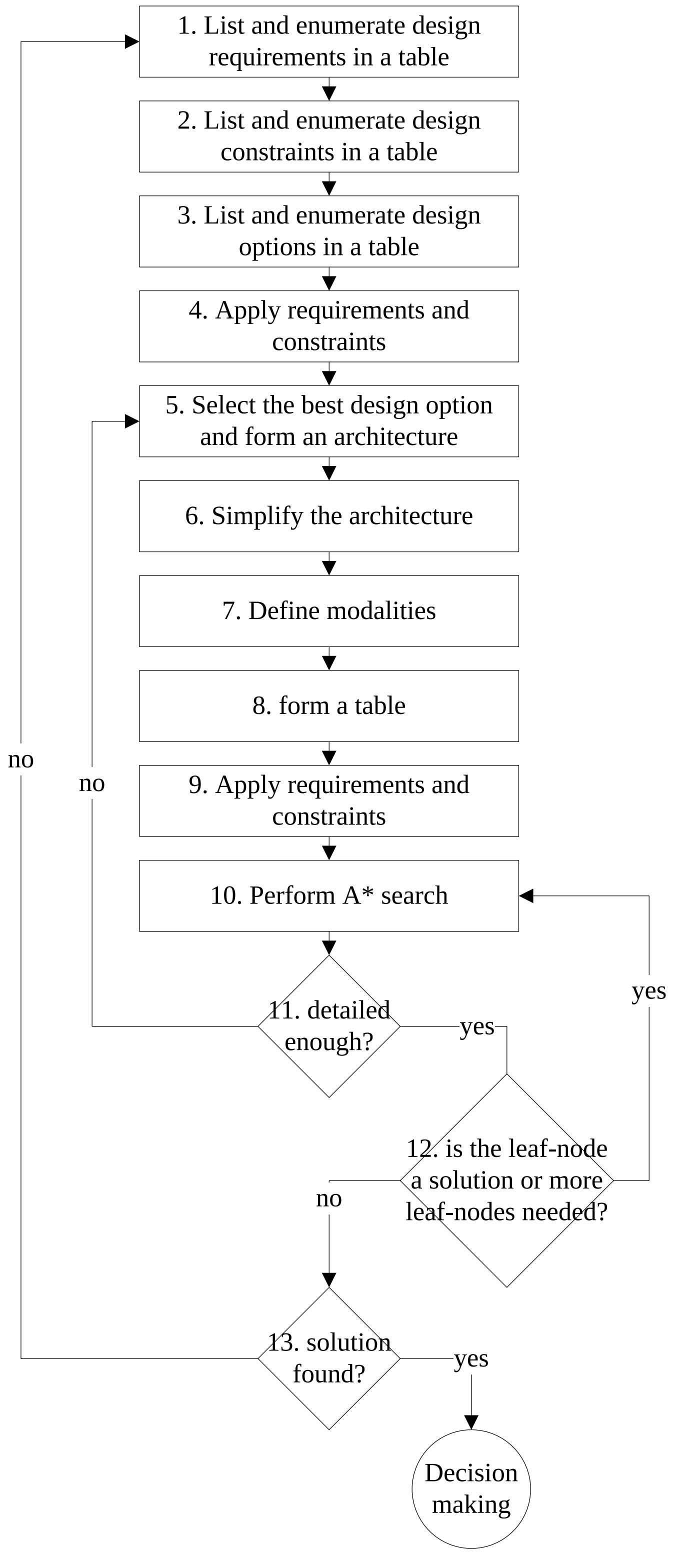

1. List and enumerate design requirements in a table
2. List and enumerate design constraints in a table
3. List and enumerate design options in a table
4. Apply requirements and constraints
5. Select the best design option and form an architecture
6. Simplify the architecture
7. Define modalities
8. form a table
9. Apply requirements and constraints
10. Perform A* search
11. detailed enough?
yes
no
12. is the leaf-node a solution or more leaf-nodes needed?
yes
no
13. solution found?
yes
no
Decision making

Figure 27 Tool design flowchart

IV. RECONFIGURABLE COMPUTING DESIGN PATTERNS

This chapter describes some example design patterns. The ones described here were used in the implementation of the filtering example for ultra-high frequency signals in Section B, Chapter III and the case study in Chapter V. They follow the standardized format and set of contents suggested by Gang of Four (GoF) [48]. These five reconfigurable computing based firmware design patterns are not in the 89 collected by André DeHon et al. [50].

A. POLYPHASE DFT FILTER BANKS

Polyphase DFT filter banks were first proposed by Vaidyanathan in 1993 [75]. They are one of the most important applications of multirate digital signal processing. A multirate system processes digital signals at different sampling rates in various parts of the system. The DFT, which stands for discrete Fourier transform, is used to convert the polyphase inputs to multiple frequency subbands. Polyphase inputs are generated by the split sequences of the input digital signal being going through polyphase filters. A subband is a specific range of frequencies in the frequency spectrum [76]. See the collaborations in section A.6 for further explanation.

We do not claim that we invented this design pattern; rather, our focus is on signal decomposition into subbands for high computational efficiency. This decomposition provides a way to process input wideband signals in different frequency bands; this makes frequency-dependent applications possible. High computational efficiency is critical for reconfigurable computing based firmware due to its limited resource.

1. Name and Classification

- Name: polyphase DFT filter banks design pattern
- Classification: digital signal processing class, filter banks subclass

2. Intent

The intent of Polyphase DFT filter banks is for multirate digital signal processing, analysis and reconstruction. Polyphase DFT filter banks can separate a wide bandwidth serial input signal at a high data sampling rate into multiple parallel subsequences so that they can be processed at a lower data sampling rate. The analysis stage converts polyphase input signals into multiple subbands for frequency-dependent applications. In the reconstruction we recombine parallel data channels into a single output for transmission.

3. Motivation

The motivation is the popular applications of subband coding to speech, audio and video and multiple-carrier data transmission [76].

4. Applicability

For single-frequency signals, an analog filter is a better choice than this design pattern.

5. Participants

There are five components in this design pattern: (1) polyphase filters, (2) an inverse discrete Fourier transform (IDFT) operator, (3) processors, (4) a discrete Fourier transform (DFT) operator, and (5) conjugated polyphase filters. These five components reside in the square box (polyphase DFT filter banks) in Figure 28; typically there is an analog-to-digital converter and a demultiplexer before, and a multiplexer and a digital-to-analog converter after these components.

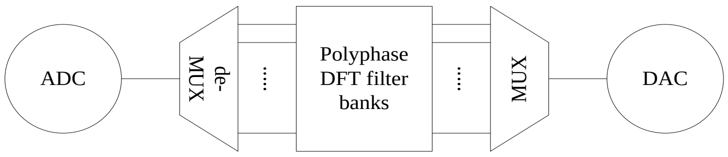


Figure 28 Typical components interfacing with polyphase DFT filter banks

6. Collaborations

Based on [76], these five components collaborate in the following way.

1. All split sequence digital signals {x[i], x[M+i], …} , for i=1 to M, M = number of channels go through polyphase filters {$H_i(z)$, for i=0 to M, and become polyphase input signals.
2. The polyphase input signals are converted to M subbands by going through an M×M inverse discrete Fourier transform operator (IDFT).
3. All M subbands are processed in parallel independently.
4. The processed M subbands go through an M×M discrete Fourier transform operator (DFT) and ready to be converted to M polyphase outputs.
5. All M output signals from the discrete Fourier transform operator are converted to M polyphase outputs {y[i], y[M+i],…}, for i=1 to M, by going through conjugated polyphase filters {$\hat{H}_i(z)$}, for i=0 to M.

Figure 29 shows these five steps.

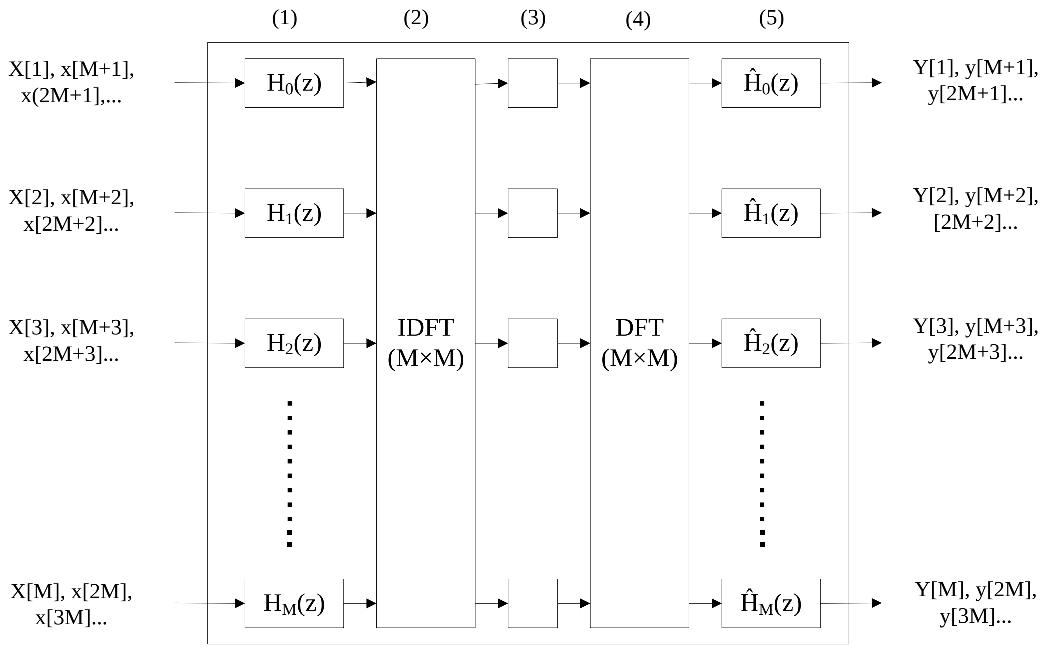


Figure 29 Polyphase DFT filter banks

The mathematical proof can be found in Appendix E.

7. Consequences (Benefits)

According to Schniter [77], the number of multiplications required for computing the DFT can be estimated as

$$Number\ of\ multiplications \approx (N + \frac{M}{2}\log_2 M)$$

where N = order of finite impulse response (FIR) filter, and M = number of polyphase components. According to [78], the propagation delay is determined by the order of polyphase finite impulse response filter (N). For linear phase (symmetrical coefficients), the propagation delay can be estimated as:

$$propagration\ delay\ \approx \frac{(N-1)}{(2\times sample\ frequency)}$$

8. Implementation

A finite impulse response filter, an inverse discrete Fourier transform operator, and a discrete Fourier transform operator consume much reconfigurable computing resources, so the number of parallel channels is limited. This design pattern can be implemented on a general-purposed computer, but our focus is on reconfigurable computing based embedded systems.

9. Algorithm

Refer to [75] for established algorithm.

10. Known Uses (Examples)

a. Example 1

The program in Table 89 (Appendix C) written in MATLAB demonstrates how to divide an input signal into 32 subbands as the analysis filter banks. In this example, we apply a set of unit sinusoids at different frequencies for the incoming data. Only 16 channels of filters in the magnitude response of the filter banks are shown in Figure 30, because fast Fourier transforms (FFTs) produce conjugate signals for real-valued inputs. Each color in Figure 30 represents a frequency subband. Signals within this subband are passed through with slight attenuation; however, signals outside this subband are greatly attenuated (blocked). Same effect is applied to other subbands.

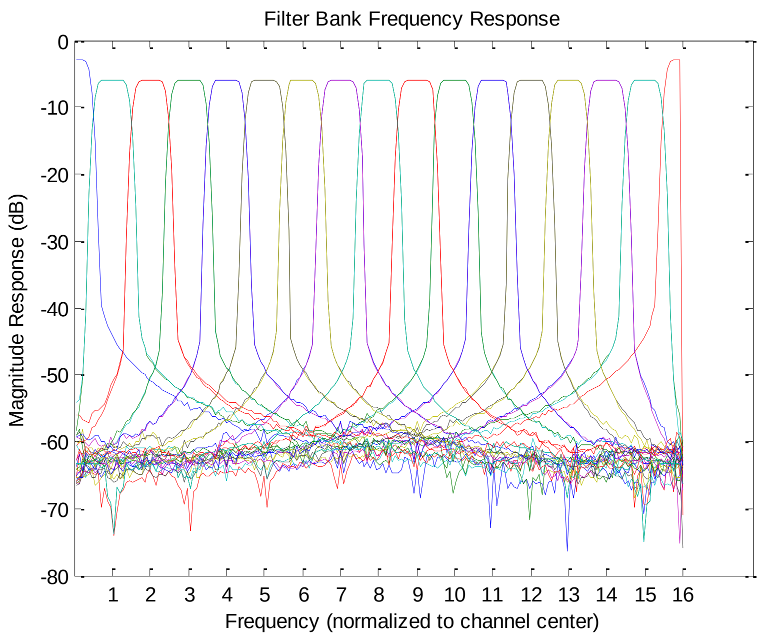


Figure 30 16 magnitude responses of a polyphase DFT filter banks for real inputs

b. Example 2

This example shows the entire operation of discrete Fourier transform filter banks which are composed of analysis and synthesis filter banks. Synthesis filter banks are used to combine multiple parallel subbands into a single output signal; its algorithm is the mirror image of analysis filter banks.

The configurations for this example are (1) the input signal is a 3-second voice recording, (2) the number of channels is 32 and the order for each polyphase FIR filter is 8, and (3) there are no changes between the polyphase inputs and outputs (a straight-through condition). We conclude that this example is successful since the input and output signals are nearly identical as shown in Figure 31. For a complete and detailed program in MATLAB, see Table 90 in Appendix C.

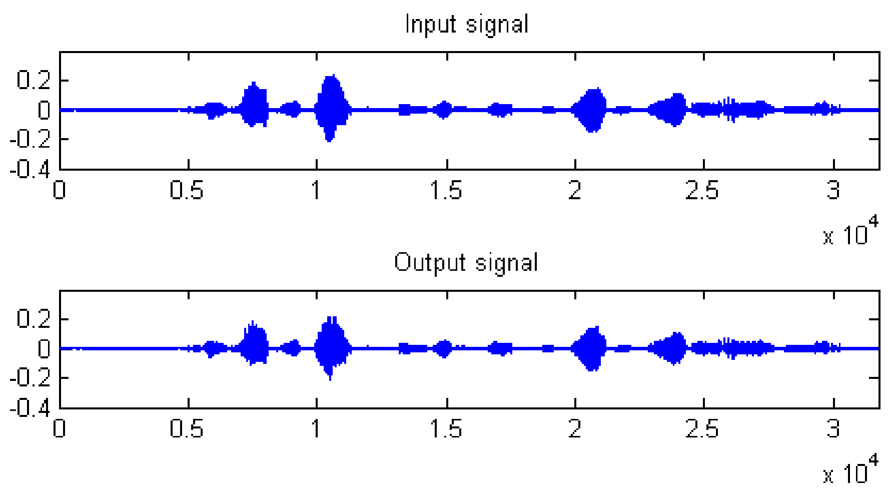


Figure 31 Signal before and after polyphase DFT filter banks

11. Related Patterns

The post-deserialization bits remapping design pattern provides inputs for the polyphase filters before the IDFT operator; the polyphase filters after the DFT operator provides inputs for the pre-serialization bits remapping design pattern.

B. DATA ALIGNMENT DESIGN PATTERN

Even though data-alignment problems have been widely addressed in telecommunication applications to produce error-free transmissions, they are unusual for reconfigurable-computing embedded systems, and only arise when dealing with ultra-fast data. In this section, we will describe a new data-alignment design pattern of ours at a high level of abstraction without bias for any type of implementation. See appendix D for background information.

1. Name and Classification

- Name: data-alignment design pattern
- Classification: ultra-fast communication class, synchronization subclass

2. Intent

When moving multiple parallel bit streams and forwarding source-synchronous sampling clocks to a device at an ultra-high data rate, we are facing two problems. One problem is caused by the shrinking of the data window. The data window is the time period when the data is stable. As shown in Figure 32, when the data rate gets faster (or data window shrinks), the sampling clock could arrive when data is in transition, or even a few bits late. Another problem is caused by different data and clock path delays; each data bit-stream arrives at the destination device at a different time. Figure 33 shows three timing cases: (1) data is sampled correctly, (2) data is sampled near transition, and (3) data is sampled at a wrong bit.

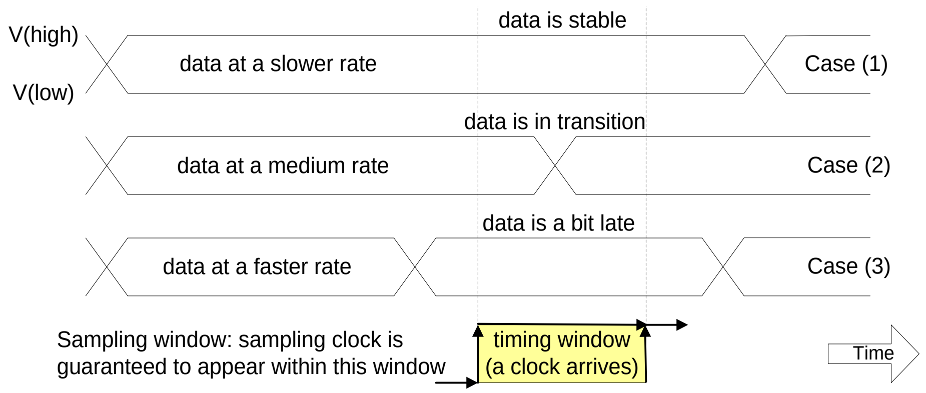


Figure 32 Shrinking of data window at higher data rate

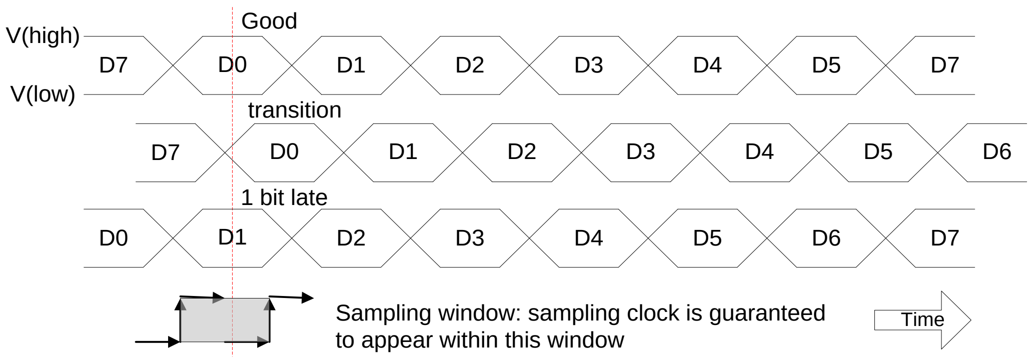


Figure 33 Three timing cases

3. Motivation

A data source sends data and clocks to a data receiver; the receiver must decide where the middle of the data bit is, and find the beginning and the end of incoming bits. This is important to sample the data correctly because the rising and falling edges of the data bits are distorted.

4. Applicability

For embedded systems, data-alignment design patterns only apply to ultra-fast (gigahertz) parallel data communication. At a lower rate, data communication typically is conducted in serial instead of parallel and data alignment is not required. In addition, at a lower rate, even the data communication is parallel; the data windows are probably wide enough to be sampled correctly by the source synchronized clocks without any calibration.

5. Participants

There are three components involved in this design pattern: a bit-alignment algorithm, a byte-alignment algorithm, and a memory device. See Figure 36 for their relationship.

6. Collaborations

These three components collaborate in the following way.

1. The bit-alignment algorithm inserts a delay to each data channel so that each bit is sampled at the middle of its data window, not at the edges.
2. The byte-alignment machine rotates the bits in a data byte until the byte matches the predefined pattern from the data source.
3. The memory device reads bit- and byte-aligned data from all input bit-streams with their local clocks, and then writes this data to a processor with a global clock for data synchronization. This is called overall alignment.

7. Consequences

The data-alignment design pattern fails to function if data windows are too narrow. The data window size at the receiver must be greater than zero to operate correctly. The shrinking of data window is caused by clock jitter, duty-cycle distortion, receiver input capacitance, power supply, and temperature, etc. [79]

8. Implementation

The bit-alignment algorithm, the byte-alignment algorithm and the memory device are typically built in the ISerDes which is in the fabric of reconfigurable computing. ISerDes stands for Input Serializer/Deserializer, which converts input data from serial to parallel format, and can be considered as a demultiplexer.

9. Algorithm

a. Bit-Alignment

Even though the precisely predetermined synchronization relationship between the data and clocks is degraded by path delays, we can align the data bits by shifting the sampling edge of the clock to the center of the data window (where data is stable) by adding delay to the data paths. A single bit-alignment procedure is described below. Figure 34 is the flowchart of this algorithm.

1. A data source sends a predefined serial data pattern to a data receiver. It initializes a timer to zero.
2. The data receiver samples the incoming serial data bits by using the synchronous sampling clock from the data source.
3. If the read data bit is stable, add delay to a timer. Being stable means that multiple consecutive reads have the same value; otherwise, data bit is unstable. Repeat steps (2) and (3) until data is unstable.
4. If the read data bit is unstable the first time, save the timer value which is the total amount of delay, T_a, from a stable state to an unstable state. Start a new timer and add delay to the new timer. Repeat steps (2) to (4).
5. If the read data bit is unstable the second time, save the timer value which is the total amount of delay, T_b, from an unstable state to another unstable state.
6. The calibration factor can be calculated as $T_a + T_b/2.7$. If this algorithm fails to complete after a certain amount of time, it will reset to step (1).

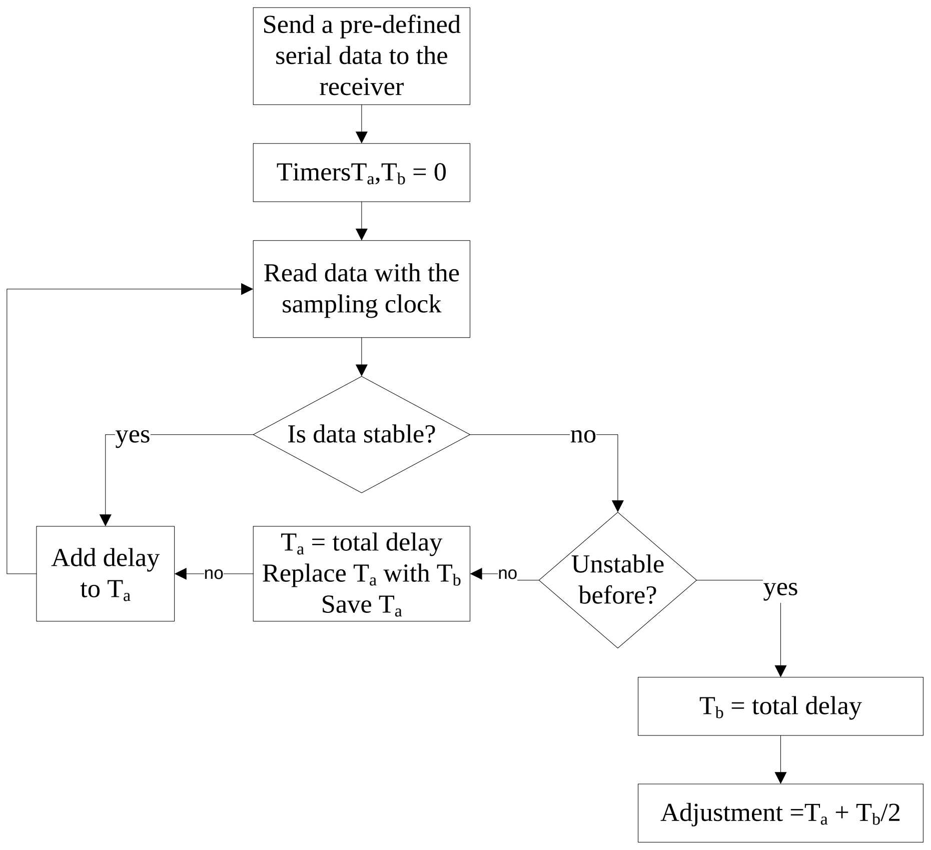

Send a pre-defined serial data to the receiver
TimersT$_a$,T$_b$ = 0
Read data with the sampling clock
Is data stable?
yes
no
Add delay to T$_a$
no
T$_a$ = total delay Replace T$_a$ with T$_b$ Save T$_a$
no
Unstable before?
yes
T$_b$ = total delay
Adjustment =T$_a$ + T$_b$/2

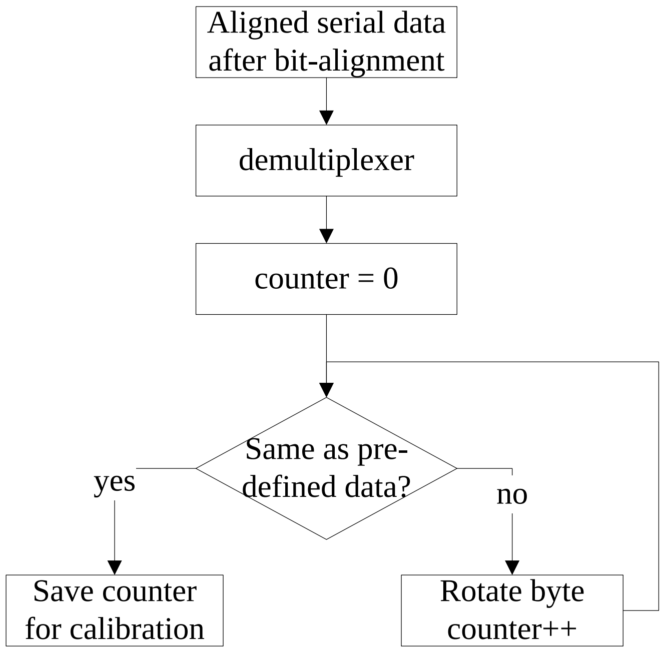
Figure 34 Bit-alignment flowchart

b. *Byte-Alignment*

If the data rate increases, the sampled data (after bit-alignment) will be possibly one or few bits late. This error can be removed if we know exactly how many bits late. The algorithm below shows the mechanism of calculating the number of bits being late for a single data bit-stream. Figure 35 is the flowchart of this algorithm.

1. A data source sends a predefined data pattern to a data receiver.
2. The data receiver aligns bit-stream according to the bit-alignment algorithm in the prior section.
3. Initialize a counter to zero.
4. The aligned bit-stream is demultiplexed into a byte (in parallel form).
5. Compare the demultiplexed byte with the predefined data byte. If they are the same, save the counter as the calibration factor; otherwise, increment the counter by one, rotate the byte by one bit left, and repeat step (5).

6. If this algorithm fails to complete after a certain amount of time, it will reset to step (1).



Aligned serial data after bit-alignment
demultiplexer
counter = 0
Same as pre-defined data?
yes
no
Save counter for calibration
Rotate byte counter++

Figure 35 Byte-alignment flowchart

c. *Overall Alignment*

Once all input bit-streams are aligned according to the bit-alignment and byte-alignment algorithms, we will perform an overall alignment below. Figure 36 is the flowchart of this algorithm.

1. The data source sends a predefined data pattern to the data receiver.
2. Bit alignment and byte alignment are performed on each bit stream.
3. When all bit streams are aligned, the data receiver sends a signal to the data source indicating that all bit-streams are aligned.
4. The data source sends a new predefined data pattern to the data receiver.
5. The data receiver detects the data pattern change.
6. The data receiver writes this data into the WRITE side of a memory device with individual local clocks.
7. The READ side of the memory device is accessed with a global clock.

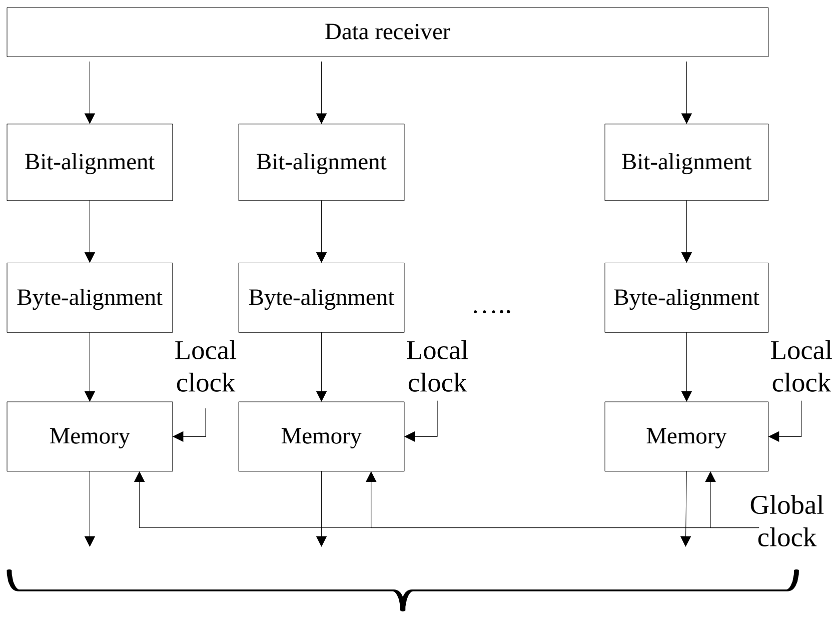


Figure 36 Overall-alignment flowchart

10. Known Uses (Examples)

The ChipSync™ features in the input of Xilinx Virtex-6 devices are able to dynamically adjust the delay of the data paths in the receiver with 75 picoseconds resolution. The BIT_ALIGN_MACHINE is similar to the bit-alignment algorithm; the BITSLIP_MACHINE is similar to the byte-alignment algorithm; and first-in first-out stacks can be used for the memory device [79]. An application example is in the first case study in Chapter V.

11. Related Patterns

There are no related design patterns.

C. POST-DESERIALIZATION BITS REMAPPING DESIGN PATTERN

Originated in telegraphy in the 1870s [80] and telephony in 1910 [81], multiplexing/demultiplexing has been widely used in telecommunications and computer networks. Multiplexing/demultiplexing technologies may be divided into space, frequency, time and code divisions. A typical application is that multiple low data rate signals are multiplexed over a single high data rate link, then demultiplexed at the other end [80].

For our application, post-deserialization means to demultiplex (split) serial data from an analog-to-digital converter into multiple parallel channels. Since the serial data from the analog-to-digital converter is in sequential order, bits must be remapped for proper digital signal processing after demultiplexing. See detailed explanation in section C.2.

1. Name and Category

- Name: post-deserialization bits remapping design pattern
- Category: communication class, serial-to-parallel subclass

2. Intent

To reduce the data rate from one device to another device, oftentimes multiple levels of demultiplexing are required. The problem is that the multiple parallel data bit-streams after demultiplexing (deserializing) are no longer in proper order for data processing. For example, a stream of ordered serial data bits from a faster device is demultiplexed into four data bit-streams at level one as shown in Figure 37, and then each data bit-stream is further demultiplexed into four data bit-streams at level two as shown in Figure 38. The 16 parallel data bit-streams (1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15, 4, 8, 12, 16) after 2 levels of demultiplexing are not usable for digital signal processing. It was necessary for us to develop a new design pattern.

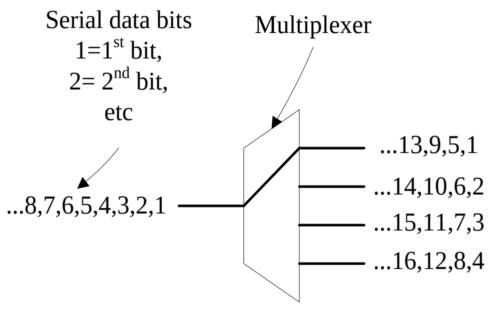


Figure 37 Level one demultiplexing

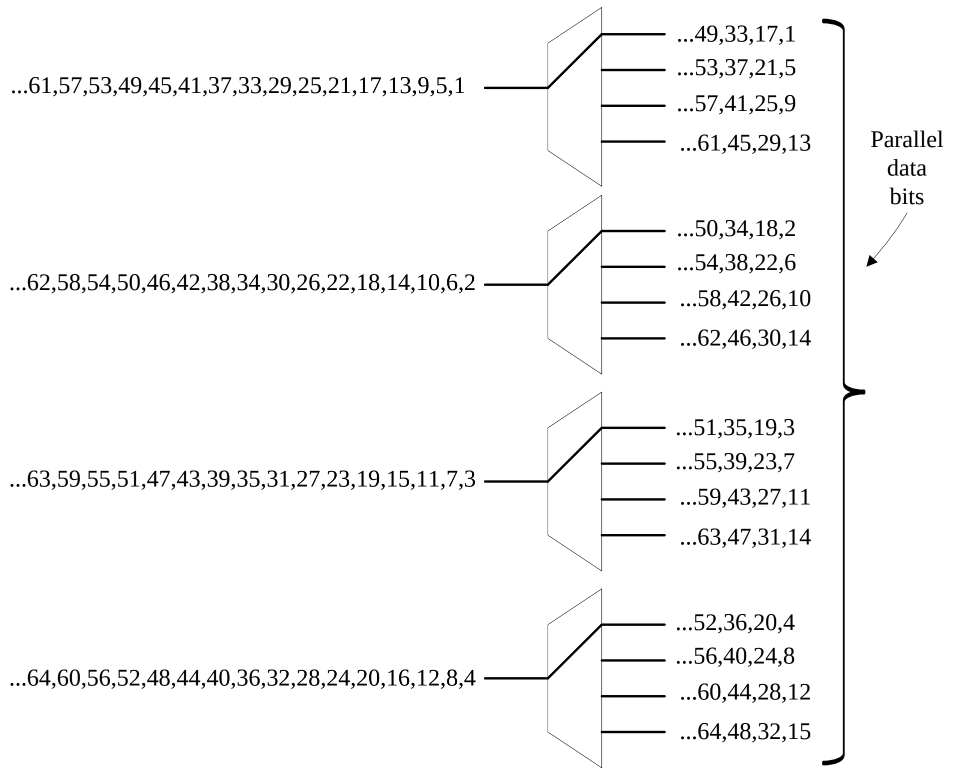


Figure 38 Level two demultiplexing

3. Motivation

Demultiplexing is used to convert serial data at a higher data rate from a faster device to parallel data at a lower data rate to a slower device.

4. Applicability

This design pattern only applies to two-level demultiplexing.

5. Participants

Simple memory addresses manipulation.

6. Collaborations

Simple memory addresses manipulation.

7. Consequences

None.

8. Implementation

This algorithm can be implemented by manipulating reconfigurable computing block RAM addresses and data byte widths with the array feature built in hardware description language.

9. Algorithm

If the overall input data width is N (N bit streams), and each bit stream is demultiplexed by an input port (such as an ISerDes) into M bit streams, then the total number of bit streams in the reconfigurable computing is N × M. Due to demultiplexing, these N × M bit-streams are not in a proper order which digital signal processing can be performed; therefore, they must be remapped. The post-deserialization bits remapping algorithm is listed in Table 40.

```
 'N_Channel = number of subbands

'N_ISerDes = 1 to N_ISerDes demultiplexer

Dim bits(1 To N_Channel * N_ISerDes) As Single

Dim bits_Post_ADC(1 To N_Channel * N_ISerDes) As Single

Private Sub Post_ADC_Remap()

For i = 1 To N_Channel Step 1

        For j = 1 To N_ISerDes

          bits_Post_ADC(i + N_Channel * (j - 1)) = bits(i * N_ISerDes - (N_ISerDes - j))

        Next j

Next i

End Sub
```

Table 40. VB6 program: post-deserialization bits remapping algorithm

10. Known Uses (Examples)

The following algorithm (written in Visual Basic 6.0) is an instantiation of the post-ADC data bits remapping algorithm in Table 40. The terms “post-ADC” and “post-deserialization" are used interchangeably in this chapter. Bits() is an array of 256 cells which contains the scattered data bits from demultiplexing, and bits_Post_ADC() is an array of 256 cells which contains the remapped data bits. The remapping algorithm is in Table 41.

```
Stage A: 128 data bit-streams coming from a data source to the reconfigurable computing

Stage B: 2: each bit-stream is demultiplexed into 2 data bit-streams. At this stage, data bits

are scattered in different memory locations in the reconfigurable computing

Stage C: put data bits in the proper order for digital signal processing

Private Sub Post_ADC_Remap()

For i = 1 To 128 Step 1

        For j = 1 To 2

                bits_Post_ADC(i + 128 * (j - 1)) =  bits(i * 2 - (2 - j))

        Next j

Next i

End Sub

Stage D: digital signal processing
```

Table 41. VB6 program: post-deserialization bits remap algorithm

11. Related Patterns

The Pre-serialization bits remapping design pattern is the counterpart of post-deserialization bits remapping design pattern.

D. PRE-SERIALIZATION BITS REMAPPING DESIGN PATTERN

References are the same as section C. For our application, pre-serialization means to multiplex (combine) parallel data from a processor into serial data for digital to analog conversion. Since the parallel data from the processor is in sequential order, bits must be remapped for proper digital to analog conversion before multiplexing (serialization). See detailed explanation in section D.2.

1. Name and Classification

- Name: pre-serialization bits remapping design pattern
- Classification: communication class, parallel-to-serial subclass

2. Intent

In Figure 39 and Figure 40, 16 parallel data bit-streams are combined (multiplexed) into a single serial output, resulting in a bit order at output (…14,10,6,2,13,9,5,1) that is not sequential.

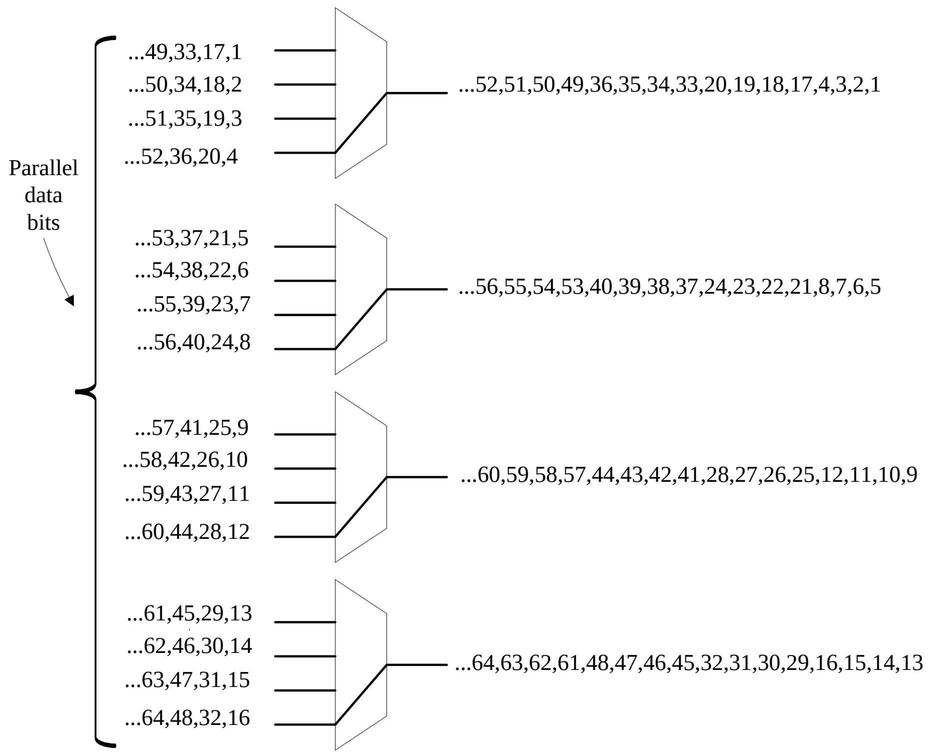


Figure 39 Level one multiplexing

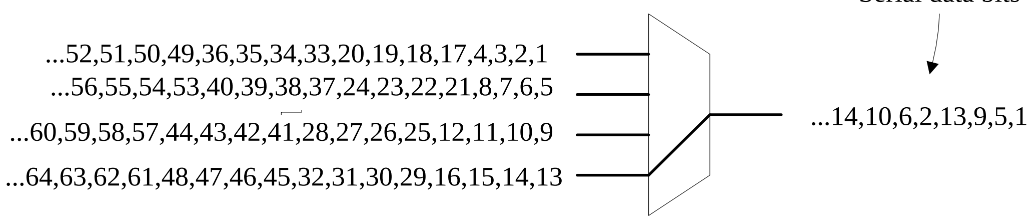


Figure 40 Level two multiplexing

3. Motivation

Multiplexing is used to convert parallel data at a lower data rate from a slower device to serial data at a higher data rate to a faster device.

4. Applicability

This design pattern only applies to 2-level multiplexing.

5. Participants

Simple memory addresses manipulation.

6. Collaboration

Simple memory addresses manipulation.

7. Consequences

None.

8. Implementation

This algorithm can be implemented by manipulating reconfigurable computing block RAM addresses and data byte widths with the array feature built in hardware description language.

9. Algorithm

Before multiplexing N parallel channels into a single output data bit stream, we must rearrange the data bits addresses for proper digital to analog conversion. The pre-serialization bits remapping algorithm is listed in Table 42.

```
For i = 1 To N_DAC_bytes
        bits2 (1 + 2 * (i - 1)) = bits1 (1 + N_DAC_RES * (i - 1))
        bits2 (2 + 2 * (i - 1)) = bits1 (2 + N_DAC_RES * (i - 1))
        bits2 ((N_DAC_bytes*2+1) + 2 * (i - 1)) = bits1 (3 + N_DAC_RES * (i - 1))
        bits2 ((N_DAC_bytes*2+2) + 2 * (i - 1)) = bits1 (4 + N_DAC_RES * (i - 1))
        bits2 ((N_DAC_bytes*4+1) + 2 * (i - 1)) = bits1 (5 + N_DAC_RES * (i - 1))
        bits2 ((N_DAC_bytes*4+2) + 2 * (i - 1)) = bits1 (6 + N_DAC_RES * (i - 1))
        bits2 ((N_DAC_bytes*6+1) + 2 * (i - 1)) = bits1 (7 + N_DAC_RES * (i - 1))
        bits2 ((N_DAC_bytes*6+2) + 2 * (i - 1)) = bits1 (8 + N_DAC_RES * (i - 1))
        bits2 ((N_DAC_bytes*8+1) + 2 * (i - 1)) = bits1 (9 + N_DAC_RES * (i - 1))
        bits2 ((N_DAC_bytes*8+2) + 2 * (i - 1)) = bits1 (10 + N_DAC_RES * (i - 1))
Next i
```

Table 42. VB6 program: pre-serialization bits remapping algorithm

10. Known Uses (Examples)

The algorithm (written in Visual Basic 6.0) in Table 43 is an instantiation of the pre-DAC data bits remapping algorithm in Table 42. The terms "pre-DAC" and "pre-serialization" are used interchangeably in this chapter. N_DAC_bytes (number of digital-to-analog converter bytes) is 32, and N_DAC_RES (digital-to-analog converter resolution) is 10-bit. Bits1() is an array of 320 cells which contain data bits after digital signal processing operations, and bits2() is an array of 320 cells which contain the remapped bits.

```
Private Sub Pre_DAC()
For i = 1 To N_DAC_bytes
        bits2 (1 + 2 * (i - 1)) = bits1 (1 + N_DAC_RES * (i - 1))
        bits2 (2 + 2 * (i - 1)) = bits1 (2 + N_DAC_RES * (i - 1))
        bits2 (65 + 2 * (i - 1)) = bits1 (3 + N_DAC_RES * (i - 1))
        bits2 (66 + 2 * (i - 1)) = bits1 (4 + N_DAC_RES * (i - 1))
        bits2 (129 + 2 * (i - 1)) = bits1 (5 + N_DAC_RES * (i - 1))
        bits2 (130 + 2 * (i - 1)) = bits1 (6 + N_DAC_RES * (i - 1))
        bits2 (193 + 2 * (i - 1)) = bits1 (7 + N_DAC_RES * (i - 1))
        bits2 (194 + 2 * (i - 1)) = bits1 (8 + N_DAC_RES * (i - 1))
        bits2 (257 + 2 * (i - 1)) = bits1 (9 + N_DAC_RES * (i - 1))
        bits2 (258 + 2 * (i - 1)) = bits1 (10 + N_DAC_RES * (i - 1))
Next i
End Sub
```

Table 43. VB6 program: pre-serialization bits remap algorithm

11. Related Patterns

The post-deserialization bits remapping design pattern is the counterpart of pre-serialization bits remapping design pattern.

E. SWITCH-AND-FILTER ARCHITECTURE

References are the same as section C. The switch-and-filter architecture is the reverse of the typical architecture used in telephony [80] with an additional analog-to-digital converter being the data pump and another additional digital-to-analog converter being the data consumer.

1. Name and Classification

- Name: switch-and-filter architecture
- Classification: ultra-fast communication class, architecture subclass

2. Intent

As technologies advance, the data sampling rates of analog-to-digital converters and digital-to-analog converters are getting faster, and the number of logic cells in an FPGA is getting higher. To accommodate these rapid changes, a generic scalable dataflow architecture is highly desirable. In addition, since the processing speed of an analog-to-digital converter (or digital-to-analog converter) is higher than that of an FPGA (the routing paths for an FPGA are programmable so are not optimized), it is necessary to have a mechanism to deserialize a single data stream at a higher data rate from an analog-to-digital converter into multiple parallel data streams at a lower data rate to an FPGA. Similarly, a mechanism to serialize multiple parallel data streams at a lower data rate from an FPGA into a single data stream to a digital-to-analog converter at a higher data rate is also required.

3. Motivation

The goal is to move ultra-fast serial data from a faster data source to a slower filter for processing by deserializing it into parallel channels, process the data, and then combine into a single serial data stream for output.

4. Applicability

This design pattern is not practical for low data rate applications, since they can transmit and receive data in serial instead of parallel.

5. Participants

There are five components involved in this design pattern, and they are a data pump, a demultiplexer, a filter, a multiplexer, and a data consumer. Figure 41 shows the relationship among these five components.

6. Collaborations

These five components collaborate in the following way.

1. A data pump produces serial data at an ultra-fast rate.
2. A demultiplexer converts serial data to parallel data at a slower rate.

3. A filter processes the parallel data.
4. A multiplexer combines all parallel data into serial data.
5. A data consumer consumes the serial data.

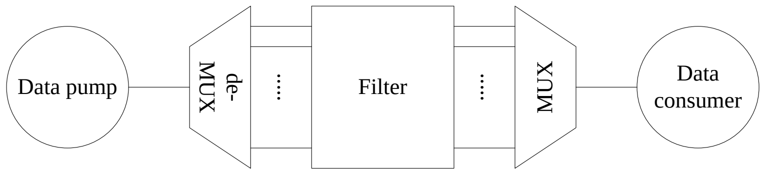


Figure 41 A switch-and-filter architecture

The data sampling rate, clock, throughput, and bit width at each stage are calculated in Table 44. In this table, we instantiate a data pump with an analog-to-digital converter (ADC), a filter with an FPGA, and a data consumer with a digital-to-analog converter (DAC).

1	A	B	C	D	E
2	ADC	#Interleaved			Defined by users
3		system clock		GHz	Defined by users
4		ADC clock	=D3/D2	GHz	System clock / #interleaved
5		#bytes	=D2		Same as #interleaved
6		sampling rate	=D4*D5	bytes/sec	ADC clock * #bytes
7		resolution_ADC	8	bits	Defined by users
8		#bits	=D5*D7	bits	#Bytes * resolution
9		throughput	=D8*D4	gigabits/sec	ADC clock * #bits
10					
11	DEMUX	#demux			defined by users
12		clock_DEMUX	=D4/D11	GHz	ADC clock / #demux
13		#bytes_DEMUX	=D5*D11	bytes	#bytes * #demux
14		sampling rate	=D13*D12	Gbytes/sec	clock * #bytes
15		#bits	=D13*D7	bits	#bytes * ADC resolution
16		throughput	=D15*D12	Gbits/sec	#bits * clock
17					
18	FPGA	#demux_FPGA			defined by users
19		#bits_DSP	=D15*D18	bits	#bits_DEMUX * #demux_FPGA
20		sampling rate_FPGA	=D12/D18	GHz	clock_DEMUX / #demux_FPGA
21		#bytes_DSP	=D19/D7	bytes	#bits_DSP / resolution
22		#bits_DAC	=D19*D30/D7	bits	#bits_DSP * resolution_DAC / resolution_ADC
23					
24	MUX	#mux			deinfed by users
25		#bits mux	=D22/D24	bits	#bits_DAC / #mux
26		clock_mux	=D20*D24	GHz	sampling rate_FPGA * #mux
27		throughput	=D25*D26	Gbits/sec	#bits_mux * clock_mux
28					
29	DAC	#mux_DAC			deinfed by users
30		resolution_DAC		bits	defined by users
31		clock_DAC	=D3	GHz	same as system clock
32		throughput	=D30*D31	Gbits/sec	resolution_DAC * clock_DAC

Table 44. Data rate, throughput, and width calculation

7. Consequences

For this architecture to function correctly, data alignment, post-deserialization bits remapping, and pre-serialization bits remapping design patterns are required as subpatterns.

8. Implementation

The most critical components in this architecture are ultra-fast analog-to-digital converter and digital-to-analog converter. The best way to implement this architecture is to use commercial-off-the-shelf (COTS) products (such as demo boards) from the ADC/DAC manufacturers, and only work with the reconfigurable computing programming.

9. Known Uses (Examples)

Figure 42 exemplifies a real system with TADC-1000 being the data pump, TDAC-2000 being the data consumer, and HAPS-62 being the filter. TADC-1000 and TDAC-2000 are from Tektronix, and HAPS-62 is a Xilinx Virtex-6 based FPGA from Synopsys. The clock, sampling rates, throughputs and bit width are calculated in Table 45.

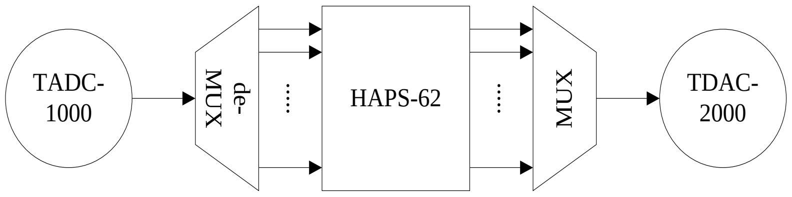


Figure 42 An instantiation of switch-and-filter architecture

ADC	#interleaved	4		defined by users
	system clock	12	GHz	defined by users
	ADC clock	3	GHz	system clock / #interleaved
	#bytes	4		same as #interleaved
	sampling rate	12	Gbytes/sec	ADC clock * #bytes
	resolution_ADC	8	bits	defined by users
	#bits	32	bits	#bytes * resolution
	throughput	96	Gbits/sec	ADC clock * #bits
DEMUX	#demux	4		defined by users
	clock_DEMUX	0.75	GHz	ADC clock / #demux
	#bytes_DEMUX	16	bytes	#bytes * #demux
	sampling rate	12	Gbytes/sec	clock * #bytes
	#bits	128	bits	#bytes * ADC resolution
	throughput	96	Gbits/sec	#bits * clock
FPGA	#demux_FPGA	2		defined by users
	#bits_DSP	256	bits	#bits_DEMUX * #demux_FPGA
	sampling rate_FPGA	0.375	GHz	clock_DEMUX / #demux_FPGA
	#bytes_DSP	32	bytes	#bits_DSP / resolution
	#bits_DAC	320	bits	#bits_DSP * resolution_DAC / resolution_ADC
MUX	#mux	8		deinfed by users
	#bits mux	40	bits	#bits_DAC / #mux
	clock_mux	3.000	GHz	sampling rate_FPGA * #mux
	throughput	120	Gbits/sec	#bits_mux * clock_mux
DAC	#mux_DAC	4		deinfed by users
	resolution_DAC	10	bits	defined by users
	clock_DAC	12	GHz	same as system clock
	throughput	120	Gbits/sec	resolution_DAC * clock_DAC

Table 45. Throughputs calculations

10. Related Patterns

The data alignment, post-deserialization bits remapping, and pre-serialization bits remapping design patterns are the basic building components for this architecture.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CASE STUDY ONE

For detailed background and test setup for this case study, refer to Appendices A and B.

A. METHODOLOGY

1. Develop Requirements and Define Constraints

The goal of this case study is to test the feasibility of digitizing radio-frequency signals up to 6 gigahertz by programming an FPGA-based embedded system. Some additional requirements are listed below.

1. Signal simulation must be frequency dependent to increase simulation accuracy based on free-space path loss equation. Free-space path loss is proportional to the square of the distance between the transmitter and receiver, and also proportional to the square of the frequency of the radio signal.

$$FSPL = \left(\frac{4\pi df}{c}\right)^2$$

   c is the speed of light, d is distance, f is frequency

2. When the number of radiofrequency sources to be simulated (N) increases scalably, the hardware complexity (cable connections) must not grow unscalably (meaning N^i, $i \geq 2$). In other words, the number of connections among N signal sources is *O(N*N)=N(N-1)÷2* which is not acceptable.
3. The system must not have power division problem. If the power loss in a transmitter is proportional to the number of receivers to be simulated, it has power division problem. This limits the number of signal sources which can be simulated. These transmitters and receivers are real physical devices, not simulated.
4. Signal simulation must be in real-time.
5. The project duration is one-year due to funding availability.
6. The instantaneous bandwidth must be in the gigahertz range instead of the conventional megahertz for our research.
7. If there are new technical challenges, it is preferable to overcome these challenges in the software domain at research and development phase to avoid expensive hardware redesign.

We group these requirements into four categories to reduce the number of conditions we have to analyze.

- Category A: items (1), (2), (3), and (7) lead to the use of a digitizer
- Category B: items (4) and (7) lead to the use of an FPGA-based embedded system (explained in section B, Chapter I)
- Category C: item (5) leads to the use of a commercial-off-the-shelf platform
- Category D: item (6) leads to new innovations

2. Form an Architecture

We form an architecture in Figure 43 to satisfy the requirements/constraints. For detailed derivation of this architecture, see filtering example in Section B, Chapter III.

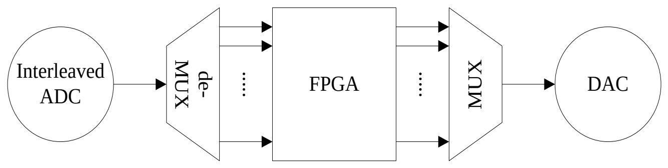


Figure 43 Architecture for ultra-wide instantaneous bandwidth signal processing

3. Build a Tree to Map Functions to Modalities

Applying the A* search with embedded ANDs methodology, we mapped functional components {ADC, de-MUX, FPGA, MUX, DAC} to modalities {HW, HW, FW, HW, HW}. For detailed mapping process, see filtering example in section B, Chapter III.

We synthesized five design patterns to meet our requirements and constraints. For detailed description of these design patterns, see the filtering example in Section B, Chapter III and Chapter IV.

When implementing the design, the discrete Fourier transform filter banks were not used. The first reason was that our primary goal for this project was to prove or disprove the concept that ultra-high instantaneous bandwidth signals can be digitized at 6 gigahertz with acceptable performance, so a simple pass-through finite impulse response filter (FIR) is sufficient to serve the purpose. The second reason was that programming discrete Fourier transform filter banks in an FPGA by using the Verilog language is not trivial; we would like to use high-level model building tools to design discrete Fourier transform filter banks in the future research. The third reason was that discrete Fourier transform filter banks might not fit into the FPGA (Xilinx Virtex-6) due to their complexity.

The pass-through finite impulse response filter has 18 taps (coefficients), h[k] where k=0, 1, 2, 3… 17. The convolution of h[k] and an input signal x[n] where n=0 to 31, representing 32 parallel input data channels, is shown in the equation below:

$$y[n] = h[n] * x[n] = x[n] * h[n] = \sum_{k=0}^{17} h[k]x[n-k]$$

If we set h[0]=1, and the rest of coefficients to zeroes, then y[n] = x[n], a pass-through condition. This algorithm can be implemented by programming multipliers and adders inside an FPGA.

4. Implementation

We designed various requirements models (theoretical design patterns) for ultra-high frequency signal filtering design. However, the implementation was accomplished by Tektronix Component Solutions at Beaverton, Oregon. This is due to the fact that the demo prototype was under development in 2012, and we did not have access to the system until late 2012.

B. FPGA SOFTWARE TEST METHODOLOGY

FPGA test methodology is iterative as shown in Figure 44. A brief description for each process is in Table 46 [42, 47,82].

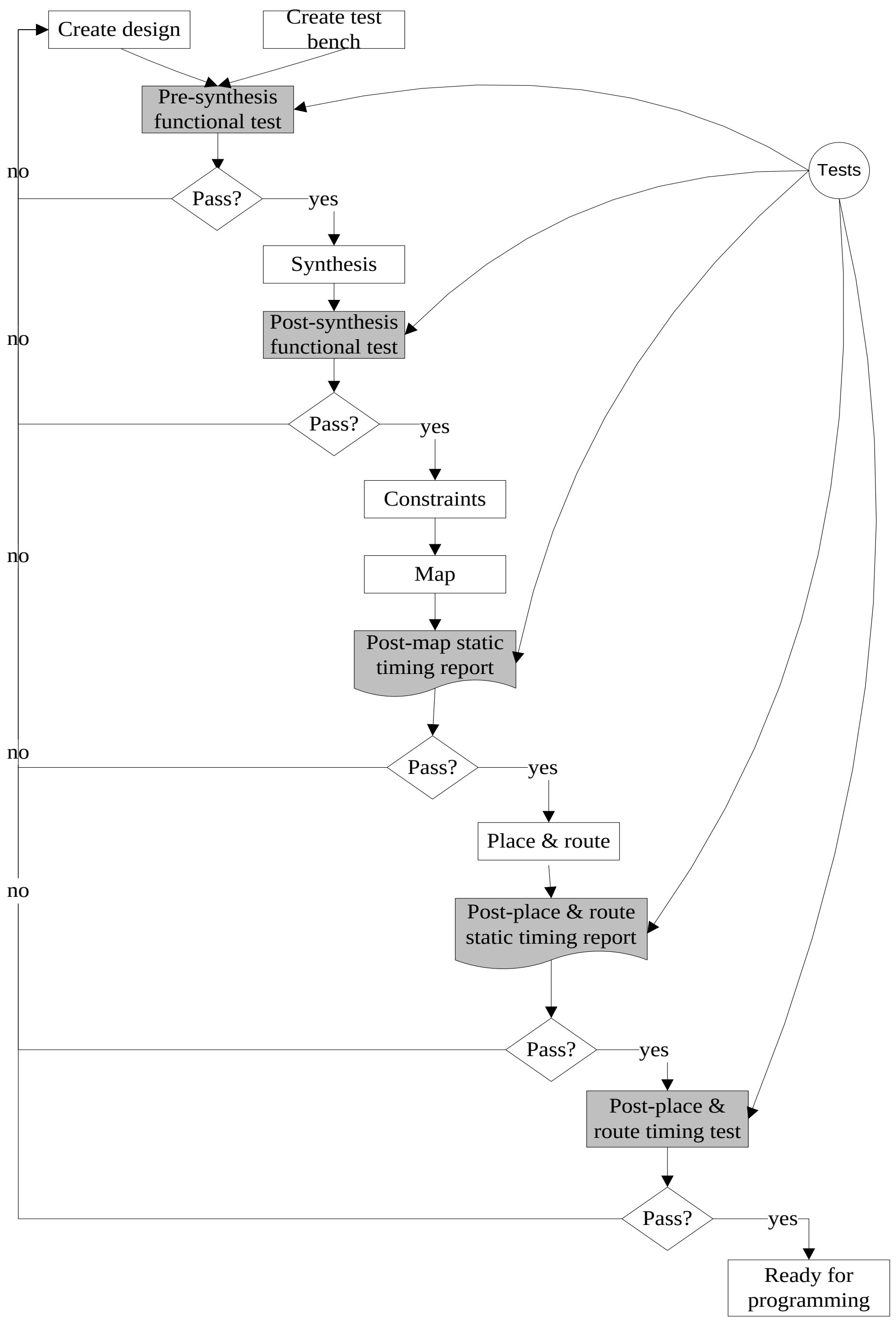

Create design
Create test bench
Pre-synthesis functional test
Tests
no
Pass?
yes
Synthesis
Post-synthesis functional test
no
Pass?
yes
Constraints
no
Map
Post-map static timing report
no
Pass?
yes
Place & route
no
Post-place & route static timing report
Pass?
yes
Post-place & route timing test
Pass?
yes
Ready for programming

Figure 44 FPGA test methodology

Process/test		Description
Create design	Process	Create designs by writing code in hardware description language, and apply reuse code as much as possible. Reuse code includes intellectual properties and modules, etc.
Create test bench	Process	A test bench, written in hardware description language code, provides a set of stimuli to create function and timing simulations.
Pre-synthesis functional test	Test	Verify the design is correct without considering timing and layout constraints. After the desired functionality is achieved, use the output data to create a self-checking test bench.
Synthesis	Process	Transform hardware description language sources into an architecture-specific design netlist (connectivity of an electronic design)
Post-synthesis functional test	Test	Differences between synthesis interpretation of language in different simulators
Constraints	Process	Timing, I/O (Input/output) pins and layout constraints
Mapping	Process	Fits the design into the available resources (such as CLBs and IOBs) on the target device. CLB: Configurable Logic Block, IOB: Input/output Block.
Post-map static timing report	Test	Determine timing violations against timing constraints by estimated logical block delays and routing delays
Place and route	Process	Places and routes the design according to device utilization and timing constraints
Post-place and route timing report	Test	Determine timing violations against timing constraints by real logical block delays and routing delays
Post-place and route timing test	Test	It allows you to check that the implemented design meets all functional and timing requirements and behaves as you expect in the device.
Programming	Process	Download design and configure FPGAs

Table 46. FPGA process definitions

C. TEST RESULTS

Considering the ADC/FPGA/DAC system as a black box, the function of this black box is to pass through a radiofrequency signal without any alteration. Under this condition, the input and output signals should be almost identical with some minor degradation caused by signal digitization and reconstruction. Based on equivalence partition and boundary conditions, we choose test signals at 500 MHz, 1 GHz, 3 GHz and 6 GHz. Three test categories and their subtests are listed below.

Functional tests (pass-through tests) include the following subtests.

- Data in the FPGA
- Comparison between an ADC and an FPGA
- Test without proper alignment software
- Test with proper alignment software

Performance tests include the following subtests.

- Power flatness
- Linearity
- Noise floor
- Sensitivity

Application tests include the following subtests. The system is tested with a Joint Electronic Warfare Effects Laboratory (JEWEL) jamming device for certain real applications.

- At bandwidths of 6 and 1.8 gigahertz
- At bandwidths of 1 gigahertz and 200 megahertz
- At bandwidth of 1 megahertz

1. Setup

a. *ADC/FPGA/DAC Specifications*

Digitizer specifications:

- Bandwidth 300KHz to 8.0 GHz (-3dB)
- Channels 1 at 12.5 GS/s, or 2 at 6.25 GS/s each
- Physical bits 8

- Effective bits 6.7-7.0 to 2 GHz, 6.2-6.9 from 2-5 GHz
- SFDR >47 dB to 5 GHz
- Input: +/- 256 mV differential into 100 ohms
- AC coupled, common mode noise limit 350 mV p-p
- Input VSWR 1.3:1 @ 2 GHz, 1.6:1 @ 6 GHz

External clock specifications:

- Frequency range 1.6GHz to 3.2 GHz1
- Short Term Jitter <400 fs

DAC specifications [90]:

- Channels 1
- Physical bits 10
- Sample rate 12 GS/s
- SFDR >45 dB to 2 GHz
- Non-Linearity 0.2% of full scale DC DNL, 0.4% of full scale DC INL
- Analog Output 8.5 GHz 3dB bandwidth

ADC inputs

- CH1+ -256mV to +256mV
- CH1- terminated with a 50-ohm load

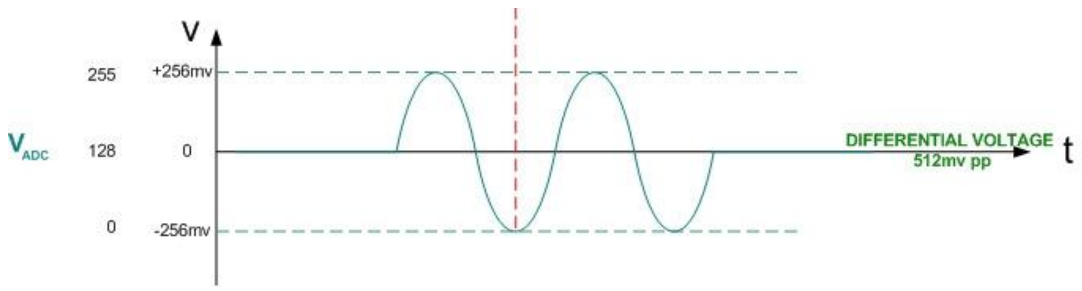


Figure 45 ADC input limits

For single-ended input, the maximum power in dBm can be calculated as below:

$$P = \frac{V_{rms}^2}{R} = \frac{\left(\frac{V_p}{\sqrt{2}}\right)^2}{R} = \frac{V_p^2}{2R} = \frac{\left(\frac{V_{pp}}{2}\right)^2}{2R} = \frac{V_{pp}^2}{8R} = \frac{0.512^2}{8\times 50ohms} = -1.84dBm$$

b. Equipment

Table 47 contains the equipment used for this case study.

	Model #	Serial #	Range	Manufacturer
Signal generator	E4438C	00686	250K~6G	Agilent
Spectrum analyzer	1164.4391.38	00080	9K-40G	Rohde Schwarz
Termination	PE6071	N/A	50-ohm	Pasternack

Table 47. Equipment models and serial numbers

2. Test Specifications

a. Alignment Tests

The digital data in the FPGA should be identical to the digital data in the analog-to-digital converter.

b. Harmonics Tests

If the input signal has no harmonics, the output signal should not have any harmonics as shown in Table 48.

Frequency	**Input**		**Output**	
	Harmonics	**Power**	**Harmonics**	**Power**
500 MHz	None	30 dB	None	-30 ± 2 dB
1 GHz	None	30 dB	None	-30 ± 3 dB
3 GHz	None	30 dB	None	-30 ± 5 dB
6 GHz	None	30 dB	None	-30 ± 8 dB

Table 48. Harmonics test specifications

c. Flatness Tests

Sweeping the frequency from 0 to 6 GHz with 10 MHz increments, the flatness deviation should be less than the specifications in Table 49.

Frequency	Input power	Output power
1 GHz	-20 dB	-20 ± 3 dB
2 GHz	-20 dB	-20 ± 4 dB
3 GHz	-20 dB	-20 ± 5 dB
4 GHz	-20 dB	-20 ± 6 dB
5 GHz	-20 dB	-20 ± 7 dB
6 GHz	-20 dB	-20 ± 8 dB

Table 49. Flatness test specifications

d. Linearity Tests

At 500 MHz, the linearity should be less than 2% for a dynamic range of 40 dB (0-40 dB).

e. Noise Floor Tests

The noise floor should be less than the specifications in Table 50.

Frequency	RBW=300 KHz	RBW=1 KHz
2.5 GHz	≤ -70 dB	≤ -60 dB
5.5 GHz	≤ -70 dB	≤ -60 dB
6 GHz	≤ -70 dB	≤ -60 dB

Table 50. Noise floor test specifications

f. Sensitivity Tests

The sensitivity should be less than the specifications in Table 51.

Frequency	RBW=300 KHz	RBW=1 KHz
2.5 GHz	≤ -65 dB	≤ -65 dB
5.5 GHz	≤ -65 dB	≤ -65 dB
6 GHz	≤ -65 dB	≤ -65 dB

Table 51. Sensitivity test specifications

g. *Test with JEWEL RF Jamming Device*

The simulated signal should be able to work with JEWEL RF jamming device from 0 to 2 gigahertz.

3. Functional Tests

a. *Data in the FPGA*

We validated our alignment algorithms by the following sequence: (1) apply a 500 MHz sinewave to a Tektronix TADC-1000 digitizer; (2) transfer the digitized data to the memory in Synopsys HAPS-62 FPGA board; (3) download the digitized data in the FPGA (HAPS-62) to a host computer through an USB interface; (4) plot the waveform by using MATLAB. The waveform (Figure 46) was identical to the input waveform; therefore, we concluded that we were able to move data from TADC-1000 to HAPS-62 FPGA successfully by using our three alignment algorithms.

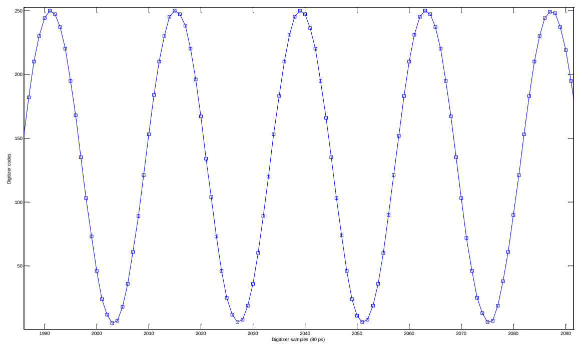


Figure 46 Digitized sinewave in the FPGA

b. *Comparison between Analog-to-digital Converter and FPGA Data*

We generated 16,384 pseudo-random patterns to check bit accuracy across the interface from TADC-1000 to the HAPS FPGA board. The data file in the analog-to-digital converter analyzer is identical to the data file in the FPGA. Table 52 only shows the first 20 LFSR patterns in the analog-to-digital converter and FPGA. LFSR stands for linear feedback shift register, an n-bit shift register which pseudo-randomly scrolls between 2^n-1 values. Once it reaches its final state, it will traverse the sequence exactly as before. Again, this table proves that we were able to move data from TADC-1000 to HAPS-62 FPGA successfully by using our three alignment algorithms.

	LFSR pattern in ADC	LFSR pattern in FPGA
File name	usbcom_data_lfsr_16k_reference.txt	usbcom_data_HAPS-62_lfsr_070912.txt
The first 20 patterns out of 16,384 pseudo-random patterns	FFFFFEFC01	FFFFFEFC01
	FFFFFEFC01	FFFFFEFC01
	FFFFFEFC01	FFFFFEFC01
	FFFFFEFC01	FFFFFEFC01
	FCFFFFFE00	FCFFFFFE00
	FCFFFFFE00	FCFFFFFE00
	FCFFFFFE00	FCFFFFFE00
	FCFFFFFE00	FCFFFFFE00
	FEFCFFFF00	FEFCFFFF00
	FEFCFFFF00	FEFCFFFF00
	FEFCFFFF00	FEFCFFFF00
	FEFCFFFF00	FEFCFFFF00
	FFFEFCFF00	FFFEFCFF00
	FFFEFCFF00	FFFEFCFF00
	FFFEFCFF00	FFFEFCFF00
	FFFEFCFF00	FFFEFCFF00
	7FFFFFFE00	7FFFFFFE00
	7FFFFFFE00	7FFFFFFE00
	7FFFFFFE00	7FFFFFFE00
	7FFFFFFE00	7FFFFFFE00

Table 52. The first 20 LFSR patterns

c. Tests without Proper Alignment Software

Without proper alignment software, the output spectrum contained numerous harmonics as shown in Figure 47, which would be unacceptable for any data processing. The picture on the left has at a unit division of 100 MHz, and the picture on the right has a unit division of 10 MHz. The input signal for this test is a one-gigahertz sinusoidal wave. Theoretically, the expected output spectrum should be a single-tone pulse (a single spike) without harmonics in frequency domain.

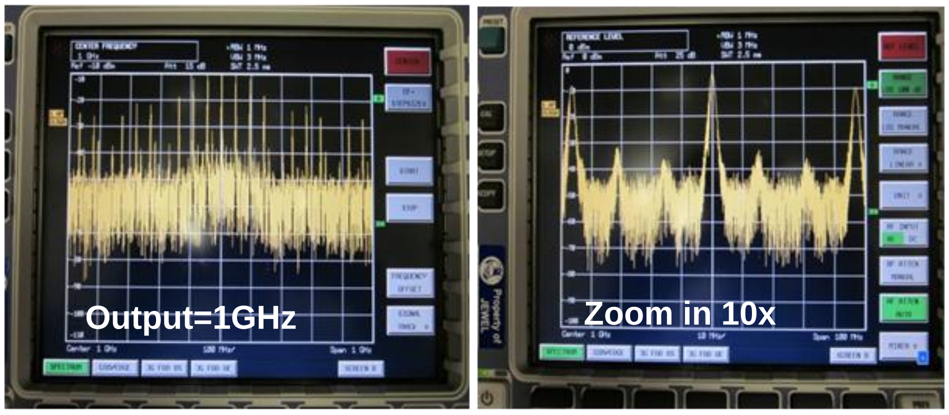


Figure 47 Tests without proper alignment software (in frequency domain)

d. *Tests with Proper Alignment Software*

With proper alignment software, input and output signals were nearly identical as shown in Figure 48 and Figure 49. We used a spectrum analyzer to measure all spectra in this section at various RBW (Resolution Bandwidth) settings. The resolution bandwidth is the smallest frequency that can be resolved, or the FFT bin size.

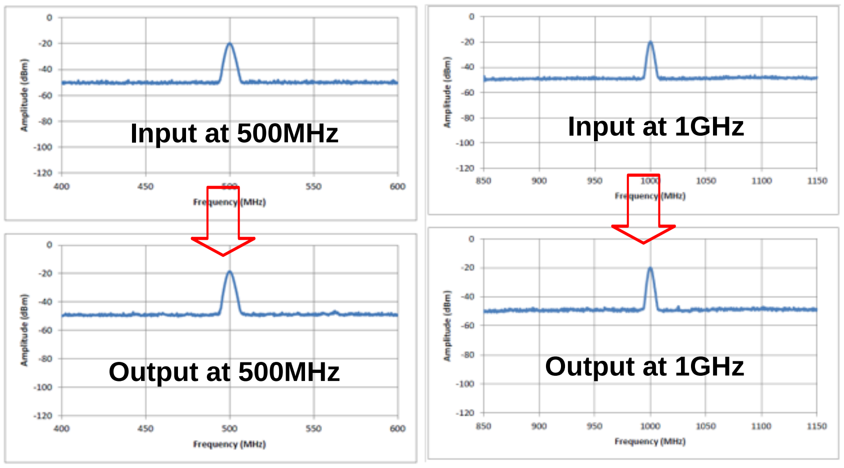


Figure 48 Signals at 500 MHz and 1 GHz; RBW=3 MHz

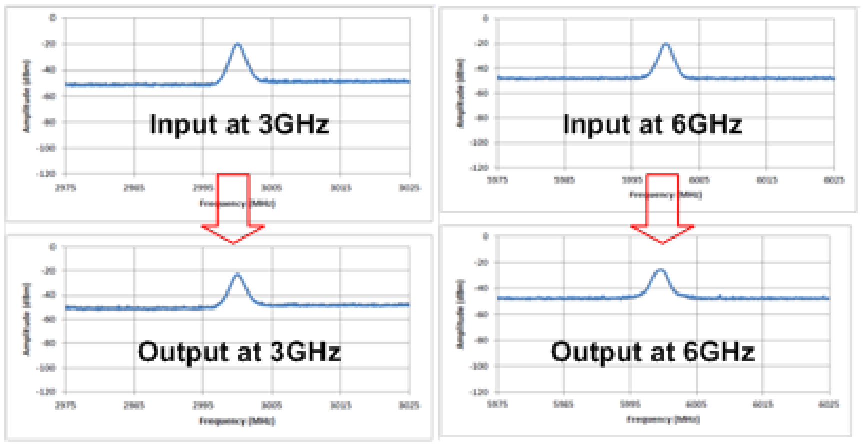


Figure 49 Signals at 3 GHz and 6 GHz, RBW=1 MHz

Test results are recorded in Table 53.

Frequency	**Input**		**Output**	
	Harmonics	**Power**	**Harmonics**	**Power**
500 MHz	None	30 dB	None	32 dB
1 GHz	None	30 dB	None	30 dB
3 GHz	None	30 dB	None	27 dB
6 GHz	None	27 dB	None	23 dB

Table 53. Harmonics test specifications

4. Performance Tests

a. Flatness Test

This test swept input signals from 0 to 6 GHz at a constant power level (-20 dBm) and a sweeping increment is 10 MHz. The output power signal dropped to -27 dB at 6 GHz as shown in Figure 50 and Table 54.

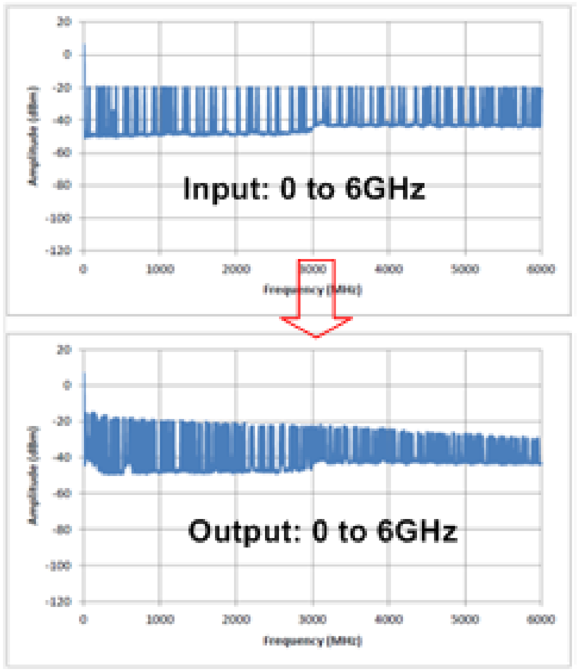


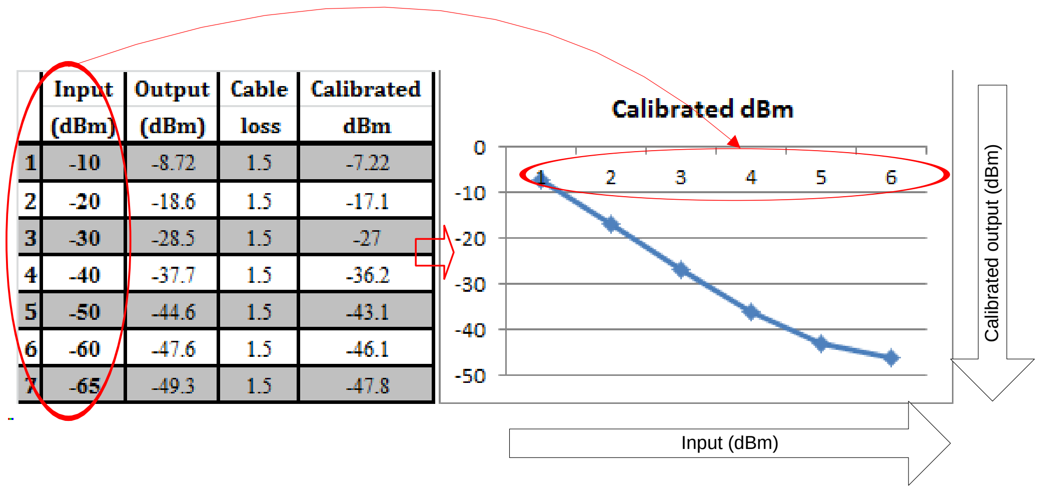
Figure 50 Sweeping, RBW=3 MHz; increment=10 MHz

Frequency	Input power	Output power
1 GHz	-20 dB	-20 dB
2 GHz	-20 dB	-20 dB
3 GHz	-20 dB	-22 dB
4 GHz	-20 dB	-23 dB
5 GHz	-20 dB	-25 dB
6 GHz	-20 dB	-27 dB

Table 54. Flatness test specifications

b. Linearity Test (500 MHz, RBW=3 MHz)

This test applied a 500 MHz sinusoidal wave at various power levels as listed in Table 55, and then observed the output power levels. Output power was adjusted for cable loss for this test. The power from -10 dBm to -40 dBm is relatively linear (about 0.2 percent); after -40 dBm, it becomes less linear.



	Input (dBm)	Output (dBm)	Cable loss	Calibrated dBm
1	-10	-8.72	1.5	-7.22
2	-20	-18.6	1.5	-17.1
3	-30	-28.5	1.5	-27
4	-40	-37.7	1.5	-36.2
5	-50	-44.6	1.5	-43.1
6	-60	-47.6	1.5	-46.1
7	-65	-49.3	1.5	-47.8

Table 55. Linearity test

c. *Noise Floor Test*

This test found the power of the noise floor in dBm at various frequencies. The noise floor is the measurement of the sum of all the noise sources and unwanted signals within a measurement system. Table 56 contains the results.

	RBW=300KHz	RBW=1KHz	
2.5 GHz	-78	-65	dBm
5.5 GHz	-75	-65	dBm
6 GHz	-75	-63	dBm

Table 56. Noise floor test

d. *Sensitivity Test*

This test finds the lowest signal power in dBm that a receiver can detect at various frequencies. Table 57 contains the results.

	RBW=300KHz	RBW=1KHz	
2.5 GHz	-73	-70	dBm
5.5 GHz	-73	-70	dBm
6 GHz	-73	-70	dBm

Table 57. Sensitivity test

5. Validate with an Existing JEWEL RF Jamming Device

We tested the system with an existing RF jamming device at various frequency bandwidths and amplitudes (strength): 6 GHz (-50 to 10 dBm), 1,800 MHz (-50 to 10 dBm), 1 GHz (-50 to -10 dBm), 200 MHz (-50 to -10 dBm), and 1 MHz (-50 to -40 dBm) as shown in Figure 51, Figure 52 and Figure 53. The jamming device was connected to the digitizer through a cable inside the laboratory.

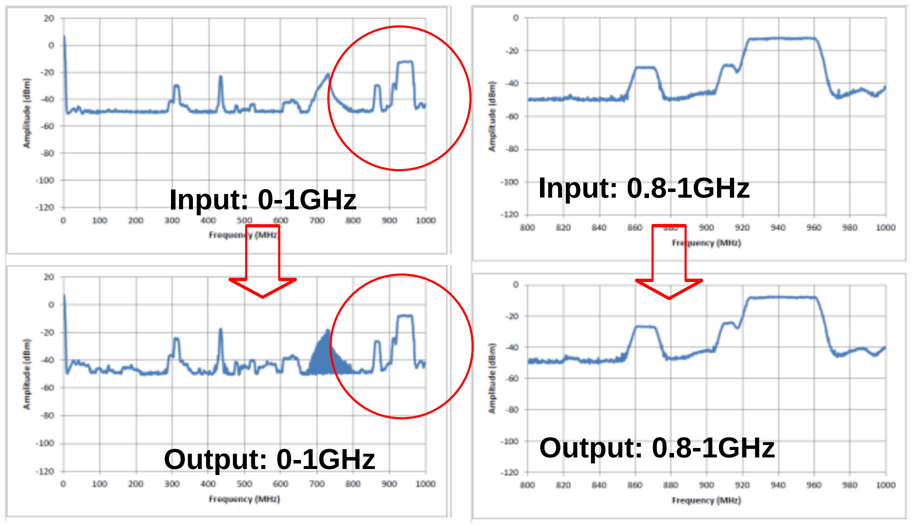


Figure 51 At bandwidths of 6 GHz and 1.8 GHz

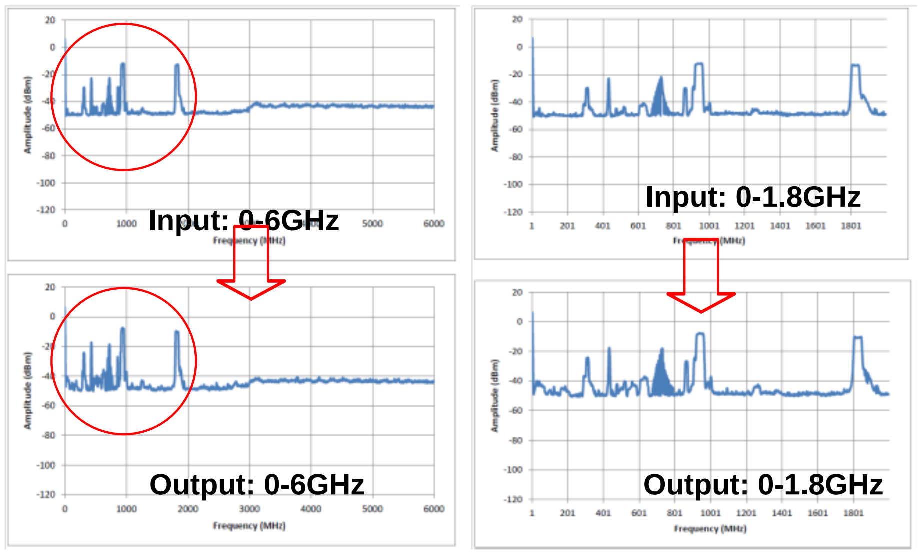


Figure 52 At bandwidths of 1 GHz and 200 MHz

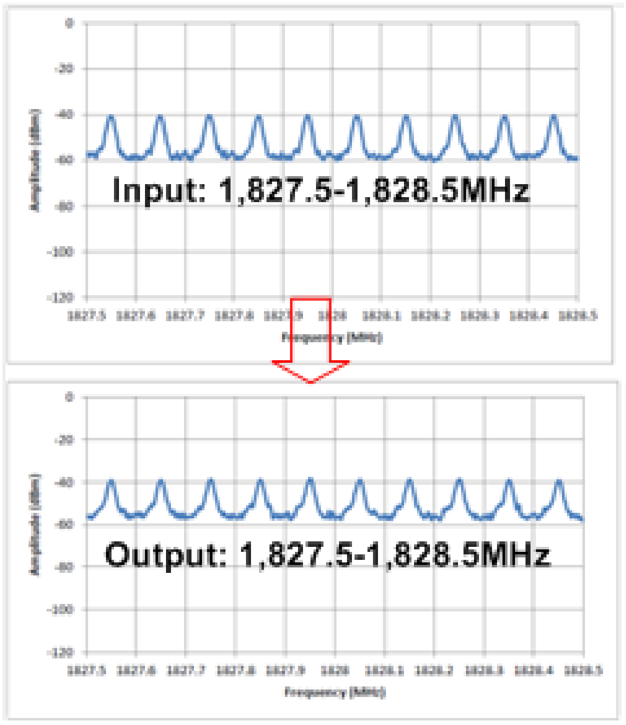


Figure 53 At bandwidth of 1 MHz

These test results are within our expectations since the signals from 0 to 2 GHz are extremely linear and stable as shown in the harmonics, linearity, sensitivity and noise floor tests.

D. TESTS CONCLUSION

Based on the test results in section C.3, it appears that our design for gigahertz signal filtering was sound.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CASE STUDY TWO

In this chapter, we use software/firmware/hardware codesign methodology to develop requirements and design for a multi-channel radar signal digital receiver which is a part of a pulse Doppler radar receiver subsystem. In the near term, the digital receiver is intended to process conventional pulse Doppler waveforms. However, it shall be capable of performing analysis on other advanced waveforms as commanded. The input carrier frequency is at 5 megahertz.

A. METHODOLOGY

1. Develop Requirements and Define Constraints

According to the requirements from our clients, the system must do the following[83]:

1. Perform analog-to-digital conversion.
2. Operate in real-time.
3. Implement range gates.
4. Digitally down-convert to baseband (including platform motion compensation).
5. Generate in-phase/quadrature (I/Q) samples.
6. Construct a pulse repetition frequency (PRF) line FIR filter and decimate.
7. Implement a fast Fourier transform (FFT) with Doppler filter banks to span Doppler bandwidth.
8. Form signal magnitude for each filter.
9. Establish a detection threshold for filter banks.
10. Declare and report target detection by filter number, and send this information to a data processor.
11. These operations shall be simultaneously performed upon three radar channels: sum, delta and guard.
12. It must be low-cost.
13. It must be well supported due to lack of experience in radar signal processing.
14. It must have high reusability to reduce development time.

We group these requirements into five categories to reduce the number of conditions we have to analyze.

- Category A: Items (1) and (11) are related to analog-to-digital converters.
- Category B: Items (2) to (10) are related to radar signal processing.
- Category C: Item (12) is related to funding availability.
- Category D: Item (13) is related to vendor's technical support ability.
- Category E: Item (14) is related to vendor's technology reusability.

Requirements (1) and (11) suggest the use of digitizers, and requirements (2)–(10) suggest the use of FPGAs. Besides the requirements and constraints from our clients, an additional constraint for our design is listed below:

- Use products of a single vendor: Each intellectual property is designed and tested for a particular piece of hardware manufactured by a particular vendor; therefore, we cannot mix intellectual properties from different vendors.

2. Form an Architecture

The data bits from an analog-to-digital converter are in serial (single-channel) at the rate of 56 megabits-per-second which can be easily processed by a filter without using parallelism (multiple parallel channels); therefore demultiplexers and multiplexers are not required. To align input serial data bits, inside the filter, there is a built-in mechanism searching for the start and stop bits for each data byte for proper data synchronization; this is accomplished automatically by most hardware chips with serial communication capability.

Even though we have sum, delta and guard channels, each channel has its own data source and can be processed independently without considering different propagation delays among them. Table 58 lists the justification for the selection of functional components and Figure 54 shows the architecture. For detailed derivation of this architecture, see filtering example in Chapter III.

Components	Requirements references	Comment
ADC	(1), (11)	For sum channel in (11)
ADC	(1), (11)	For delta channel in (11)
ADC	(1), (11)	For guard channel in (11)
Filter	(2), (3), (4), (5), (6), (8), (9)	Perform basic radar signal processing
Processor	(7), (10)	Receive detected target signals from the filter; perform discrete fast Fourier transform if required

Table 58. Architectural components

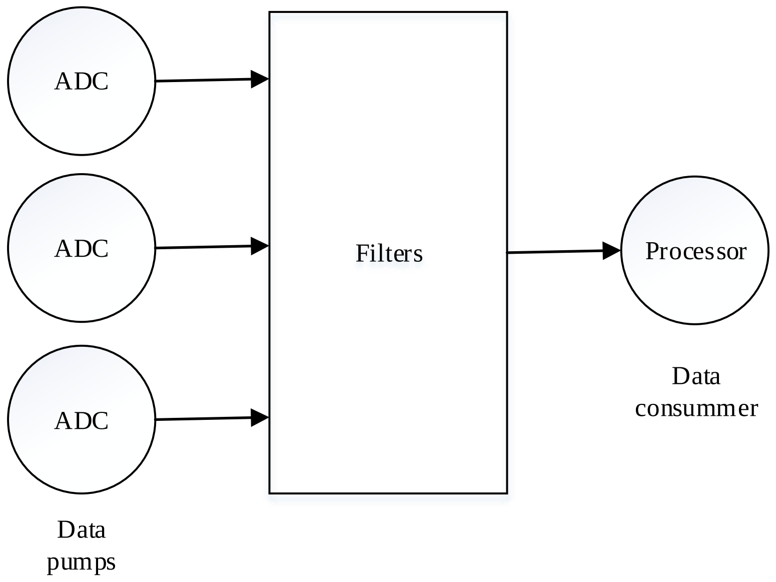


Figure 54 An architecture

3. Build a Tree to Map Functions to Modalities

Table 59 shows the first level of mapping. Table 60 and Table 61show the second level of mapping. Here, "ADC" stands for analog-to-digital converter; "F" stands for filter and "P" stands for processor.

A	ADC	ADC	ADC	F	P	Possible mapping
A1	HW	HW	HW	SW	*	Software does not work for multiple Doppler signals. For hundreds (thousands) of Doppler signals, parallelism must be used. This parallelism is not for the overall architecture, but for the processing inside the filter (F).
A2	HW	HW	HW	FW	*	OK
A3	HW	HW	HW	HW	*	The cost is too high and the design is not flexible.
A4	HW	HW	HW	*	SW	OK
A5	HW	HW	HW	*	FW	The cost is medium and the design is too complex.
A6	HW	HW	HW	*	HW	The cost is too high and the design is not flexible.

Table 59. Node A

A2	ADC	ADC	ADC	F	P	Possible mapping
A2-1	HW	HW	HW	FW	SW	OK
A2-2	HW	HW	HW	FW	FW	The cost is medium and the design is too complex.
A2-3	HW	HW	HW	FW	HW	The cost is too high and the design is not flexible.

Table 60. Node A2

A4	ADC	ADC	ADC	F	P	Possible mapping
A4-1	HW	HW	HW	SW	SW	Software does not work for multiple Doppler signals. For hundreds (thousands) of Doppler signals, parallelism must be used. This parallelism is not for the overall architecture, but for the processing inside the filter (F).
A4-2	HW	HW	HW	FW	SW	OK
A4-3	HW	HW	HW	HW	SW	The cost is too high and the design is not flexible.

Table 61. Node A4

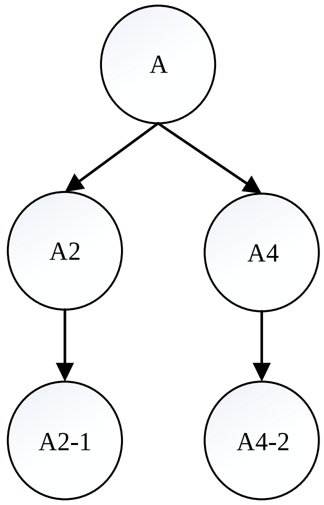


Figure 55 Node A2 (A4) is the only solution

The functional components {ADC, ADC, ADC, filter, computer} are mapped to the modalities {HW, HW, HW, FW, SW}. Before assigning design patterns (reusable intellectual properties for this case) to the functional components, we first must select a qualified vendor. Table 62 lists four candidates manufacturing FPGA systems in terms of cost, technical support and technology reuse. Technology reuse is defined as the products (such as intellectual properties and example codes) from a vendor that can be reused for our design. Vendors A, B and C are pseudo names; but Pentek is the actual vendor selected for our second case study.

	Vendor-A	Vendor-B	Pentek	Vendor-C
Model	Model A	Model B	78661/4995A/4953	Model C
Cost	$17,500	$249,726	$21,925	$24,303
Technical Support	$600 per year	Annual renewal fee of $45,000	Free for one project	Fee based
Technology Reuse	Radar, software radio, electronic warfare (limited IPs)	High performance DSP systems (limited IPs)	Conventional Radar signal processing (basic radar function IPs)	Cellular phone applications (limited IPs)

Table 62. Vendors analysis

We disqualify Vendor-B and Vendor-C for the following reasons.

- Vendor-B: the product cost is too high; the annual software license renewal fee is $45K.
- Vendor-C: the focused market is cellular phone applications; therefore, it has low technology reusability for radar signal processing.

The costs for implementation and technical support are about the same for Pentek and Vendor-A, so the deciding factor is the technology reusability. After studying their published literatures and conversing with their engineers through telephone calls, we concluded that Pentek was the optimal vendor for our project since Pentek had more intellectual properties (reusable assets) than Vendor-A for radar signal processing.

4. Reusable Assets

We use the same methodology to map our requirements to Pentek intellectual properties. Pentek provides many built-in IPs which are frequently used for radar signal processing. Analog-to-digital Acquisition IP modules capture and move data into memories. Digital down converter (DDC) IP cores decimate input samples and output In-phase/quadrature (I&Q) values. Beamformer IP core has a power meter that continuously measures the individual average power output, and threshold detector to automatically send an interrupt to the processor if the average power level of any digital down converter core falls below or exceeds a programmable threshold [84].

We will build an OR tree with embedded ANDs (RG • DC • IQ • DEC • FFT • FMG • THD) to find the optimal reusable assets for our requirements. Here, RG stands for "range gate"; DC stands for "digital down convert"; IQ stands for "I/Q samples"; DEC stands for "decimation"; FFT stands for "fast Fourier transform "; FMG stands for "form filter signal magnitude "; THD stands for "threshold detection." These components represent the requirements.

There are 3 modalities (IP cores) {AM, DC, BF}; AM stands for "A/D acquisition and memory control"; DC stands for "DDC IP core "; BF stands for "Beamforming IP core. Table 63 shows 21 possible mappings. Table 64 shows possible 18 possible mappings with RG=AM; Table 65 shows 15 possible mappings with RG=AM and DC=DC; Table 66 shows 12 possible mappings with RG=AM, DC=DC and I/Q=DC;

Table 67 shows 9 possible mappings with RG=AM, DC=DC, I/Q=DC and DEC=DC; Table 68 shows 6 possible mappings with RG=AM, DC=DC, I/Q=DC, DEC=DC and FMG=BF.

A2-1	RG	DC	I/Q	DEC	FFT	FMG	THD	Possible matching
B1	AM	*	*	*	*	*	*	yes
B2	DC	*	*	*	*	*	*	no
B3	BF	*	*	*	*	*	*	no
B4	*	AM	*	*	*	*	*	no
B5	*	DC	*	*	*	*	*	yes
B6	*	BF	*	*	*	*	*	no
B7	*	*	AM	*	*	*	*	no
B8	*	*	DC	*	*	*	*	yes
B9	*	*	BF	*	*	*	*	no
B10	*	*	*	AM	*	*	*	no
B11	*	*	*	DC	*	*	*	yes
B12	*	*	*	BF	*	*	*	no
B13	*	*	*	*	AM	*	*	no
B14	*	*	*	*	DC	*	*	no
B15	*	*	*	*	BF	*	*	no
B16	*	*	*	*	*	AM	*	no
B17	*	*	*	*	*	DC	*	no
B18	*	*	*	*	*	BF	*	yes
B19	*	*	*	*	*	*	AM	no
B20	*	*	*	*	*	*	DC	no
B21	*	*	*	*	*	*	BF	yes

Table 63. Expanding node A2-1

B1	RG	DC	I/Q	DEC	FFT	FMG	THD	Possible matching
B1-1	AM	AM	*	*	*	*	*	no
B1-2	AM	DC	*	*	*	*	*	yes
B1-3	AM	BF	*	*	*	*	*	no
B1-4	AM	*	AM	*	*	*	*	no
B1-5	AM	*	DC	*	*	*	*	yes
B1-6	AM	*	BF	*	*	*	*	no
B1-7	AM	*	*	AM	*	*	*	no
B1-8	AM	*	*	DC	*	*	*	yes
B1-9	AM	*	*	BF	*	*	*	no
B1-10	AM	*	*	*	AM	*	*	no
B1-11	AM	*	*	*	DC	*	*	no
B1-12	AM	*	*	*	BF	*	*	no
B1-13	AM	*	*	*	*	AM	*	no
B1-14	AM	*	*	*	*	DC	*	no
B1-15	AM	*	*	*	*	BF	*	yes
B1-16	AM	*	*	*	*	*	AM	no
B1-17	AM	*	*	*	*	*	DC	no
B1-18	AM	*	*	*	*	*	BF	yes

Table 64. Expanding node B1

B1-2	RG	DC	I/Q	DEC	FFT	FMG	THD	Possible matching
B1-2-1	AM	DC	AM	*	*	*	*	no
B1-2-2	AM	DC	DC	*	*	*	*	yes
B1-2-3	AM	DC	BF	*	*	*	*	no
B1-2-4	AM	DC	*	AM	*	*	*	no
B1-2-5	AM	DC	*	DC	*	*	*	yes
B1-2-6	AM	DC	*	BF	*	*	*	no
B1-2-7	AM	DC	*	*	AM	*	*	no
B1-2-8	AM	DC	*	*	DC	*	*	no
B1-2-9	AM	DC	*	*	BF	*	*	no
B1-2-10	AM	DC	*	*	*	AM	*	no
B1-2-11	AM	DC	*	*	*	DC	*	no
B1-2-12	AM	DC	*	*	*	BF	*	yes
B1-2-13	AM	DC	*	*	*	*	AM	no
B1-2-14	AM	DC	*	*	*	*	DC	no
B1-2-15	AM	DC	*	*	*	*	BF	yes

Table 65. Expanding node B1-2

B1-2-2	RG	DC	I/Q	DEC	FFT	FMG	THD	Possible matching
B1-2-2-1	AM	DC	DC	AM	*	*	*	no
B1-2-2-2	AM	DC	DC	DC	*	*	*	yes
B1-2-2-3	AM	DC	DC	BF	*	*	*	no
B1-2-2-4	AM	DC	DC	*	AM	*	*	no
B1-2-2-5	AM	DC	DC	*	DC	*	*	no
B1-2-2-6	AM	DC	DC	*	BF	*	*	no
B1-2-2-7	AM	DC	DC	*	*	AM	*	no
B1-2-2-8	AM	DC	DC	*	*	DC	*	no
B1-2-2-9	AM	DC	DC	*	*	BF	*	yes
B1-2-2-10	AM	DC	DC	*	*	*	AM	no
B1-2-2-11	AM	DC	DC	*	*	*	DC	no
B1-2-2-12	AM	DC	DC	*	*	*	BF	yes

Table 66. Expanding node B1-2-2

B1-2-2-2	RG	DC	I/Q	DEC	FFT	FMG	THD	Possible matching
B1-2-2-2-1	AM	DC	DC	DC	AM	*	*	no
B1-2-2-2-2	AM	DC	DC	DC	DC	*	*	no
B1-2-2-2-3	AM	DC	DC	DC	BF	*	*	no
B1-2-2-2-4	AM	DC	DC	DC	*	AM	*	no
B1-2-2-2-5	AM	DC	DC	DC	*	DC	*	no
B1-2-2-2-6	AM	DC	DC	DC	*	BF	*	yes
B1-2-2-2-7	AM	DC	DC	DC	*	*	AM	no
B1-2-2-2-8	AM	DC	DC	DC	*	*	DC	no
B1-2-2-2-9	AM	DC	DC	DC	*	*	BF	yes

Table 67. Expanding node B1-2-2-2

B1-2-2-2-6	RG	DC	I/Q	DEC	FFT	FMG	THD	Possible matching
B1-2-2-2-6-1	AM	DC	DC	DC	AM	BF	*	no
B1-2-2-2-6-2	AM	DC	DC	DC	DC	BF	*	no
B1-2-2-2-6-3	AM	DC	DC	DC	BF	BF	*	no
B1-2-2-2-6-4	AM	DC	DC	DC	*	BF	AM	no
B1-2-2-2-6-5	AM	DC	DC	DC	*	BF	DC	no
B1-2-2-2-6-6	AM	DC	DC	DC	*	BF	BF	yes

Table 68. Expanding node B1-2-2-2-6

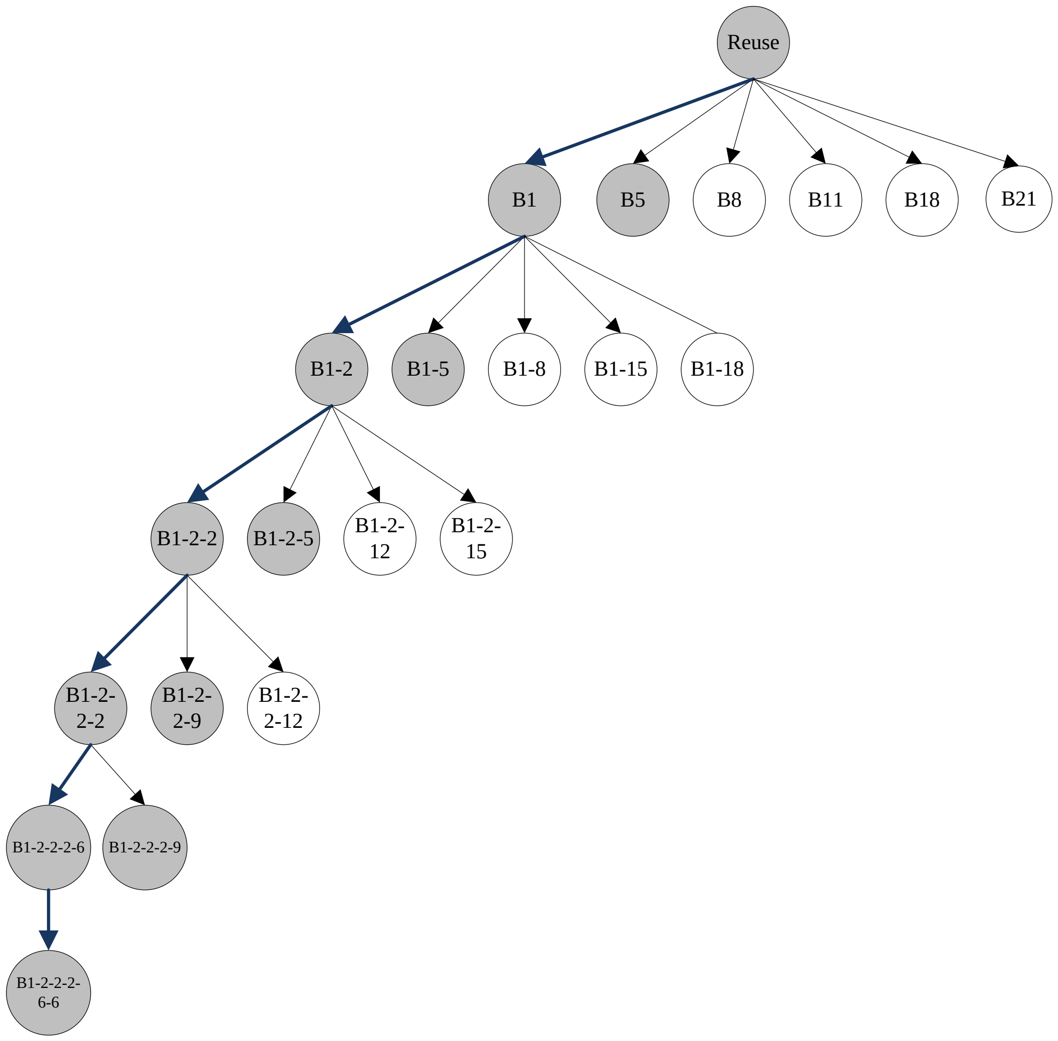


Figure 56 Reusable assets mappings

The apparently best solution is B1-2-2-2-6-6 {AM, DC, DC, DC, *, BF, BF} as shown in Figure 56 and Table 69. Six out of seven (86 percent) requirements are mapped to reusable assets; only one requirement (14 percent of all requirements), the Fast Fourier Transform Doppler filter bank, cannot be mapped to an intellectual property module, and we will have to design this module.

Requirements	Mapped IP cores
RG (range gate)	AM (ADC acquisition and memory control)
DC (digital down convert)	DC (DC IP core)
IQ (I/Q samples)	DC (DDC IP core)
DEC (decimation)	DC (DDC IP core)
FFT (fast Fourier transform)	None
FMG (form filter signal magnitude)	BF (beamforming IP core)
THD (threshold detection)	BF (beamforming IP core)

Table 69. Map requirements to IP cores

B. TESTS

1. Purpose of Our Tests

The purpose of this section is to prove that we are able to implement correct designs consistent with the design pattern mapping in Table 69. Filter signal magnitude forming (FMG) and threshold detection (THD) were not tested due to the lack of funding as shown in Table 70.

Requirements	Mapped IP cores	Tested
RG (range gate)	AM (ADC acquisition and memory control)	Yes
DC (digital down convert)	DC (DDC IP core)	Yes
IQ (I/Q samples)	DC (DDC IP core)	Yes
DEC (decimation)	DC (DDC IP core)	Yes
FFT (fast Fourier transform)	None	Yes
FMG (form filter signal magnitude)	BF (beamforming IP core)	No
THD (threshold detection)	BF (beamforming IP core)	No

Table 70. Map requirements to IP cores

2. Test Specifications

Our overall goal is to demonstrate the capability in programing Pentek digital signal processing card to perform pulse Doppler processing to detect two Doppler shifts, one at 40 kilohertz and the other at 25 kilohertz; the carrier frequency is at 5 megahertz. Detailed test specifications are in Section B.4 [85].

3. Pentek Software IP Cores

There are three major IP cores built in Pentek embedded system and they are analog-to-digital converter acquisition and memory control, digital down-converter (DDC) and beamformers as described in Table 71 [86].

IP cores	Description
ADC acquisition & memory control	"Each IP module can receive data from any of the four ADCs or a test signal generator. Each IP module has an associated memory bank for buffering data in FIFO mode or for storing data in transient capture mode. All memory banks are supported with DMA engines for easily moving ADC data through the PCIe interface. DMA, direct memory access, is a way to access memory without going through the central processing unit. PCIe (peripheral components interconnect express) is a high-speed serial computer expansion standard."[86]
DDC (digital down converter)	"Each DDC has an independent 32-bit tuning frequency setting that ranges from DC to ƒs, where ƒs is the ADC sampling frequency. Each DDC can have its own unique decimation setting, supporting as many as four different output bandwidths for the board. Decimations can be programmed from 2 to 65,536 providing a wide range to satisfy most applications. The decimating filter for each DDC accepts a unique set of user-supplied 18-bit coefficients. The 80% default filters deliver an output bandwidth of 0.8*ƒs/N, where N is the decimation setting. The rejection of adjacent-band components within the 80% output bandwidth is better than 100 dB. Each DDC delivers a complex output stream consisting of 24-bit I + 24-bit Q or16-bit I + 16-bit Q samples at a rate of ƒs/N."[86]
Beamformer	"Each DDC core contains programmable I & Q phase and gain adjustments followed by a power meter that continuously measures the individual average power output. In addition, each DDC core includes a threshold detector to automatically send an interrupt to the processor if the average power level of any DDC core falls below or exceeds a programmable threshold. A programmable summation block provides summing of any of the four DDC core outputs. A power meter and threshold detect block is provided for the summed output."[86]

Table 71. Pentek intellectual property cores

4. Tests Configurations, Methodology and Results

a. *Signals and IP Cores Configurations*

The first input signal was a pulse radar signal at 5.04 megahertz with pulse width of 1.14 microseconds. This signal simulated an echo at 5 megahertz carrier frequency with 40 kilohertz Doppler shift frequency as shown in Table 72.

Input signal #1		
	Frequency	**Period**
Input signal	5.04E+06	
Tuning frequency	5.00E+06	
Doppler shift	4.00E+04	2.5E-05
Pulse width	1.14E-06	

Table 72. Input signal #1 characteristics

The second input signal was a pulse radar signal at 5.025 megahertz with pulse width of 1.14 microseconds. This signal simulated an echo at 5 megahertz carrier frequency with 25 kilohertz Doppler shift frequency as shown in Table 73.

Input signal #2		
	Frequency	**Period**
Input signal	5.025E+06	
Tuning frequency	5.00E+06	
Doppler shift	2.50E+04	4.00E-05
pulse width	1.14E-06	

Table 73. Input signal #2 characteristics

The periodic frequency (PRF) is 109,375 hertz; the PRF is the number of pulses per second. The reciprocal of the pulse repetition frequency is the pulse repetition period (PRT) as shown in Table 74.

Trigger (PRF)		
	Frequency (PRF)	**Period (PRT)**
trigger period	109,375	9.14E-06

Table 74. External trigger characteristics

We removed the 5 megahertz carrier frequency form the input signals with a DDC at a tuning frequency of 5 megahertz. DDC stands for digital down-converter which converts a digitized signal to a baseband signal. Baseband is the original band of frequencies of the signal before being modulated with 5 megahertz carrier for transmission. After down conversion, we decimated the baseband signal at a factor of 16; decimation means a reduction in the number of samples. For this case, we kept every 16th sample of the input digital signal. DDC sampling rate is calculated as (ADC sampling rate ÷ 16) = (56 megahertz ÷ 16) = 3.5 megahertz. The configurations of DDC IP core are listed in Table 75.

DDC		
	Frequency	**Period**
decimation	16	
DDC sampling rate	3.50E+06	2.86E-07

Table 75. DDC IP core configurations

The configurations of ADC IP core are listed in Table 76. The ADC sampling rate is 56 megahertz.

ADC			
	Frequency	**Period**	**Note**
ADC sampling rate	5.60E+07	1.79E-08	
software delay (1st)	60	1.125E-06	Delay after the trigger
#ADC samples (1st)	80	1.429E-06	Samples after the 1st delay
software delay (2nd)	11	2.500E-07	Delay after the 1st sampling
		2.804E-06	
#ADC samples (2nd)	80	1.429E-06	Samples after the 2nd delay
		4.232E-06	

Table 76. ADC IP core configurations

b. *Methodology (Software Program in C Programming Language)*

The program for the FPGA is briefly described below:

- Step 1: Wait for an external trigger.
- Step 2: Once triggered, delay for 60 ADC cycles.
- Step 3: Capture ADC samples for 80 ADC cycles.
- Step 4: Decimate at 16 (only keep 16th data sample).
- Step 5: Filter out aliasing and noise.
- Step 6: Store 4 DDC samples into a FIFO memory.
- Step 7: Delay for another 11 ADC cycles.
- Step 8: Capture ADC samples for 80 ADC cycles.
- Step 9: Decimate at 16 (only keep 16th data sample).

- Step 10: Filter out aliasing and noise.
- Step 11: Store 4 DDC samples into the same FIFO memory.
- Repeat steps 1 through 11 for 64 times.
- Transfer FIFO data to a workstation.
- Separate signals one and two into two different data files.
- Apply FFT to both data files by using MATLAB.

c. *Test Results*

Figure 57 and Figure 58 show the captured two Doppler shifts in two-dimensional and three-dimensional plots. The frequency and power level for the first Doppler shift are 25 kilohertz and about 90,000 raw counts. The frequency and power level for the second Doppler shift are 40 kilohertz and about 130,000 raw counts.

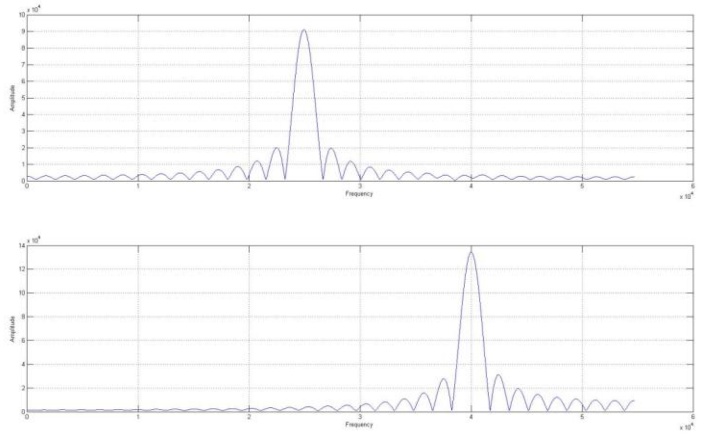


Figure 57 Two detected pseudo pulse Doppler target returns in a 2-D plot

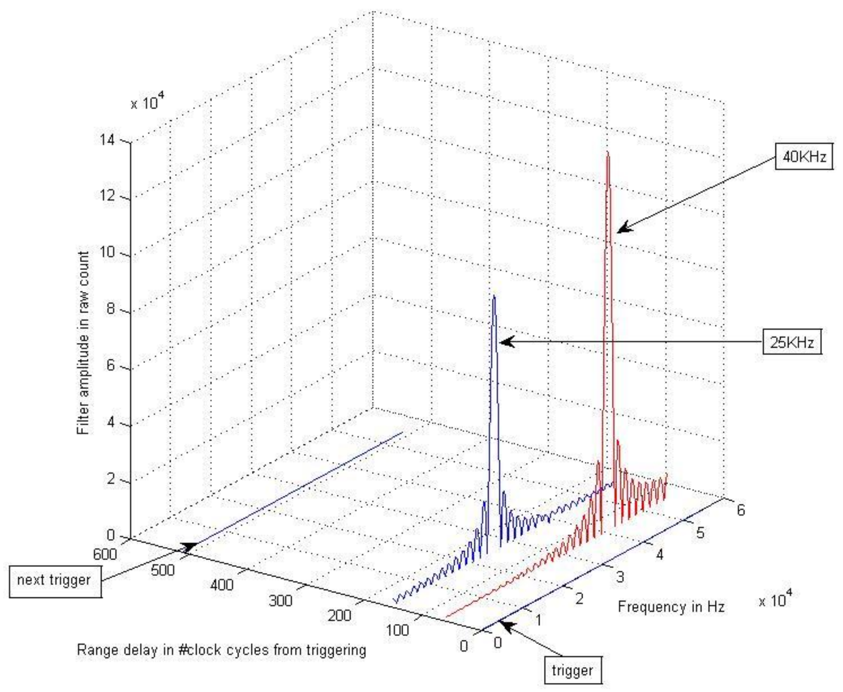


Figure 58 Two detected pseudo pulse Doppler target returns in a 3-D plot

The generated source Doppler signals are shown in Figure 59. The orange square wave represents the trigger signals; the taller blue pulse represents the 40 kilohertz Doppler shift at about 4.2 volts; the shorter blue pulse represents the 25 kilohertz Doppler shift at about 3 volts; the green and purple pulses are used to measure timing for blue pulses (Doppler shifts).

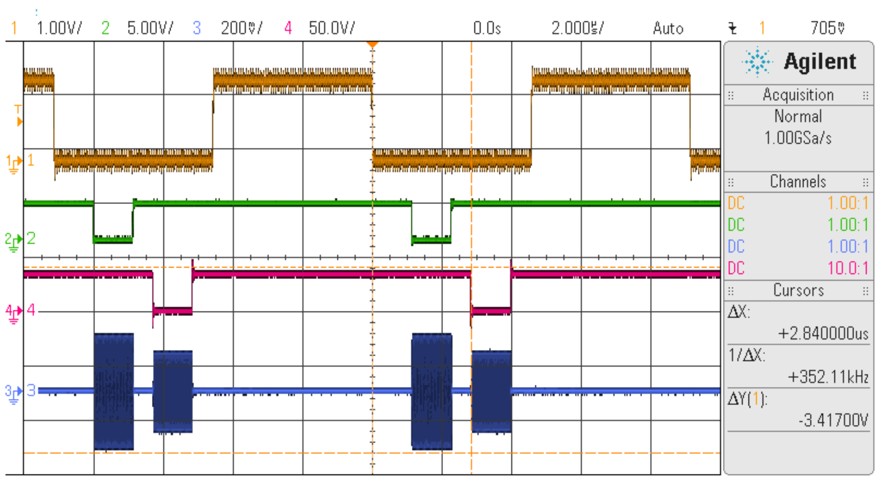


Figure 59 Two input signals before detection

For this case study, we are only looking for the presence of Doppler shifts and their frequencies; the exact power levels of these signals are not important. As a result, based on Table 77, we conclude that our test is successful.

		Generated	Captured
	Delay	60 ADC cycles	60 ADC cycles
1st Doppler	Frequency	40 kilohertz	40 kilohertz
	Power	4.2 volts	140K raw count
	Delay	11 ADC cycles	11 ADC cycles
2nd Doppler	Frequency	25 kilohertz	25 kilohertz
	Power	3.0 volts	90K raw count

Table 77. Generated and captured Doppler shifts

C. TESTS CONCLUSION

By using our new software/firmware/hardware codesign methodology, we showed we can partition the embedded system into appropriate modalities and then map them to existing intellectual properties for design efficiently.

We would like to point out that this is not the first time we worked on this project. From 2011 to 2012, we worked on the same project with ten engineers, $3.16 million and a period of more than 12 months, but failed to deliver any software product that worked, even though all software and hardware parts were procured and partial analog hardware (e.g., a waveform generator and an analog microwave receiver) was designed and built.

In 2013, with two engineers, $90K and a period of five months, we were able to deliver a Doppler range gating successfully without reusing any software tools and design from the previous work. This was because the software tool license from the first attempt had expired and we could not afford to pay the annual renewal fee of $45,000 in 2013 and there were no software deliverables from the first attempt anyway. Table 78 shows the comparisons between these two attempts; the first attempt did not use any systematic methodology and the second attempt used the systematic software/firmware/hardware codesign methodology.

SW/FW/HW methodology	Duration	Cost	Man	Software deliverables
No	12+ months (2011-2012)	$3.16M	10	None
Yes	5 months (May-Sep, 2013)	$95K	2	Doppler range gating
Ratio	3:1	33:1	5:1	

Table 78. Comparisons between with and without the SW/FW/HW methodology

VII. CASE STUDY THREE—HIDDEN MARKOV MODEL

A. INTRODUCTION

A hidden Markov model (HMM) is a triple (Π, A, B) as described below [87].

Π = (π_i) is the vector of the initial state probabilities, where $1 \leq i \leq M$, M = the number of hidden states; each hidden state has M outgoing transitions to the other M-1 states and back to itself.

$T = (a_{ij})$ is the transition matrix; $\Pr(x_{i_t} | x_{j_{t-1}})$ is the probability from a hidden state x_j at time t-1 to another hidden state x_i at time t.

$C = (b_{ij})$ is the confusion matrix; $\Pr(y_i | x_j)$ is the probability from a hidden state x_j to an observed state y_i at time t.

Two assumptions are made when calculating a hidden Markov model: one is that each probability in the state transition matrix and in the confusion matrix is time-independent, and the other is that the choice of state is made entirely on the basis of the previous state (first order Markov model).

Evaluation and decoding are two important applications for hidden Markov models. Evaluation uses a forward algorithm to calculate the probability of an observation sequence given a particular hidden Markov model. If a sequence of observations is described by multiple hidden Markov models, we can use the forward algorithm to select the most probable hidden Markov model. Decoding uses the Viterbi algorithm to determine the most probable sequence of hidden states given a sequence of observations for a particular hidden Markov model.

Table 79 summarizes the calculations for initialization and recursion as well as the objectives for forward algorithm and Viterbi algorithm.

	Forward algorithm	Viterbi algorithm
Observed states	$Y^{(k)} = y_{k_1}, \dots, y_{k_T}$	
Hidden states	$X_i = (X_{i_1}, X_{i_2}, \dots, X_{i_T})$	
Initial probability	$\pi(j)$	
Transition matrix	T_{ij}	
Confusion matrix	C_{jk_1}	
t=1 Initialization	$\alpha_1(j) = \pi(j) C_{jk_1}$	$\delta_1(j) = \pi(j) C_{jk_1}$
t=2 to N Recursion	$\alpha_{t+1}(j) = \sum_{i=1}^{n} \alpha_t(i)\, T_{ij} C_{jk_t}$	$\delta_t(i) = \max_j (\delta_{t-1}(j) T_{ji}\, C_{ik_t})$
Objective	$\Pr\left(Y^{(k)}\right) = \sum_{j=1}^{n} \alpha_T(j)$	$\phi_t(i) = \operatorname{argmax}_j (\delta_{t-1}(j) T_{ji})$

Table 79. Forward algorithm and Viterbi algorithm

B. FORWARD ALGORITHM CASE STUDY

1. Develop Requirements and Define Constraints

The requirement is to map the functional components in the forward algorithm to software, firmware or hardware. The constraint is to assign all functional components to one single modality (software, firmware or hardware) without mixing them together to avoid interface design among different platforms (modalities).

2. Form an Architecture

The functional components are an analog-to-digital converter (ADC), a process for extracting initial state probabilities (I), a process for extracting probabilities from a transition matrix (T), a process for extracting probabilities from a confusion matrix (C), recursive computations (partial probabilities) (R), and the sum of all partial probabilities (S) as shown in Figure 60.

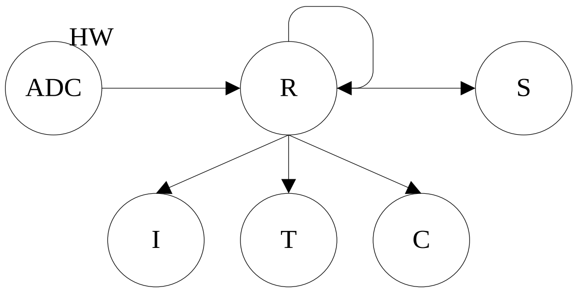


Figure 60 Hidden Markov model architecture

The analog-to-digital converter (ADC) must be hardware since input signal is analog. The initial, transition and confusion probabilities are predefined in memory. The forward algorithm is based on recursive computations, i.e. obtaining the new value by using the old value.

3. Build a Tree to Map Functions to Modalities

To simplify the design, we will focus on the core computations of forward algorithm without considering human interface software and system configurations. In addition, we prefer to assign all remaining functions (I, T, C, R and S) to one single modality (software, firmware or hardware) without mixing them together to avoid interface design among different platforms (modalities).

Table 80 and Table 81 show the first mapping solution (software) and Table 82 shows the second mapping solution (firmware) for the two possible options shown in Figure 61. Here, “ADC” stands for analog-to-digital converter; “I” stands for initialization; “T” stands for transition matrix; “C” stands for confusion matrix; “R” stands for recursion; “S” stands for sum of partial probabilities.

A	ADC	I	T	C	R	S	Possible Mapping	Justification
A1	SW	*	*	*	*	*	NO	ADC must be hardware
A2	FW	*	*	*	*	*	NO	ADC must be hardware
A3	HW	*	*	*	*	*	OK	ADC must be hardware
A4	*	SW	*	*	*	*	OK	
A5	*	FW	*	*	*	*	OK	
A6	*	HW	*	*	*	*	NO	too costly, not flexible
A7	*	*	SW	*	*	*	OK	
A8	*	*	FW	*	*	*	OK	
A9	*	*	HW	*	*	*	NO	too costly, not flexible
A10	*	*	*	SW	*	*	OK	
A11	*	*	*	FW	*	*	OK	
A12	*	*	*	HW	*	*	NO	too costly, not flexible
A13	*	*	*	*	SW	*	OK	
A14	*	*	*	*	FW	*	OK	
A15	*	*	*	*	HW	*	NO	too costly, not flexible
A16	*	*	*	*	*	SW	OK	
A17	*	*	*	*	*	FW	OK	
A18	*	*	*	*	*	HW	NO	too costly, not flexible

A3	ADC	I	T	C	R	S	Possible Mapping	Justification
A3-1	HW	SW	*	*	*	*	OK	
A3-2	HW	FW	*	*	*	*	OK	
A3-3	HW	HW	*	*	*	*	NO	too costly, not flexible
A3-4	HW	*	SW	*	*	*	OK	
A3-5	HW	*	FW	*	*	*	OK	
A3-6	HW	*	HW	*	*	*	NO	too costly, not flexible
A3-7	HW	*	*	SW	*	*	OK	
A3-8	HW	*	*	FW	*	*	OK	
A3-9	HW	*	*	HW	*	*	NO	too costly, not flexible
A3-10	HW	*	*	*	SW	*	OK	
A3-11	HW	*	*	*	FW	*	OK	
A3-12	HW	*	*	*	HW	*	NO	too costly, not flexible
A3-13	HW	*	*	*	*	SW	OK	
A3-14	HW	*	*	*	*	FW	OK	
A3-15	HW	*	*	*	*	HW	NO	too costly, not flexible

Table 80. Nodes A and A3

Both A3-1 (I=SW) and A3-2 (I=FW) are acceptable, but first we will explore the branch for I=SW.

A3-1	ADC	I	T	C	R	S	Possible Mapping	Justification
A3-1-1	HW	SW	SW	*	*	*	OK	
A3-1-2	HW	SW	FW	*	*	*	NO	mutually exclusive
A3-1-3	HW	SW	HW	*	*	*	NO	too costly, not flexible
A3-1-4	HW	SW	*	SW	*	*	OK	
A3-1-5	HW	SW	*	FW	*	*	NO	mutually exclusive
A3-1-6	HW	SW	*	HW	*	*	NO	too costly, not flexible
A3-1-7	HW	SW	*	*	SW	*	OK	
A3-1-8	HW	SW	*	*	FW	*	NO	mutually exclusive
A3-1-9	HW	SW	*	*	HW	*	NO	too costly, not flexible
A3-1-10	HW	SW	*	*	*	SW	OK	
A3-1-11	HW	SW	*	*	*	FW	NO	mutually exclusive
A3-1-12	HW	SW	*	*	*	HW	NO	too costly, not flexible

A3-1-1	ADC	I	T	C	R	S	Possible Mapping	Justification
A3-1-1-1	HW	SW	SW	SW	*	*	OK	
A3-1-1-2	HW	SW	SW	FW	*	*	NO	mutually exclusive
A3-1-1-3	HW	SW	SW	HW	*	*	NO	too costly, not flexible
A3-1-1-4	HW	SW	SW	*	SW	*	OK	
A3-1-1-5	HW	SW	SW	*	FW	*	NO	mutually exclusive
A3-1-1-6	HW	SW	SW	*	HW	*	NO	too costly, not flexible
A3-1-1-7	HW	SW	SW	*	*	SW	OK	
A3-1-1-8	HW	SW	SW	*	*	FW	NO	mutually exclusive
A3-1-1-9	HW	SW	SW	*	*	HW	NO	too costly, not flexible

A3-1-1-1	ADC	I	T	C	R	S	Possible Mapping	Justification
A3-1-1-1-1	HW	SW	SW	SW	SW	*	OK	
A3-1-1-1-2	HW	SW	SW	SW	FW	*	NO	mutually exclusive
A3-1-1-1-3	HW	SW	SW	SW	HW	*	NO	too costly, not flexible
A3-1-1-1-4	HW	SW	SW	SW	*	SW	OK	
A3-1-1-1-5	HW	SW	SW	SW	*	FW	NO	mutually exclusive
A3-1-1-1-6	HW	SW	SW	SW	*	HW	NO	too costly, not flexible

A3-1-1-1-1	ADC	I	T	C	R	S	Possible Mapping	Justification
A3-1-1-1-1-1	HW	SW	SW	SW	SW	SW	OK	solution
A3-1-1-1-1-2	HW	SW	SW	SW	SW	FW	NO	mutually exclusive
A3-1-1-1-1-3	HW	SW	SW	SW	SW	HW	NO	too costly, not flexible

Table 81. Nodes A3-1, A3-1-1, A3-1-1-1 and A3-1-1-1-1

One solution is node A3-1-1-1-1-1 = {ADC, I, T, C, R, S} = {HW, SW, SW, SW, SW, SW}. We can get another solution by expanding node A3-2 (I=FW).

A3-2	ADC	I	T	C	R	S	Possible Mapping	Justification
A3-2-1	HW	FW	SW	*	*	*	NO	mutually exclusive
A3-2-2	HW	FW	FW	*	*	*	OK	
A3-2-3	HW	FW	HW	*	*	*	NO	too costly, not flexible
A3-2-4	HW	FW	*	SW	*	*	NO	mutually exclusive
A3-2-5	HW	FW	*	FW	*	*	OK	
A3-2-6	HW	FW	*	HW	*	*	NO	too costly, not flexible
A3-2-7	HW	FW	*	*	SW	*	NO	mutually exclusive
A3-2-8	HW	FW	*	*	FW	*	OK	
A3-2-9	HW	FW	*	*	HW	*	NO	too costly, not flexible
A3-2-10	HW	FW	*	*	*	SW	NO	mutually exclusive
A3-2-11	HW	FW	*	*	*	FW	OK	
A3-2-12	HW	FW	*	*	*	HW	NO	too costly, not flexible

A3-2-2	ADC	I	T	C	R	S	Possible Mapping	Justification
A3-2-2-1	HW	FW	FW	SW	*	*	NO	mutually exclusive
A3-2-2-2	HW	FW	FW	FW	*	*	OK	
A3-2-2-3	HW	FW	FW	HW	*	*	NO	too costly, not flexible
A3-2-2-4	HW	FW	FW	*	SW	*	NO	mutually exclusive
A3-2-2-5	HW	FW	FW	*	FW	*	OK	
A3-2-2-6	HW	FW	FW	*	HW	*	NO	too costly, not flexible
A3-2-2-7	HW	FW	FW	*	*	SW	NO	mutually exclusive
A3-2-2-8	HW	FW	FW	*	*	FW	OK	
A3-2-2-9	HW	FW	FW	*	*	HW	NO	too costly, not flexible

3-2-2-2	ADC	I	T	C	R	S	Possible Mapping	Justification
A3-2-2-2-1	HW	FW	FW	FW	SW	*	NO	mutually exclusive
A3-2-2-2-2	HW	FW	FW	FW	FW	*	OK	
A3-2-2-2-3	HW	FW	FW	FW	HW	*	NO	too costly, not flexible
A3-2-2-2-4	HW	FW	FW	FW	*	SW	NO	mutually exclusive
A3-2-2-2-5	HW	FW	FW	FW	*	FW	OK	
A3-2-2-2-6	HW	FW	FW	FW	*	HW	NO	too costly, not flexible

A3-2-2-2-2	ADC	I	T	C	R	S	Possible Mapping	Justification
A3-2-2-2-2-1	HW	FW	FW	FW	FW	SW	NO	mutually exclusive
A3-2-2-2-2-2	HW	FW	FW	FW	FW	FW	OK	solution
A3-2-2-2-2-3	HW	FW	FW	FW	FW	HW	NO	too costly, not flexible

Table 82. Nodes A3-2, A3-2-2, A3-2-2-2 and A3-2-2-2-2

Another solution is node A3-2-2-2-2-2 = {ADC, I, T, C, R, S} = {HW, FW, FW, FW, FW, FW}. Figure 61 shows these two options.

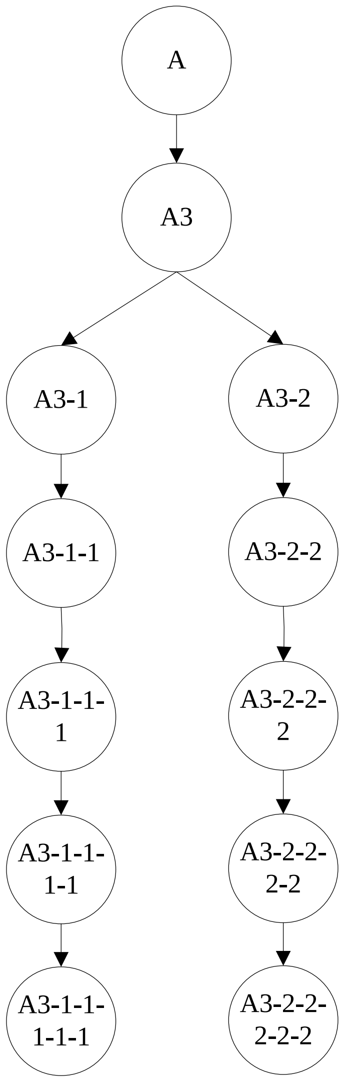


Figure 61 Two possible options

4. Discussion

Our methodology allows for two acceptable solutions here, so it is best to expand both on the design tree. The software implementation of the forward algorithm has the advantages of low design complexity and low cost, but its disadvantage is being slow in speed. The firmware implementation has the advantage of being fast in speed, but has the disadvantages of medium cost and high design complexity.

If speed is critical, firmware implementation is a better choice over software for two reasons. First, if there are multiple hidden Markov models for a sequence of observations, we can compute these models in parallel (using firmware) instead of serial (using software) and then select the one with the best probability for the observation. Second, each intermediate hidden state depends on the probabilities of all previous hidden states, transition probabilities, and confusion probabilities; the computations from all previous hidden states can be performed in parallel (using firmware) instead of serial (using software) for time efficiency. Table 83 summarizes the advantages and disadvantages of these two options.

Options	Option 1	Option 2
Leaf-node	A3-1-1-1-1-1	A3-2-2-2-2-2
Partitioning	Software	Firmware
Design complexity	Low	High
Speed	Slow	Fast
Cost	Low	Medium
Sequential	All sequential	Sequential for recursion
Parallel	None	Calculate partial probabilities for all M hidden states at time t in parallel (simultaneously)
Multiple hidden Markov models	Too slow for software	When multiple hidden Markov models are used, all models can be calculated simultaneously and then the model with the best probabilities is selected for the observed sequence

Table 83. Two options

C. METHODOLOGY FOR VITERBI ALGORITHM

The mapping of the Viterbi algorithm to software, firmware or hardware is similar to forward algorithm except that the summation (Σ) in the forward algorithm is replaced with max to calculate the most likely route to the current position, rather than the total probability. In addition, the Viterbi algorithm remembers the best route to the current position by maintaining a “back-pointer” through the argmax calculation. Thus a design for it will have a few differences but many similarities.

VIII. CONCLUSION AND RECOMMENDATIONS FOR FUTURE RESEARCH

A. OUR CLAIM

Based on the test results in Chapters V and VI, we claim that rather than the trial-and-error approach being currently practiced for embedded system design, our new software/firmware/hardware codesign methodology using more software engineering has the potential to systematically build correct designs efficiently to satisfy the requirements provided by the stakeholders.

B. OUR CONTRIBUTIONS

Our first contribution is to create a new software/firmware/hardware codesign methodology to systematically build correct designs efficiently to satisfy the requirements provided by the stakeholders. This codesign methodology includes requirements development, architecture forming, software/firmware/hardware partitioning, design-pattern mapping, new-design pattern synthesis, integration, and testing.

Software/hardware partitioning is difficult in codesign according to the codesign group at U.C. Berkeley. Our codesign methodology first builds an tree with conjunctions and disjunctions of possible mappings from functional components to the options of software, firmware, and hardware following requirements and constraints; second, rates the cost of each mapping; third, searches the tree to find a minimum weighted sum of the costs; last, identifies existing design patterns once design is selected and otherwise, synthesizes new design patterns.

Our second contribution is the identification of five design patterns for reconfigurable-computing based embedded systems; these design patterns could be added to the 89 patterns collected by André DeHon et al. at California Institute of Technology [50]. The data alignment design pattern can be used to align multiple parallel data bit-streams and forwarded source-synchronous sampling clocks to ensure correct data sampling. The post-deserialization bits remapping design pattern can be used to remap bit addresses after demultiplexing from an analog-to-digital converter for data processing. The pre-serialization bits remapping design pattern can be used to remap bit addresses before multiplexing for serial transmission. The polyphase DFT filter banks can be used for dividing a wide bandwidth input signal into multiple frequency subbands, processing all subbands in parallel independently and differently, and then combining the processed subbands into a single serial output for transmission. The switch-and-filter architecture design pattern can be used to move ultra-fast serial data from a faster data source to a slower filter for processing by deserializing it into parallel channels, process the data, and then combine parallel data into a single serial data stream for output. We have applied our software/firmware/hardware codesign methodology to two projects with successful results. One project was for ultra-wide instantaneous bandwidth signal digitization for a period of two years from 2011 to 2012 as described in the first case study. The other was airborne interceptor Doppler range gating for a period of five months in 2013 as described in the second case study. We also applied our methodology to a third case study that permitted more solution options, to illustrate the flexibility of the methodology, but this study has not yet been implemented.

C. FUTURE RESEARCH DIRECTIONS

To simplify the analysis, we used an ***OR*** tree and ***A**** search with embedded ***AND*** algorithm for software/firmware/hardware partitioning and design pattern mapping. To handle more complex design problems, a methodology of using an ***AND/OR*** tree and ***AO**** search algorithm should be investigated since AO* is the appropriate generalization of A* then.

Even though cost estimation was not critical for our case studies of software/firmware/hardware partitioning and in choosing design patterns, it could be important for other applications. In the future, we would like to investigate the cost estimation for low-level design implementation, such as identifying cost drivers, modelling the cost-estimating relation for each cost driver, selecting the best probability distribution models, and calculating weighted sum for different cost drivers.

More research should be conducted in identifying and cataloging design patterns for firmware and hardware because these patterns can drastically improve the success rate and efficiency of design.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. CASE STUDY ONE BACKGROUND

Appendix A presents more detailed background for our first case study in Chapter V.

A. CHALLENGES WE ARE FACING

1. Background (Two-Ray Segment Propagation Model)

Joint Electronic Warfare Effect Laboratory uses two-ray segment propagation model to simulate the transmitter-to-receiver relationship. This model suggests that the transmitting and receiving antennas are close to the ground, so that there are two paths from a transmitter to a receiver: the direct path, and a second path due to ground reflection. In lab, the distance for these paths is simulated by power attenuation. When the range is less than R_{CROSS} (crossover range) the path loss is approximated by “1 over R squared” free space model. When the range is more than R_{CROSS}, the path loss is approximated by “1 over R to the 4” model [88].

Crossover range

$$R_{CROSS} = \frac{4\pi \cdot h_1 \cdot h_2}{\lambda}$$

R_{CROSS}	= cross over range
h_1	= transmitting antenna height
h_2	= receiving antenna height
λ	= wavelength of the transmitted signal

Path loss for ranges less than crossover

$$P_R = \frac{P_t \cdot G_t \cdot G_R \cdot \lambda^2}{4\pi \cdot R^2}$$

Or in dB form:

$$P_R = P_t + G_t + G_R - 32.4 - 20 \log f - 20 \log R$$

Equivalent attenuation between a transmitter and a receiver:

$$A_R = P_R - P_t = G_t + G_R - 32.4 - 20 \log f - 20 \log R \qquad \text{equation (1)}$$

Path loss for ranges more than crossover

$$P_R = \frac{(P_t \cdot G_t \cdot G_R) \cdot (h_1 \cdot h_2)^2}{R^4}$$

Or in dB form:

$$P_R = P_t + G_t + G_R + 20 \log (h_1 \times h_2) - 40 \log R$$

Equivalent attenuation between a transmitter and a receiver:

$$A_R = P_R - P_t = G_t + G_R + 20 \log (h_1 \times h_2) - 40 \log R \qquad \text{equation (2)}$$

P_R	=	received power in dBm
P_t	=	transmitted power in dBm
G_t	=	transmitter antenna gain
G_R	=	receiver antenna gain
λ	=	transmitted wavelength
R	=	distance between transmitter and receiver in kilometers
f	=	frequency in MHz

Some assumptions are made when using the two-ray model. (1) The terrain must be relatively flat, since only one reflection is calculated. (2) The antenna gains do not vary appreciably over the desired ranges. (3) The range from the antenna is long enough to be in the far-field (the distance must be greater than $2D^2/ \lambda$, where D is the largest dimension of the antenna, λ is the transmitted wavelength). When calculating attenuation, soil type and polarizations are also considered (not addressed in this dissertation).

2. Using an Example to Illustrate the Challenges in Signal Simulation

Figure 62 shows an example of signals interference caused by a ground-jamming vehicle, represented by EMI (electromagnetic interference), to the communication between two airplanes (a transmitter and a receiver). The power loss due to distance is simulated by using a two-ray segment propagation software model.

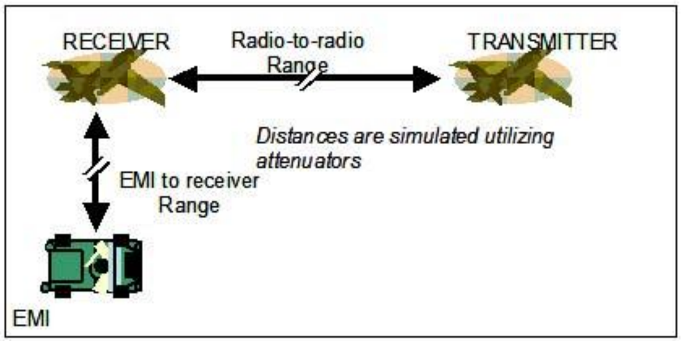


Figure 62 Distance is simulated by path loss

To simulate field condition, we use path loss equations to calculate signal attenuation caused by distance, and apply this calculated power attenuation to a programmable attenuator. In addition, a combiner is used to combine signals as an antenna receiving multiple radiofrequency signals. A splitter is used to split signal into multiple equal amplitude signals so that signals can be tested and measured simultaneously. For example, the lab simulation for the signal interference among two airplanes and one ground-jamming vehicle is shown in Figure 63. The functions and limitations for each component used in this simulation are described in Table 84.

	Purpose	Limitation
Attenuators	In field, a received signal gets weaker with increasing distance. In lab, the signal is weakened with an attenuator.	Time delay, accuracy, flat fading response, power consumption
Combiners	In field, signal and EMI combine in the radio receiver antenna. In lab, we use combiner to combine signals.	Power loss, narrow bandwidth, isolation (sneak path). Sneak path is defined as a signal at one input of a combiner/splitter sneaks over the other input.
Splitters	Split the signal into two signals with equal power for test and measurement.	

Table 84. Purpose and limitation of attenuators, combiners, splitters

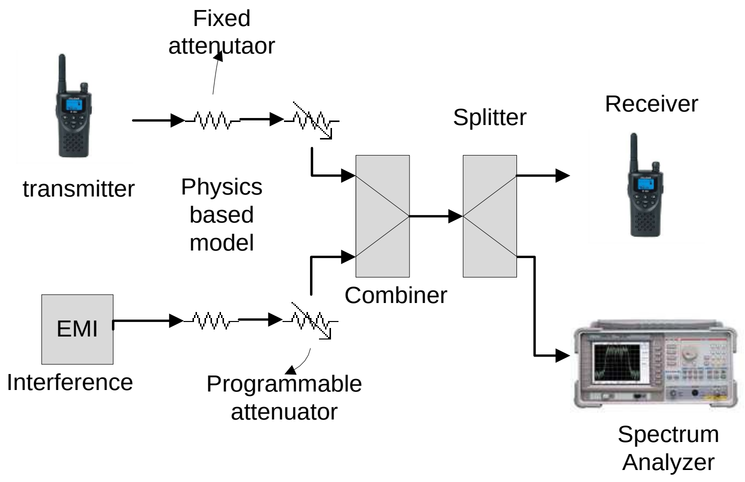


Figure 63 Simulation by using programmable attenuators

Another limitation of analog approach to signals simulation is that the number of interconnections among signal sources increases unscalably when the number of sources increases as illustrated in Table 85, Figure 64 and Figure 65.

The number of connections among N transmitters/receivers for analog and digital approaches can be calculated in equations (3) and (4).

$$N\text{ (analog)} = N \times (N-1)\text{, bidirectional} \qquad \text{equation (3)}$$

$$N\text{ (digital)} = 2 \times N\text{, bidirectional} \qquad \text{equation (4)}$$

$$\frac{N(analog)}{N(digital)} = \frac{N(N-1)}{2N} = \frac{N-1}{2}$$

A comparison between analog and digital approaches is listed in Table 85 for 1, 2, 3, 4, 8, 16, 32, 64 and 128 signal sources.

#sources	Analog Inter-connections	Digital inter-connections
1	-	2
2	2	4
3	6	6
4	12	8
8	56	16
16	240	32
32	992	64
64	4,032	128
128	16,256	256

Table 85. Analog and digital interconnections

To show the comparison in complexity graphically between analog and digital approaches, four signal sources are used in Figure 64 and eight signal sources are used in Figure 65.

N = 4

N (analog) = 4×3 = 12 (bi-directional)

N (digital) = 8 (bi-directional)

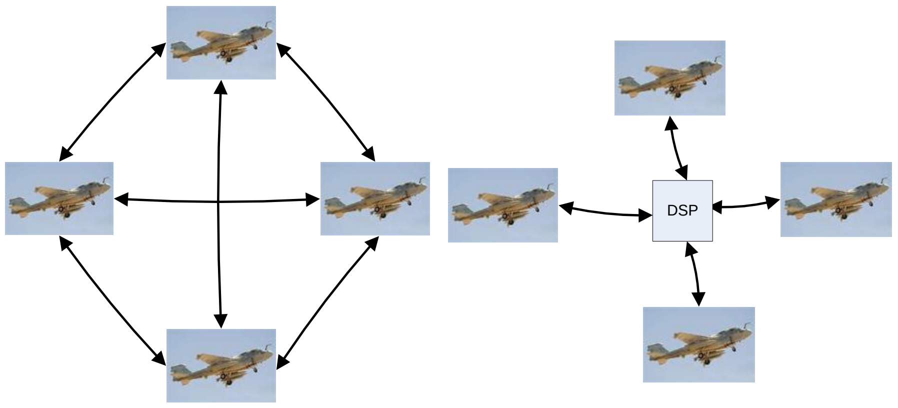


Figure 64 N=4, N(analog)=12, N(digital)=8

N=8

N (analog) = 8×7 = 56 (bi-directional)

N (digital) = 16 (bi-directional)

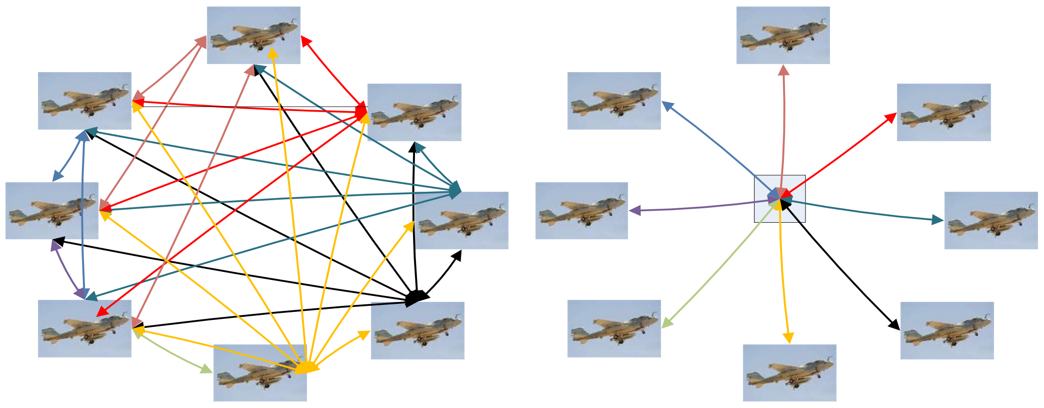


Figure 65 N=8, N(analog)=56, N(digital)=16

To reduce the limitations in Table 84 and Table 85, an ADC/FPGA/DAC system was used consisting of (1) an analog-to-digital converter, (2) an FPGA, and (3) a digital-to-analog converter. An analog-to-digital converter converts input radiofrequency signals to digital data. An FPGA reads digital data from an analog-to-digital converter, process them, and then outputs the processed data to a digital-to-analog converter. The digital-to-analog converter in terms converts the processed digital data to radiofrequency signals. With this approach, we can (1) simulate frequency-dependent power attenuation by using polyphase discrete Fourier transform filter banks inside an FPGA (attenuation factors are calculated by an external personal computer), (2) combine signals by superimposing (adding and subtracting) numerical data, (3) split signals by numerical duplications, and (4) replace physical interconnections with software in an FPGA as shown in Figure 66.

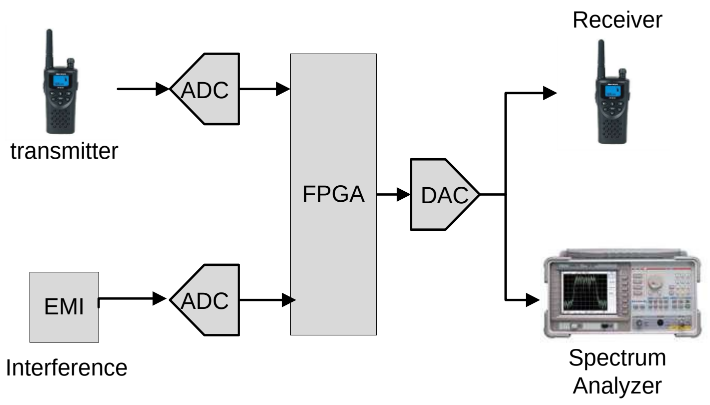


Figure 66 Simulation by using an ADC/FPGA/DAC system

B. HOW DO ADC/FPGA/DAC SYSTEMS SOLVE OUR PROBLEMS

The ADC/FPGA/DAC approach solves several problems inherent to the analog approach.

- Frequency-fading issue: From equations (1) and (2), we can see that the power attenuation is not only dependent of distance, but also frequency. Polyphase filter banks permit easier implementation of this.
- Power-loss issue: There will be no power insertion loss caused by analog combiners and splitters, since we are dealing with pure numbers inside an FPGA. A total power loss budget for analog and digital approaches are listed in Table 86. QD4-Linker and QD8-Linker are analog attenuators for

4 and 8 radiofrequency sources respectively used in Joint Electronic Warfare Effects Laboratory (JEWEL) for signal simulation.

Power Loss	Analog	Digital
QD4-Linker (4 RF sources)	28dB	None
QD8-Linker (8 RF sources)	32dB	None

Table 86. Analog and digital power budget

- Isolation Issue: A “sneak path” is no longer an issue since we combine multiple signals by superimposing them together digitally (numerically) by addition and subtraction.
- Time Delay Issue: The time delay for an electromechanical relay attenuator is replaced by software attenuation. The delay is in microseconds instead of hundreds-of-milliseconds. The timing budget for analog and digital approaches is listed in Table 87.

	Analog	Digital
QD4-Linker (4 RF sources)	900 ms	3 us ± 5 ns
QD8-Linker (8 RF sources)	300 ms	3 us ± 5 ns

Table 87. Analog and digital time delays

- Proximity issue: If current non-software and non-digital technologies are used for electronic warfare simulation, all electronic signals being tested must be located in proximity (most likely inside the same laboratory). However, once electronic signals are digitized, distributed testing can be achieved. For example, a remote digitized signal can be fed into a digital signal processing unit (an FPGA for our case) through a network.
- Recordability issue: Due to the nature of analog signals and unavailability of recording media for high-speed signals, laboratory software simulation with hardware-in-the-loop is based on real-time measurements. By digitization, the signals can be stored more easily in a permanent memory for further processing. This implies that not all real signal sources must be present in a simulated electronic warfare environment, as some signal sources can be played back from a memory device.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. TEKTRONIX ADC/FPGA/DAC DEMO SYSTEM

Our first case study in Chapter V is based on a Tektronix DCM–Digitizer/DCM-DAC/HAPS-DSP single channel demo system as shown in Figure 67. DCM stands for data converter module. DCM-Digitizer is an 8-bit analog-to-digital converter converting analog input signal to digital format. DCM-DAC is a 10-bit digital-to-analog converter converting digital data to analog waveform. HAPS-64 has two Xilinx Virtex-6 FPGAs for digital signal processing.

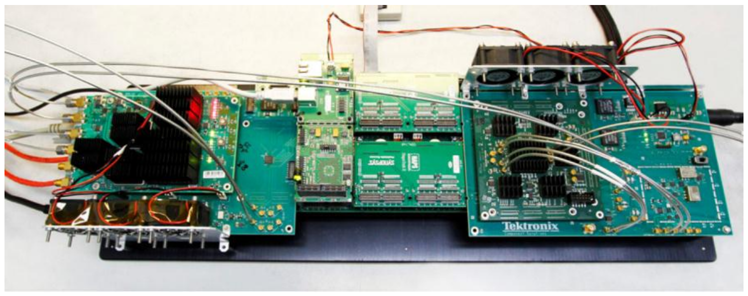


Figure 67 ADC/FPGA/DAC demo system

Simplified and detailed overall architecture diagrams are shown in Figure 68 and Figure 69. The operations of each component are described in the subsequent sections.

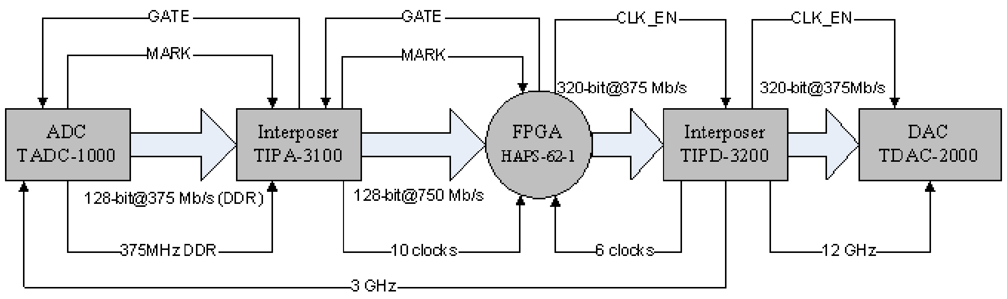


Figure 68 A simplified overall architecture for our case study

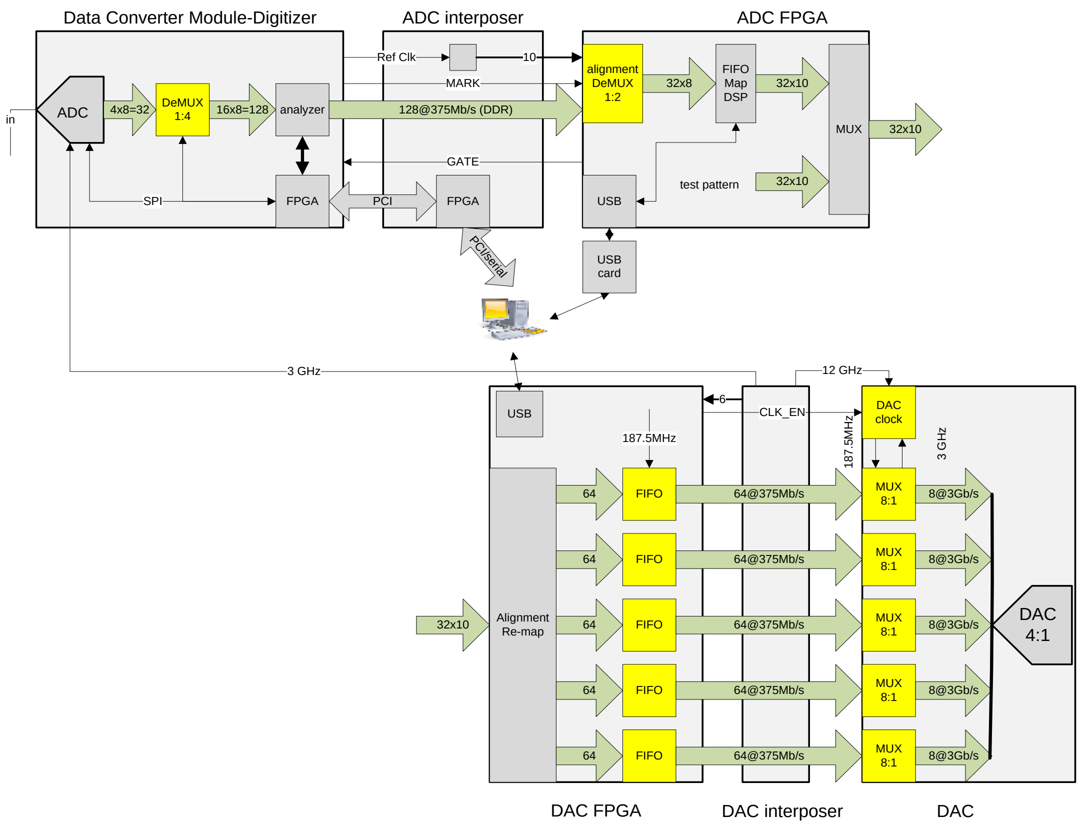


Figure 69 A detailed overall architecture for our case study

A. TADC-1000 DIGITIZER

The digitizer converts input analog signals to digital signals at a clock rate of 3 GHz. Inside the analog-to-digital converter, there are 4 interleaved analog-to-digital converters (A, B, C and D), so the output data rate at the analog-to-digital converter is 3 GS/s × 4 (channels) = 12 GS/s, or 3 GS/s × 4 channels × 8 (bits/channel) = 96 Gb/s. These 4 channels (32 bits) are further demultiplexed by 4 to 128 bits, so that the sampling rate can be decreased from 3 GS/s to 375 Mb/s (DDR), since 375 Mb/s × 2 × 128 bits = 96 Gb/s.

The output samples from the analog-to-digital converter are in the following sequence:

A1, B1, C1, D1, A2, B2, C2, D2, A3, B3, C3, D3, A4, B4, C4, D4

The GATE signal can be used to start and stop data output from the module. The MARK signal flags the output of each 128-bit word of the data capture triggered by the assertion of the GATE signal to a precision of one sample and a resolution or granularity of 16 samples. Figure 70 shows the internal architecture for TADC-1000 digitizer.

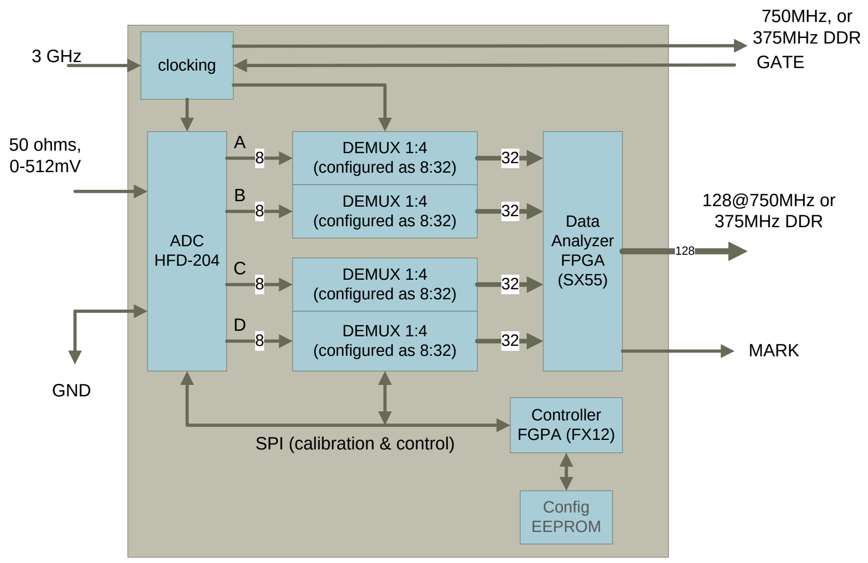


Figure 70 TADC-1000 architecture

B. TIPA-3100 ADC INTERPOSER

The ADC interposer passes 128-bit data for maximum data integrity. A double data rate (DDR) reference clock from the digitizer is buffered and multiplied to provide 10 clocks to various clock domains in the HAPS FPGA for high speed data input [89]. The reasons having ten clocks instead of one are (1) the way that the FPGA implements regional clocks requires different clock inputs to clock different I/O banks that are receiving the signals, and (2) it is easier to maintain alignment of 12–13 signals with one clock than 128 signals with one clock. Figure 71 shows the architecture for TIPA-3100 interposer.

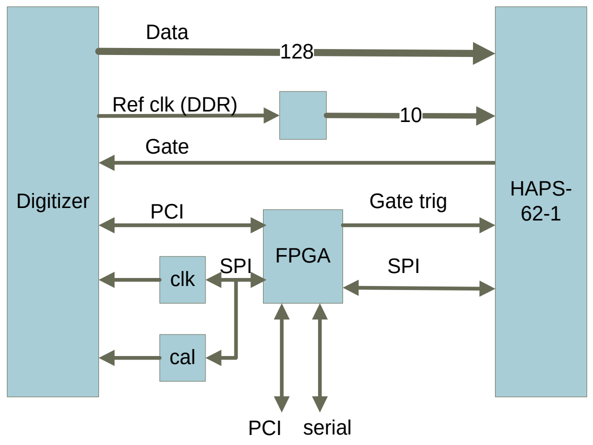


Figure 71 TIPA-3100 architecture

C. HAPS-62-1 FPGA

The HAPS-62-1 has two Xilinx Virtex-6 FPGAs (P/N XC6VLX760-1FF1760C). The resource for each XC6VLX760-1 is listed in Table 88.

XC6VLX760-1		Amount
Logic cells		758,784
CLBs	Slices	118,560
	Max Distributed RAM (Kb)	8,080
Block RAM blocks	18 Kb	1,440
	36 Kb	720
	Max (Kb)	25,920
MMCMs (450MHz)		18
DSP48E1 slices (450MHz)		864
I/O		1,200
GTX transceivers (Gb/s)		5
Speed (MHz)		450

Table 88. XC6VLX760-1 resource

Configurable logic blocks (CLBs) are the main logic resources for implementing digital sequential as well as combinatorial circuits. The mixed-mode clock manager (MMCM) is used to generate multiple clocks with defined phase and frequency relationships to a given input clock. GTX stands for gigabit transceiver.

D. TIPD-3200 DAC INTERPOSER

The DAC interposer passes unimpeded 320-bit data for maximum data integrity. A double data rate (DDR) reference clock from the digital-to-analog converter is buffered and distributed to provide six clocks to the FPGA for output of high speed data [89]. Figure 72 shows the architecture for TIPD-3200 interposer.

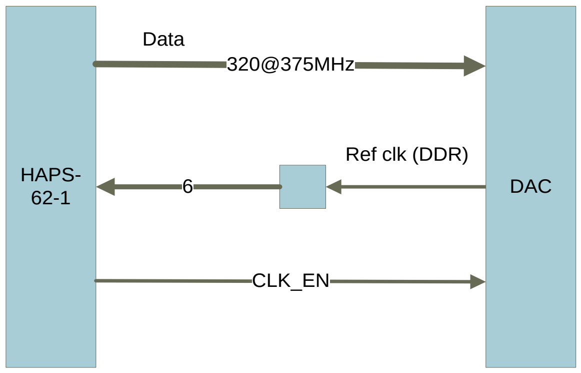


Figure 72 TIPD-3200 architecture

E. TDAC-2000 DAC

The TDAC-2000 is a single-channel waveform generation engine comprised of multiplexers and a digital-to-analog converter operating at 12 GS/s as shown in Figure 73.

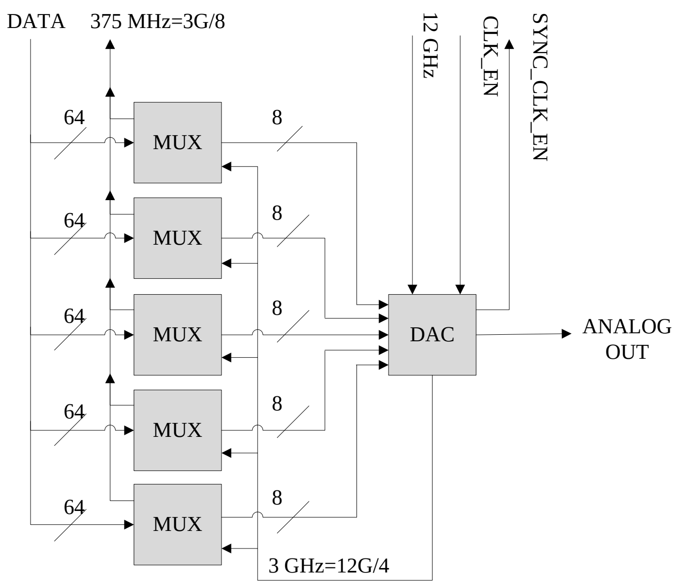


Figure 73 TDAC-2000 architecture

Data is supplied to the multiplexers via 320 data lines at 375 Mb/s using clocks generated by the digital-to-analog converter and multiplexers from a 12 GHz input clock.

MUX 1 (divided by 8): 320 to 40 channels

Input:	3 Gb/s ÷ 8 = 375 Mb/s
	5 × 64-bit@375 Mb/s = 120 Gb/s
Output:	5 × 8-bit@3 Gb/s = 120 Gb/s

MUX 2 (divided by 4, inside digital-to-analog converter): 40 to 10 channels

Input:	5 × 8-bit@3 Gb/s = 120 Gb/s
Output:	5 × 2-bit@12 Gb/s = 120 Gb/s

CLK_EN is used to start and stop of the analog waveform output. SYNC_CLK_EN is used for multiple DACs alignment [90]. When input data is processed in the FPGA, the output data might have fractions. This is why the digital-to-analog converter sampling rate (120 Gb/s) is higher than analog-to-digital converter sampling rate (96 Gb/s).

APPENDIX C. POLYPHASE DFT FILTER BANKS EXAMPLES

A. POLYPHASE DFT FILTER BANKS EXAMPLE 1

The program in Table 89, written in MATLAB, demonstrates how to cause an FPGA to divide an input signal into 32 subbands using polyphase DFT analysis filter banks.

```
% M = number of channels (subbands), N = number of taps in each polyphase FIR filter
M = 32;
N = 8;
b = fir1(M*N-2,1/M);                         % find coefficients for FIR filter
b = [b,zeros(1,M*N-length(b))];
% re-arrange coefficients to polyphase format
B = flipud(reshape(b,M,N));
Hq = cell(M,1);
% create polyphase FIR filters
for k=1:M
        Hq{k} = dfilt.dffir(B(k,:));
end
F = qfft(‘length’,M,’scale’,0.5*ones(1,log2(M)));     % set FFT length
g = 1/prod(F.ScaleValues);
% number of frequencies to sweep
% sweep from 0 to pi
Nfreq = 200;
w = linspace(0,pi,Nfreq);
P = 100;
t = 1:M*N*P;
HH = zeros(M,length(w));
for j=1:length(w)
         x = sin(w(j)*t);                    % input signal
        X = [x(:);zeros(M*ceil(length(x)/M)-length(x), 1)];
% re-arrange input to polyphase format
        X = reshape(X,M,length(X)/M);
        Y = zeros(size(X));
% create FIR filter bank
```

```
		for k=1:M
			Y(k,:) = filter(Hq{k},X(k,:));
		end
		Y = fft(F,Y);				% create subbands
		HH(:,j) = var(Y.')';			% store output power
end
s = 1/prod(scalevalues(F));
HH = HH*s^2;
% plot output power
figure(1)
plot(w,10*log10(HH))
title('Filter Bank Frequency Response')
xlabel('Frequency (normalized to channel center)')
ylabel('Magnitude Response (dB)')
set(gca,'xtick',(1:M/2)*w(end)/M*2)
set(gca, 'xticklabel',(1:M/2))
```

Table 89. MATLAB program: polyphase DFT analysis filter banks

B. POLYPHASE DFT FILTER BANKS EXAMPLE 2

The program in Table 90 demonstrates how to cause an FPGA to divide and reconstruct input signals using polyphase DFT filter analysis and synthesis filter banks.

```
M=32;				%Number of channels, (decimation factor)
r=8;				%number of taps in each sub filter
N=r*M-1;			%order of the prototype filter
H=fir1(N, 1/M);			%FIR filter
%reshape the filter in matrix form (decomposition, filter bank generation)
hh=reshape(H,M,length(H)/M);
 y=[];
X=y;
y1=y;
zi=zeros(M, r-1);
zi1=zi;
x=wavread('test.wav');		%reading input signal
xx=reshape(x,M,length(x)/M);	%reshape input sigal into matrix form
```

```
yy=zeros(size(xx));                        %set matrix yy same size as xx
%Analysis filter bank, filtering the parallel channel data in xx with filter bank hh
for k=1:M
        [yy(k,:),zi(k,:)]=filter(hh(k,:),1, xx(k,:), zi(k,:));
end
yy=ifft(yy);
 %process subband signals here
 yy=fft(yy);
 %Synthesis filter bank processing
for m=1:M
        [yy(m,:),zi1(m,:)]=filter(hh(M+1-m,:),1,yy(m,:),zi1(m,:));
End
 %restore the output signal in a vector form and multiple the constant to overcome the loss
y=M*M*reshape(yy,1,length(x));
delay=length(H)-M+1;              %Processing delay for the filter bank
%reorder the output signal to overcome the processing delay
y=[y(delay:end) y(1:delay-1)];
%calculate the difference between input and output signal
dif=x-y';
 %plot input, output and magnitude difference
figure
subplot(311)
plot(1:length(x), real(x))
axis([0 length(x) -.4 .4])
title('Input signal')
subplot(312)
plot(1:length(y), real(y))
axis([0 length(y) -.4 .4])
title('Output signal')
```

Table 90.   MATLAB program: polyphase DFT analysis and synthesis filter banks

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. BACKGROUND KNOWLEDGE

A. SETUP TIME AND HOLD TIME REQUIREMENTS

A flip-flop is a circuit that has two stable output states (0 and 1) and can be used as a memory device to store information. The output states of a flip-flop can be changed by signals applied to one or more inputs. An edge triggered flip-flop is set (to state 1) or reset (to state 0) by inputs and a clock signal during the low-to-high or high-to-low transition of a clock pulse. Edge triggered flip-flops are the most important building blocks in a reconfigurable computing.

The data and clock signals must be synchronized so that when a clock triggers a flip-flop, it reads correct data at the input. This synchronization-relationship is guaranteed by the source device; however, when data and source clock are forwarded to a destination device with a propagation delay, data and clock may no longer be in synchronization (alignment). This problem is especially prominent for data at an ultra-high rate, since the workable data window is very narrow.

B. INHERENT TIMING WINDOW

Every flip-flop has restrictive time regions around the active clock edge in which input should not change. The setup time is the interval before the clock where the data must be held stable. The hold time is the interval after the clock where the data must be held stable. To satisfy setup time and hold time requirements for a flip-flop, a clock path (the trace from a clock pin to the clock port of a flip-flop) must have a longer propagation time delay (not data rate) than a data path (the trace from an input pin to the data port of a flip-flop), so that data will arrive before the clock sampling edge.

The clock path delay must be longer than the data path delay to ensure correct data sampling as explained earlier; so mathematically, the least amount of time that the clock can be behind data is “the minimum clock path delay–the maximum data path delay.” Similarly, the most amount of time that the clock can be behind data is “the maximum clock path delay–the minimum data path delay.” We define δ(least) as the least amount of time which data is ahead of the clock; and δ(most) as the most amount of time which data is ahead of clock. We can calculate δ(least) and δ(most) in the following equations.

$$\delta(\text{least}) = \text{Min clock path delay} - \text{Max data path delay}$$

$$\delta(\text{most}) = \text{Max clock path delay} - \text{Min data path delay}$$

The inherent timing window is defined as the time period between δ(least) and δ(most), expressed as (δ(least), δ(most)). The clock sampling edge is guaranteed to arrive within the inherent timing window after data is arrived.

C. DEMULTIPLEXER

To reduce the data rate from a faster device to a slower device, we need to use a demultiplexer. A demultiplexer is a device that takes a single input signal and selects one of many data-output lines connected to a single input [91]. Figure 74 shows an example of a one-to-four demultiplexer.

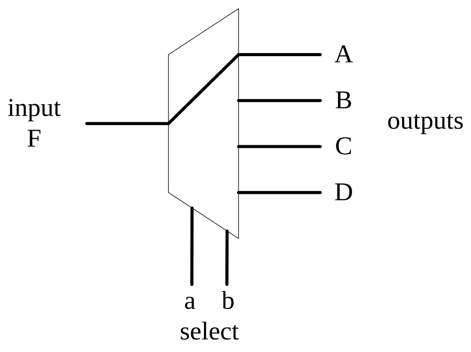


Figure 74 A one-to-four demultiplexer

We can express this one-to-four demultiplexer in Boolean equations as below.

$$A = \left(F \bullet \bar{a} \bullet \bar{b}\right)$$

$$B = \left(F \bullet a \bullet \bar{b}\right)$$

$$C = (F \bullet \bar{a} \bullet b)$$

$$D = (F \bullet a \bullet b)$$

D. MULTIPLEXER

A multiplexer is a device that selects one of several analog or digital input signals and forwards the selected input into a single line [91]. A multiplexer of 2^n inputs has n select lines, which are used to select which input line to send to the output [92].Typically the a and b inputs are cycle regularly through the space of all possible bits. Figure 75 shows an example of a four-to-one multiplexer.

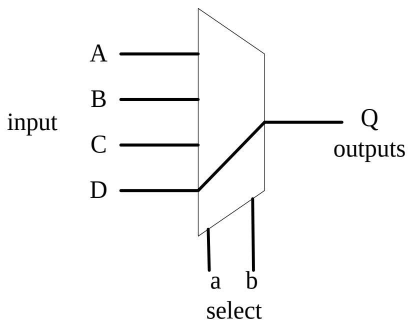


Figure 75 A four-to-one multiplexer

We can express this four-to-one multiplexer as a Boolean equation:

$$Q = \left(A \bullet \bar{a} \bullet \bar{b}\right) + \left(B \bullet a \bullet \bar{b}\right) + (C \bullet \bar{a} \bullet b) + (D \bullet a \bullet b)$$

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E. DERIVATION OF POLYPHASE DFT FILTER BANKS

In this Appendix, we derive the equation of DFT analysis filter banks by expressing filter banks in z-domain and then applying polyphase decomposition equation to these filter banks. This proof was developed by Professor Cristi at Naval Postgraduate School, Monterey, California [93] and the author.

A. POLYPHASE DFT FILTER BANKS REPRESENTATION IN Z-DOMAIN

In frequency domain, filter banks are expressed as a set of filters with frequency responses derived from a prototype filter $H(\omega)$ as:

$H_k(\omega) = H(\omega - k\frac{2\pi}{M})$, k = 0,…,M-1, M≥2 and k is an integer, 2π/M is spacing

In z domain, the transfer functions are expressed as:

$$H_k(z) = H(e^{-jk\frac{2\pi}{M}}z) \tag{1}$$

Proof:

A single filter in frequency domain H(ω) is expressed as:

$$H(\omega) = DFT\{h[n]\} = \sum_{n=0}^{N-1} h[n]e^{-j\omega n} \tag{2}$$

Figure 76 shows a single filter in frequency domain.

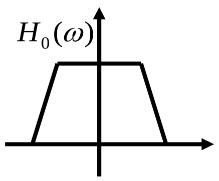


Figure 76 A single filter in frequency domain

For a bank of M filters spaced at 2π/M, the filter banks are expressed as:

$H_k(\omega) = H(\omega - k\frac{2\pi}{M})$, k = 0, 1… M-1, M≥1 and k is an integer

Figure 77 shows a filter bank of M filters spaced at 2π/M.

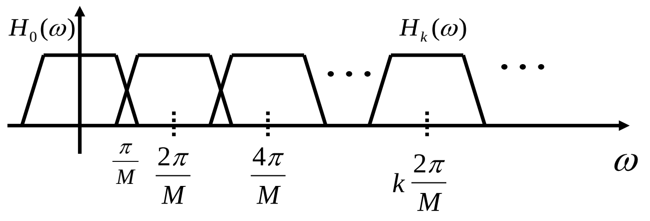


Figure 77 A filter bank of M filters spaced at 2π/M in the frequency domain

(A) Express M filters by substituting H(ω) with $H_k(\omega)$, and h[n] with $h_k[n]$ in (2).

$$H_k(\omega) = \sum_{n=0}^{N-1} h_k[n]e^{-j\omega n} \tag{3}$$

(B) Express M filters by substituting ω with $(\omega - k\frac{2\pi}{M})$ in (2).

$$H(\omega - k\frac{2\pi}{M}) = \sum_{n=0}^{N-1} h[n]e^{-j(\omega-k\frac{2\pi}{M})n} = \sum_{n=0}^{N-1} h[n]e^{jk\frac{2\pi}{M}n}e^{-j\omega n} \tag{4}$$

Since $H_k(\omega) = H(\omega - k\frac{2\pi}{M})$, (3) = (4).

$$\sum_{n=0}^{N-1}\{h_k[n]\}e^{-j\omega n} = \sum_{n=0}^{N-1}\{h[n]e^{jk\frac{2\pi}{M}n}\}e^{-j\omega n}$$

$$h_k[n] = h[n]e^{jk\frac{2\pi}{M}n} \tag{5}$$

Transfer $h_k[n]$ to z domain.

$$H_k(z) = \sum_{n=0}^{N-1} h_k[n]z^{-n}$$

$$= \sum_{n}^{N-1} h[n]e^{jk\frac{2\pi}{M}n}z^{-n}, \text{ from (5)}$$

$$= \sum_{n=0}^{N-1} h[n]e^{-jk\frac{2\pi}{M}(-n)}z^{-n}$$

$$= \sum_{n=0}^{N-1} h[n](e^{-jk\frac{2\pi}{M}}z)^{-n}$$

$$= \sum_{n=0}^{N-1} h[n]z'^{-n} = H_k(z')$$

$$z' = (e^{-jk\frac{2\pi}{M}})(z)$$

$$H_k(z) = H_k(z') = H(e^{-jk\frac{2\pi}{M}}z) \square$$

We summarize the filter banks relationship in Table 91. The basic technique in proving equation (1) is converting filter banks expression in frequency domain to time domain, and then from time domain to z domain.

Domain	Filter Banks Expression	Transform	Prototype
Frequency	$H_k(\omega) = H(\omega - k\frac{2\pi}{M})$	$H(\omega) = DFT\{h[n]\} = \sum_{n=0}^{N-1} h[n]e^{-j\omega n}$	H(ω)
Time	$h_k[n] = h[n]e^{jk\frac{2\pi}{M}n}$	Intermediate step to bridge frequency and z domains	h[n]
Z	$H_k(z) = H(e^{-jk\frac{2\pi}{M}}z)$	$H_k(z) = Z\{h_k[n]\} = \sum_{n=0}^{N-1} h_k[n]z^{-n}$	H(z)

Table 91. Filter banks expressions

B. POLYPHASE DECOMPOSITION

We decompose the input signal sequence x[n] into its periodically interleaved subsequences in z-domain by using general polyphase decomposition equation below:

$$X(z) = \sum_{n=-\infty}^{\infty} x[n]\, z^{-n}$$

We also decompose the finite impulse response filter into polyphase components as below.

$$\mathrm{H(z)} = \sum_{n=-\infty}^{\infty} \mathrm{h[n]z^{-n}}$$

$$= \sum_{p=0}^{M-1} z^{-p} H_p(z^M) \tag{6}$$

$$= H_0(z^M) + z^{-1}H_1(z^M) + \cdots + z^{-(M-1)}H_{M-1}(z^M)$$

$$H_p(z^M) = \sum_{n=-\infty}^{+\infty} h[nM + p]z^{-Mn} \tag{7}$$

C. ANALYSIS POLYPHASE DFT FILTER BANKS WITH M FILTERS

The polyphase DFT filter banks with M filters are expressed as polyphase components:

$$H_k(z) = \sum_{p=0}^{M-1} z^p w_M^{kp}\, E_{-p}(z^M) \tag{8}$$

$$w_M = e^{-j\frac{2\pi}{M}} \tag{9}$$

Proof:

$$H(z) = \sum_{p=0}^{M-1} z^{-p} H_l(z^M) \text{ , from (6)}$$

$$H(z) = \sum_{p=0}^{M-1} z^{p}\, E_{-p}(z^M)\text{, replace } H_p \text{ with } E_p\text{, and p with –p} \tag{10}$$

From (1) and (9), we replace z with $w_M^k z$ in (10).

$$H_k(z) = H(e^{-jk\frac{2\pi}{M}} z) \; = H(e^{(-j\frac{2\pi}{M})k} z) = H\left(w_M^k z\right)$$

$$= \sum_{p=0}^{M-1} z^p w_M^{kp}\, E_{pl}(z^M w_M^{kM}) \; = \sum_{p=0}^{M-1} z^p w_M^{kp}\, E_{-p}(z^M)\square$$

$$w_M^{kM} = e^{-j\frac{2\pi kM}{M}} = e^{-j2\pi k} = 1$$

We express polyphase DFT filter banks (8) in a matrix form.

$$\begin{bmatrix} H_0(z) \\ H_1(z) \\ \vdots \\ H_{M-1}(z) \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & w_M & \cdots & w_M^{M-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & w_M^{M-1} & \ldots & w_M^{(M-1)^2} \end{bmatrix} \begin{bmatrix} E_0(z^M) \\ zE_1(z^M) \\ \vdots \\ z^{M-1}E_{M-1}(z^M) \end{bmatrix} \tag{11}$$

M-point DFT, N=M

$$X[k] = \mathrm{DFT}\{x[n]\} = \sum_{n=0}^{M-1} x[n] e^{-j(\frac{2\pi}{M}k)n}$$

$$X[0] = \sum_{n=0}^{M-1} x[n] = x[0] + x[1] + \cdots + x[M-1]$$

$$X[1] = \sum_{n=0}^{M-1} x[n] e^{-j\left(\frac{2\pi}{M}\right)n} = x[0] + x[1]e^{j\left(\frac{2\pi}{M}\right)} + \cdots + \; x[M-1]e^{j\left(\frac{2\pi(M-1)}{M}\right)}$$

$$= x[0] + x[1]w_M + \cdots + x[M-1]w_M^{M-1}$$

$$X[M-1] = \sum_{n=0}^{M-1} x[n] e^{-j\left(\frac{2\pi}{M}(M-1)\right)n}$$

$$= x[0] + x[1]e^{j\left(\frac{2\pi(M-1)}{M}\right)} + \cdots + x[M-1]e^{j\left(\frac{2\pi(M-1)^2}{M}\right)}$$

$$= x[0] + x[1]w_M^{M-1} + \cdots + x[M-1]w_M^{(M-1)^2}$$

We express the matrix in equation (11) graphically as shown in Figure 78.

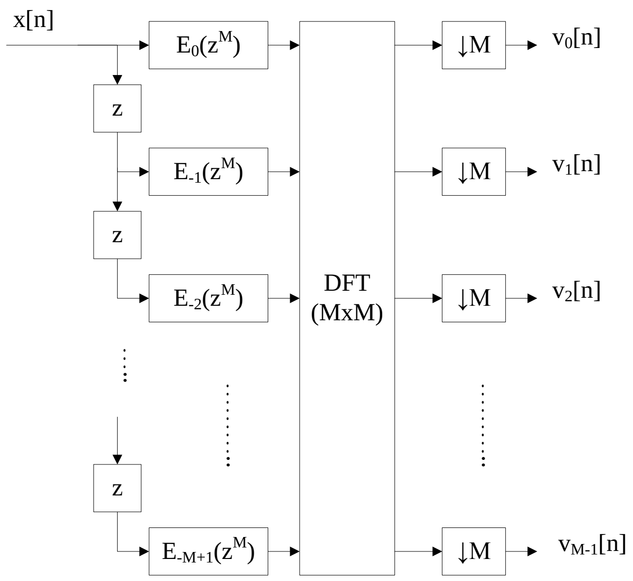


Figure 78 M-filter polyphase representation

Each output from DFT is a polyphase component with a non-zero value only at every Mth term. We can down-sample by M without losing information. In addition, since $E_{-p}(z^M)$ is polyphase, we apply the Noble Identity [75] to Figure 78 and obtain Figure 79.

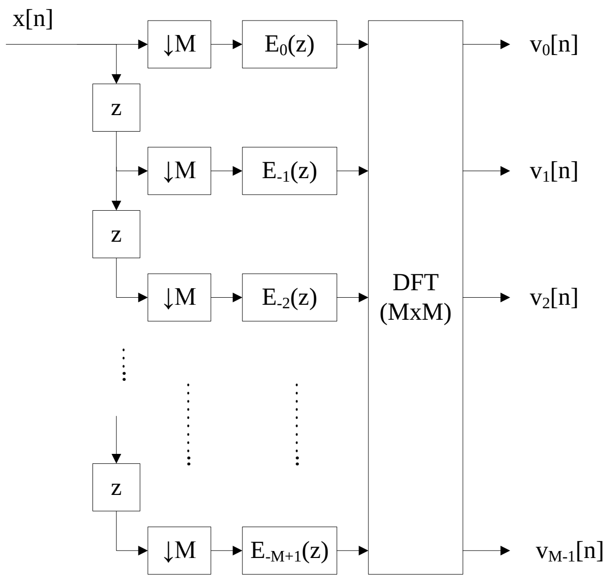


Figure 79 Apply Noble identity to polyphase representation

D. SYNTHESIS POLYPHASE DFT FILTER BANKS WITH M CHANNELS

The analysis polyphase DFT filter banks network in Figure 79 are used to separate a wide-bandwidth serial input signal into M parallel subbands so that they can be processed by digital signal processing at lower sampling rates. Once all subband signals are processed, we have a nearly perfect reconstruction by applying the same principle for synthesis network as shown in Figure 80. There is a multiplying factor M after IDFT to compensate for 1/M in IDFT.

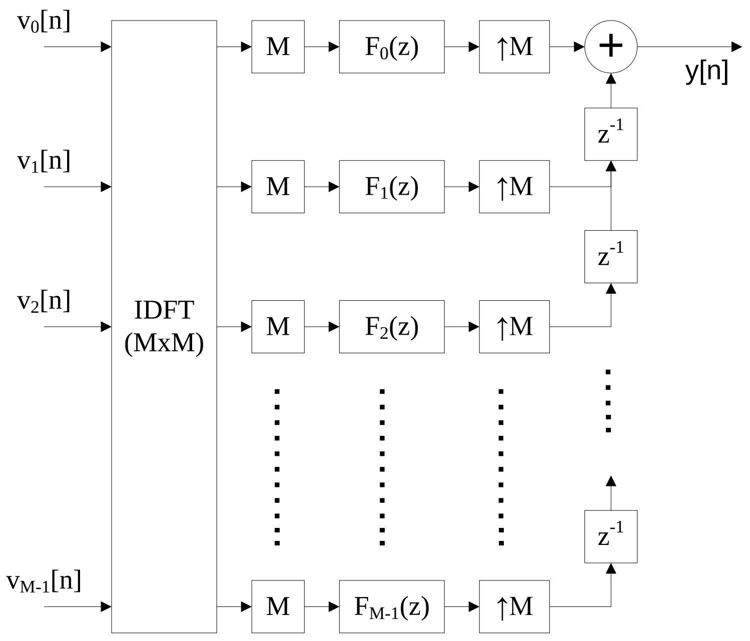


Figure 80 Synthesis network

LIST OF REFERENCES

[1] W. H. Wolf, “Hardware-Software Codesign of Embedded Systems,” *Proc. IEEE* vol. 82, no. 7, Jul. 1994, pp. 967–989.

[2] Codesign Group, U.C. Berkeley, “A framework for hardware-software co-design of embedded systems.” [Online]. http://embedded.eecs.berkeley.edu/research/hsc/. Accessed Apr. 16, 2013.

[3] J. A. Stankovic, “Real-time and embedded systems,” *Proc. ACM Computing Surveys* vol. 28, no.1, pp. 205–208, Mar. 1996.

[4] T. Smith, “Electronic Warfare (EW) Principles and Concepts,” class lecture, Naval Postgraduate School, Monterey, CA, Jun. 2010.

[5] S. Edwards, “Microprocessors or FPGAs?: Making the right choice,” *RTC Magazin*e, Feb. 2011.

[6] J. Krasner, “Model-based design and beyond: solutions for today’s embedded systems requirements,” Embedded Market Forecasters, American Technology International, Los Angeles, CA, Jan. 2004.

[7] W. Forster et al., “Automated generation of explicit connectors for component based hardware/software interaction in embedded real-time systems,” in *2008 IEEE International Symposium on Parallel and Distributed Processing*, Miami, FL, Apr. 2008, pp. 1–8.

[8] R. F. Paige et al., “Revealing complexity through domain-specific modelling and analysis,” in *Development, Operation and Management of Large-Scale Complex IT systems Monterey workshop*, Monterey, CA, Mar. 2012, pp. 251–265.

[9] I. Sommerville, “Integrated requirements engineering: A tutorial,” *Proc. Software, IEEE*, pp. 16–23, 2005.

[10] J. Cleland-Huang, “Software requirements,” Software Engineering, School of Computing, DePaul University. [Online]. http://www.google.com/url?sa=t&rct=j&q=&esrc=s&frm=1&source=web&cd=1&cad=rja&ved=0CCsQFjAA&url=http%3A%2F%2Fwww.researchgate.net%2Fpublication%2F228381037_Software_Requirements%2Ffile%2F3deec51dc556b2c082.pdf&ei=4T_YUrqpFdWwoQSz9oHYBw&usg=AFQjCNFKhqmbQU64GLTMwZ7KjgiHIWrgww&sig2=mulRZEX2rJMNu6Qi4k0NoQ. Accessed May 20, 2013.

[11] D. Leffingwell and D. Widrig, *Managing Software Requirements*, 2nd ed. Boston: Addison-Wesley, 2003, pp. 165–172.

[12] C.J. Neill, "Requirements engineering: the state of the practice," *Proc. Software, IEEE Journal*, vol. 20, no. 6, pp. 40–45, Nov./Dec. 2003.

[13] F. P. Brooks, Jr., *The Mythical Man-Month* [anniversary ed]. Boston: Addison-Wesley, 1995, p. 20.

[14] W. W. Royce, "Managing the development of large software systems: Concepts and techniques," in *WESCON Technical Papers*, Los Angeles, CA, August, 1970. Reprinted in *Proc. of the Ninth International Conference on Software Engineering*, Monterey, CA, Mar. 1987, pp. 328–338.

[15] B. W. Boehm, "A spiral model of software development and enhancement," *Proc. IEEE Computer*, vol. 21, no. 5, pp. 61–72, May, 1998.

[16] P. Kruchten, "Architectural blueprints—The '4+1' view model of software architecture," *Proc. IEEE Software*, vol. 12, no. 6, pp. 42–50, Nov. 1995.

[17] P. Kruchten, "Tutorial: introduction to the rational unified process®," in *The 24th International Conference on Software Engineering*, Orlando, FL, May 2002, pp. 703–703.

[18] G. Booch, *Object-Oriented Design with Application.* Redwood: The Benjamin/Cummings Publishing Company, Inc., 1991.

[19] A. Cockburn, *Agile Software Development: The Cooperative Game*, 2nd ed. Boston: Pearson Education, 2007.

[20] R. C. Martin and G. Melnik, "Tests and requirements, requirements and tests: A Möebius strip," *Proc. IEEE Software*, vol. 25, no. 1, pp. 54–59, Jan./Feb. 2008.

[21] "Manifesto for Agile software development," (2001), Agile Alliance, http://agilemanifesto.org/. Accessed Apr. 16, 2013.

[22] D. Harel, "Statecharts: A visual formalism for complex systems," *Proc. Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, Jun. 1987.

[23] B. Berenbach et al., *Software & Systems Requirements Engineering in Practice.* New York: McGraw Hill, 2009, pp. 20–16, 73–124.

[24] B. Hailpern and P. Tarr, "Model-driven development: The good, the bad, and the ugly," *Proc. IBM Systems Journal*, vol. 45, no. 3, pp. 451–461, 2006.

[25] J. Tolvanen and S. Kelly, "Integrating models with domain-specific modeling languages," in *the 10th Workshop on Domain-Specific Modeling*, article no. 10, Reno/Tahoe, NV, Oct. 2010.

[26] G. Booch et al., "An MDA manifesto," in *MDA Journal*, May 2004.

[27] R. W. Selby, Ed., *Software Engineering, Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research.* Hoboken, NJ: John Wiley & Sons, 2007.

[28] National Instruments, "Shortening the embedded design cycle with model-based design," National Instruments, Austin, TX, 2012.

[29] G. D. Micheli and R. K. Gupta, "Hardware/software codesign," *Proc. IEEE*, vol. 85, no. 3, pp. 349–365, Mar. 1997.

[30] F. Balarin, *Hardware-Software Codesign of Embedded Systems, The POLIS approach.* MA: Kluwer Academic Publishers, 1997.

[31] F. Vahid, "What is hardware/software partitioning?" *ACM SIGDA Newsletter*, vol. 39, no. 6, p. 1-1, Jun. 2009.

[32] L. P. Carloni et al., "Platform-based design for embedded systems," in *Embedded Systems Handbook*. Boca Raton, FL: CRC Press, 2005.

[33] K. Keutzer et al., "System level design: Orthogonolization of concerns and platform-based design,'' *Proc. IEEE Trans. Computer-Aided Design Integrated Circuits Syst*., vol. 19, no. 12, pp. 1523–1543, Dec. 2000.

[34] A. Sangiovanni-Vincentelli et al., "Benefits and challenges for platform-based design," *Proc. Design Automation Conference (DAC)*, San Diego, CA, Jun. 2004, pp. 409–414.

[35] A. Sangiovanni Vincentelli, "Quo Vadis SLD: Reasoning about the trends and challenges of system level design," *Proc. IEEE*, vol. 95, no. 3, pp. 467–506, Mar. 2007.

[36] A. Pinto, "Metropolis design guidelines," Codesign Group, U.C. Berkeley, Berkeley, CA, Nov., 2004.

[37] J. Teich and C. Haubelt, *Digitale Hardware/Software-Systeme: Synthese und Optimierung*, 2nd ed. Berlin, Germany: Springer-Verlag, 2007.

[38] J. Teich, "Embedded system synthesis and optimization," in *Workshop Syst. Des. Autom*. Rathen, Germany, Mar. 2000, pp. 9–22.

[39] J. Teich, “Hardware/software codesign: The past, the present, and predicting the future,” *Proc. IEEE*, vol. 100, pp. 1411–1430, May, 2012.

[40] K. Vissers, “Programming models and architectures for FPGA platforms,” in *the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Washington DC, Sep. 2004, pp. 1–1.

[41] L. Cardelli and P. Wegner, “On understanding types, data abstraction, and polymorphism,” *Proc. ACM Computing Surveys*, vol.17, no. 4, pp. 471–523, Dec. 1985.

[42] Xilinx Inc., *ISE Design Suite Overview Xilinx ISE Help (v 13.1)* M*anual*. San Jose, CA: Xilinx Inc., 2011.

[43] Altera Inc., *Altera VHDL Basics*. Altera Inc., San Jose, CA, 2013.

[44] A. A. Jerraya et al., *Behavioral Synthesis and Component Reuse With VHDL, Norwell*. Boston, MA: Kluwer, 1998.

[45] T. Riesgo et al., “Design methodologies based on hardware description languages,” *Proc. IEEE Trans. Ind. Electron.*, vol. 46, no. 1, pp. 3–12, Feb. 1999.

[46] E. Monmasson and M. N. Cirstea, “FPGA design methodology for industrial control systems—A review,” *Proc. IEEE Transactions on Industrial Electronics*, vol. 54, no.4, pp. 1824-1842, Aug. 2007.

[47] Xilinx Inc., “FPGA design flow overview,” Xilinx Inc., San Jose, CA, 2009.

[48] E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley Professional Computing Series, 1995.

[49] EventHelix, “Embedded system design patterns.” [Online]. http://www.eventhelix.com/realtimemantra/patterns/#.UgT31NJJ4rw. Accessed Apr. 23, 2013.

[50] A. DeHon et al., “Design patterns for reconfigurable-computing,” in *IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM 2004), Napa Valley, CA, Apr. 20–23, 2004.

[51] V. Meeldijk, *Electronic Components Selection and Application Guidelines*. New York: John Wiley & Sons, 1995.

[52] The American Society of Mechanical Engineers, “Dimensioning and tolerancing, Y14.5, an American National Standard,” 2009, pp. 224.

[53] “Directive 2002/95/EC of the European parliament and of the council of 27 January 2003,” *Official Journal of the European Union*, Feb. 2003.

[54] N. J. Nilsson, *Problem-solving Methods in Artificial Intelligence.* New York: McGraw-Hill Book Company, 1971.

[55] R. G. Lyons, *Understanding Digital Signal Processing*. Upper Saddle River, NJ: Prentice Hall PRT, 2001, pp. 23–29.

[56] J. Kriegbaum, “FPGA’s vs. ASIC’s,” in *EE Times*, Sep. 2004.

[57] Texas Instruments Inc., “TMS320C6678 multicore fixed and floating-point digital signal processor datasheets,” Texas Instruments Inc., Dallas, TX, May 2013.

[58] G. Blake et al., “A survey of multicore processors,” in *IEEE Signal Processing Magazine,* Nov. 2009.

[59] Xilinx Inc., Virtex-6 FPGA data sheet: DC and switching characteristics DS152 (v3.5), Xilinx Inc., San Jose, CA, May 17, 2013.

[60] R. Rinker et al., “An automated process for compiling dataflow graphs into reconfigurable hardware,” *Proc. IEEE Transactions on VLSI Systems,* vol. 9, no. 1, pp. 130–139, Feb. 2001.

[61] E. A. Lee and D. G. Messerschmitt, “Synchronous data flow,” *Proc. IEEE,* vol. 75, no. 9, pp. 1235–1245, Sep. 1987.

[62] W. D. Hillis and G. L. Steele, “Data parallel algorithms,” *Proc. Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, Dec. 1986.

[63] M. Gokhale and R. Minnich, “FPGA Computing in a Data Parallel C,” in *IEEE workshop on FPGAs for Custom Computing Machines (FCCM)*, Napa Valley, CA, Apr. 1993, pp. 94–101.

[64] S. Guccione and M. Gonzalez, “A data-parallel programming model for reconfigurable architectures,” in *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, Napa Valley, CA, Apr. 1993, pp. 79–87.

[65] V. M. Bove, Jr. and J. A. Watlington, “Cheops: A reconfigurable data-flow system for video processing,” *Proc. IEEE Transactions on Circuits and Systems for Video Technology*, vol. 5, no. 2, pp. 140–149, Apr. 1995.

[66] E. Caspi et al., “Stream computations organized for reconfigurable execution (SCORE): Introduction and tutorial,” UC Berkeley BRASS Group, Berkeley, CA, Aug. 2000.

[67] T. Callahan, J. Hauser and J. Wawrzynek, “The Garp architecture and C compiler,” *Proc. IEEE Computer*, vol. 33, no. 4, pp. 62–69, Apr. 2000.

[68] C. L. Seitz, “The Cosmic Cube,” in *Communications of the ACM —Special section on computer architecture (CACM)*, vol. 28, no. 1, pp. 22–33 Jan. 1985.

[69] T. V. Eicken *et al.*, “Active messages: A mechanism for integrated communication and computation,” in *The 19th Annual Symposium on Computer Architecture*, Queensland, Australia, May 1992, pp. 430–440.

[70] M. Snir and W. Gropp, *MPI: The Complete Reference*, 2nd ed. Cambridge: MIT Press, 1998.

[71] M. Jones et al., “Implementing an API for distributed adaptive computing systems,” in *Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa Valley, CA, Apr. 1999, pp. 222.

[72] C. Seitz, “System timing,” in *Introduction to VLSI Systems.* Boston*:* Addison-Wesley, 1980.

[73] B. Smith, “Architecture and Applications of the HEP Multiprocessor Computer System,” *Proc. the International Society for Optical Engineering*, pp. 241–248, 1982.

[74] Arvind et al., “I-structures: Data structures for parallel computing,” in *the Workshop on Graph Reduction (Springer-Verlag Lecture Notes in Computer Science 279).* Santa Fe, New Mexico, Sep. 1986.

[75] P.P. Vaidyanathan, *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[76] D. Zhou, “A review of polyphase filter banks and their application,” Technical report, Air Force Research Laboratory, Information Directorate, Rome Research Site, Rome, New York, Sep. 2006.

[77] P. Schniter, “Computational savings of polyphase/DFT filterbanks.” [Online]. http://cnx.org/content/m10930/2.3/. Accessed Jun.1, 2013.

[78] Lowegian International, “FIR filter properties.” [Online]. http://www.dspguru.com/dsp/faqs/fir/properties. Accessed Sep. 20, 2013.

[79] Xilinx Inc., “16-Channel, DDR LVDS interface with per-channel alignment, application note: Virtex-5 FPGAs, XAPP855 (v1.0), “Xilinx Inc., San Jose, CA, Oct. 2006.

[80] A. Ralston and E. Reilly, "Baudot code," *Encyclopedia of Computer Science*, 3rd ed.. New York: IEEE Press/Van Nostrand Reinhold, 1993.

[81] A. Kennelly, "Biographical Memoir of George Owen Squier 1865–1934," in *Biographical Memoirs vol. XX-4th Memoir, National Academy of Sciences of the United States of America*, annual meeting, 1938.

[82] Xilinx Inc., "ISE in-depth tutorial UG695 (v14.1)," Xilinx Inc., San Jose, CA, Apr. 2012.

[83] T. Kihm, "Digital receiver (DSP) requirements," Airborne Interceptor Research Laboratory, Naval Air Warfare Center Weapons Division, Point Mugu, CA, Apr. 2013.

[84] Pentek Inc., "4-channel 200 MHz A/D with DDCs and Virtex-6 FPGA—x8 PCIe datasheet," Pentek Inc., Upper Saddle River, NJ, May 2013.

[85] T. Kihm, "FY13 AI Project Interim Report, "Airborne Interceptor Research Laboratory, NAWCWD, Point Mugu, CA, Nov. 21, 2013.

[86] Pentek Inc., *Model 78661 4-Channel 200 MHz A/D with DDCs and Virtex-6 FPGA-x8 PCIe* [brochure]. Upper Saddle River, NJ: Pentek Inc., 2013.

[87] L. Rabiner and B. Juang, "An introduction to HMMs," in *IEEE ASSP Magazine*, vol. 3, pp. 4-16, Jan. 1986.

[88] M. Midzor, "EWP-200 Electronic Warfare Compatibility T&E Principles," presentation, Joint Electronic Warfare Effects Laboratory, Naval Warfare Center Weapon Division, Point Mugu CA, Jun. 2011.

[89] Tektronix Component Solutions, *HAPS Interpose Modules, TIPA-3100 and TIPD-3200 datasheets*. Tektronix Component Solutions, Beaverton, OR, Aug. 2011.

[90] Tektronix Component Solutions, "DCM-DAC Module Specification, version 2.0 (802-2914-00) datasheets," Tektronix Component Solutions, Beaverton, OR, Nov. 2010.

[91] T. Dean, *Network + Guide to Networks, Course Technology*, 5th ed. Boston, MA: Cengage Learning, 2010, pp. 82–85.

[92] D. Debashis, *Basic Electronic.* London, United Kingdom: Dorling Kindersley, 2010, pp. 557.

[93] R. Cristi, *Modern Digital Signal Processing*. Pacific Grove, CA: Thomson Brooks/Cole, 2004.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California