OULUN YLIOPISTO
UNIVERSITY of OULU

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Teemu Vikamaa

EMBEDDED PROGRAMMABLE SOFT-CORE CONTROL SUBSYSTEM FOR DIGITAL FRONT-END

Master's Thesis
Degree Programme in Computer Science and Engineering
May 2016

Vikamaa T. (2016) Embedded Programmable Soft-Core Control Subsystem for Digital Front-End. University of Oulu, Degree Programme in Computer Science and Engineering. Master's Thesis, 43 p.

ABSTRACT

Field-programmable gate arrays (FPGAs) provide a fully reconfigurable platform for soft-core processors. Such processors can be implemented using the general-purpose programmable logic of the FPGA, alleviating the need to incorporate hard-wired processors in embedded systems where high processing power is often not a primary design requirement.

An example of such embedded system is the digital radio-frequency front-end of a software-defined radio. A digital front-end (DFE) can be implemented as an FPGA-based system, with an embedded microcontroller executing various control functions. This kind of microcontroller subsystem is well suited for implementation using one or more soft processor cores.

The implementation part of this thesis presents a software-programmable soft-core subsystem for a digital radio-frequency front-end. The subsystem was built around a MicroBlaze soft processor core, and it was implemented on a Xilinx Virtex-7 FPGA. The design and implementation of the subsystem were carried out using Vivado design tools, and the associated software was developed in Xilinx SDK using the C programming language. The design was composed with the IP-centric, schematic-based design methodology in Vivado IP Integrator.

To verify its functionality, the subsystem was integrated into the full DFE design and tested on an ADS7-V1 evaluation board. The tests included latency measurements for automatic gain control, which is considered a soft real-time feature. However, no explicit latency constraint was defined for this particular implementation. The measurement results were compared against requirements given in the IEEE 802.11a/g specification.

Keywords: digital radio front-end, embedded system, FPGA, MicroBlaze, microcontroller, soft microprocessor, software-defined radio

Vikamaa T. (2016) Sulautettu ohjelmistopohjainen ohjausosa digitaaliselle RF-etupäälle. Oulun yliopisto, tietotekniikan tutkinto-ohjelma. Diplomityö, 43 s.

TIIVISTELMÄ

Ohjelmoitavat porttimatriisit (Field-programmable gate array, FPGA) muodostavat vapaasti muokattavan alustan ohjelmistopohjaisille mikroprosessoreille. Ohjelmistopohjaisiin mikroprosessoriytimiin pohjautuva järjestelmä voidaan toteuttaa käyttämällä FPGA:n ohjelmoitavaa logiikkarakennetta. Tällaiset toteutukset vähentävät tarvetta kovopohjaisille prosessoreille sulautetuissa järjestelmissä, joilta ei välttämättä vaadita erityisen suurta prosessointikapasiteettia.

Eräs esimerkki tällaisesta sulautetusta järjestelmästä on ohjelmistoradion digitaalinen radiotaajuusetupää. Digitaalinen etupää (Digital front-end, DFE) on mahdollista toteuttaa FPGA-pohjaisena järjestelmänä, johon sisällytetään ohjaustoimintoja suorittava sulautettu mikrokontrolleri. Tällainen mikrokontrollerimoduuli on helppo toteuttaa ohjelmistopohjaisen prosessoriytimen avulla.

Tämän työn käytännön osassa esitellään ohjelmistopohjainen ohjausmoduuli digitaalisen radion radiotaajuusosaan. Moduuli on rakennettu MicroBlaze-prosessoriytimen ympärille ja toteutettu Xilinxin Virtex-7 FPGA-piirillä. Moduulin suunnittelu ja synteesi tehtiin Vivado-suunnitteluohjelmistolla, ja prosessoriytimen ajama ohjelmakoodi toteutettiin C-ohjelmointikielellä Xilinx SDK –kehitysympäristössä. Moduulin suunnitteluun käytettiin Vivado IP Integratorin IP-keskistä, kaaviopohjaista suunnittelumenetelmää. IP:llä (Intellectual Property) tarkoitetaan erilaisia valmiita prosessoriytimiä, joita yhdistelemällä voidaan toteuttaa digitaalisia piirejä.

Ohjausmoduulin toiminta varmennettiin integroimalla se kokonaiseen DFE-järjestelmään, jota testattiin ADS7-V1-evaluointipiirilevyllä. Testeihin kuului automaattisen vahvistusohjauksen vasteaikamittauksia. Automaattista vahvistusohjausta voidaan pitää reaaliaikaisena toimintona, mutta sen toimintanopeus ei ollut tässä järjestelmässä olennaista. Mittausten tuloksia vertailtiin IEEE 802.11a/g-spesifikaatiossa annettuihin vaatimuksiin.

Avainsanat: digitaalinen RF-etupää, sulautettu järjestelmä, FPGA, MicroBlaze, mikrokontrolleri, ohjelmistopohjainen mikroprosessori, ohjelmistoradio

TABLE OF CONTENTS

ABSTRACT
TIIVISTELMÄ
TABLE OF CONTENTS
FOREWORD
ABBREVIATIONS
1. INTRODUCTION 9
    1.1. The Structure of This Document 9
2. DIGITAL RADIO 11
    2.1. The Radio Frequency Front-End 11
        2.1.1. Single-Conversion Superheterodyne 11
        2.1.2. Multiple-Conversion Superheterodyne 12
        2.1.3. Direct Conversion 12
    2.2. The Digital Front-End 12
    2.3. Software-Defined Radio 13
    2.4. The DFE Architecture Considered in This Thesis 14
    2.5. Related work 15
        2.5.1. WARP 15
        2.5.2. USRP 15
3. ARCHITECTURE OF EMBEDDED SYSTEMS AND MICROCONTROLLERS 17
    3.1. Introduction to Embedded Systems 17
    3.2. Microcontrollers 18
    3.3. Soft Processor Cores 18
    3.4. Abstraction Levels in Soft Processor Design 18
    3.5. Tools for IP-based Processor Design 19
    3.6. Hardware/Software Codesign and Coverification 20
    3.7. Field-Programmable Gate Arrays 21
        3.7.1. Architecture 21
        3.7.2. Implementation Technologies 23
4. IMPLEMENTATION 25
    4.1. The Hardware Environment 25
    4.2. Vivado Design Suite 25
        4.2.1. Block design 26
        4.2.2. Synthesis and Implementation Flow 26
    4.3. The Implemented Subsystem 26
        4.3.1. Module Listing 27
        4.3.2. The MicroBlaze Core 28
        4.3.3. The Local Memory 28
        4.3.4. The AXI Bus 28
    4.4. Implemented Software Functionality 29
        4.4.1. Initialization and Module Resetting 29
        4.4.2. Transmitter Gain Control 29
        4.4.3. System Error and Warning Messages 30
        4.4.4. Cooling Fan Control 30

4.4.5. Interactive Usage 31
4.4.6. Software Development Tools 32
4.5. Verification 32
5. EXPERIMENTS AND ANALYSIS 33
5.1. Implementation Results 33
5.2. Maximum Clock Frequency 33
5.3. Software Build Results 34
5.4. Software Gain Control Performance 35
6. DISCUSSION AND FURTHER DEVELOPMENT 36
7. SUMMARY 38
8. REFERENCES 39
9. APPENDICES 43

FOREWORD

The implementation work on this thesis was conducted at Nokia Solutions and Networks in 2015–2016. The purpose was to study the feasibility and utility of a soft-core control processor as a component of an experimental digital front-end implementation.

My gratitude goes to Dr. Tech. Jani Boutellier for supervising the thesis and for providing helpful advice and guidance. I also thank Chief Engineer Jukka Lahti for acting as the second examiner. I want to thank Juha Yrjänäinen, my manager and technical advisor, for providing me the opportunity to write my thesis at Nokia, and for offering valuable insight and inspiration.

I also thank my family for all their help and support. Finally, I want to mention all my colleagues at Nokia Solutions and Networks who provided valuable feedback and helped me in my everyday tasks. This work would not have been possible without you!

Teemu Vikamaa

Oulu, May 26, 2016

ABBREVIATIONS

AGC	Automatic Gain Control
ALU	Arithmetic and Logic Unit
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BRAM	Block Random Access Memory
BSP	Board Support Package
CLB	Configurable Logic Block
CMOS	Complementary Metal-Oxide Semiconductor
CPLD	Complex PLD
CPU	Central Processing Unit
CTS	Clear to Send
DC	Duty Cycle
DDR	Double Data Rate
DFE	Digital (Radio Frequency) Front-End
DIMM	Dual In-line Memory Module
DSP	Digital Signal Processing
EEPROM	Electrically Erasable Programmable ROM
EPROM	Erasable Programmable ROM
ELF	Executable and Linking Format
FIFO	First In, First Out
FMC	FPGA Mezzanine Card
FPGA	Field-Programmable Gate Array
GCC	GNU Compiler Collection
GPIO	General-Purpose Input/Output
GUI	Graphical User Interface
HDL	Hardware Description Language
HKMG	High-K Metal Gate
HPC	High Pin Count
HPL	High Performance, Low-Power
HW	Hardware
I/O	Input/Output
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
IF	Intermediate Frequency
IP	Intellectual Property (legal concept)
I²C	Inter-Integrated Circuit
LED	Light-Emitting Diode
LMB	Local Memory Bus
LUT	Lookup Table
MAC	Media Access Control
MB	MicroBlaze soft microprocessor
MCU	Microcontroller Unit
MDM	MicroBlaze Debug Module
MMU	Memory Management Unit
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
OFDM	Orthogonal Frequency-Division Multiplexing

PHY	Physical Layer (in the Open Systems Interconnection model)
PLD	Programmable Logic Device
PLL	Phase-Locked Loop
PWM	Pulse Width Modulation
RAM	Random Access Memory
RF	Radio Frequency
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RPM	Revolutions per Minute
RTL	Register Transfer Level
RTOS	Real-Time Operating System
RTS	Request to Send
SDK	Software Development Kit
SDR	Software-Defined Radio
SMA	SubMiniature version A
SoC	System on a Chip
SPI	Serial Peripheral Interface
SRAM	Static RAM
SW	Software
Tcl	Tool Command Language
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
VHDL	Very high speed integrated circuit HDL

ºC	Degrees Celsius
F	Farads
Hz	Hertz (cycles per second)
Ω	Ohms
s	Seconds
Sp	Samples

1. INTRODUCTION

The digitalization of radio transceivers has been a groundbreaking development in communication technology. Replacing analog signal processing components with their digital counterparts has enabled continued advances in transceiver efficiency. Moreover, this development has enabled a more software-centric implementation of various components.

The RF front-end is the most critical part of a radio, as it generally defines the limits for flexibility and performance of the transceiver. A digital implementation of the RF front-end (the digital front-end, DFE) is paramount to the realization of the software-defined radio (SDR) concept. SDR provides various advantages over previous technologies, such as high adaptability, interoperability, and opportunistic spectrum utilization.

The system described in this thesis constitutes one submodule of DFE. This system is an example of a soft-core microcontroller unit. Microcontrollers are a type of microprocessor that has been used to implement embedded control in various engineered systems for decades. As embedded systems have proliferated into various industries, microcontrollers have become a central part of our everyday lives.

As field programmable gate arrays (FPGAs) have become increasingly common as a platform for embedded systems, there is a growing demand for general-purpose processors integrated with FPGAs. One solution to address this demand is to incorporate hard-wired processors into the FPGA chip. Another alternative is to implement soft-core processors in the FPGA logic fabric. While the former alternative is potentially more efficient in terms of performance, area, and power, soft-core processors are more flexible and do not require any specialized hardware components. [1]

This thesis describes a MicroBlaze-based soft processor subsystem, implemented on a Xilinx 7th series FPGA. The purpose of this subsystem is to act as a controller of a DFE, which is implemented on the same FPGA fabric. The subsystem constitutes of a MicroBlaze soft processor core, and various other customizable IP (Intellectual Property) cores, along with the software program that is executed by the processor during operation. Block RAM (BRAM) was used to store program instructions and data; external memory was not required. The implemented subsystem is also denoted as the MCU (Microcontroller Unit).

To verify its functionality, the MCU subsystem was integrated into the full DFE design and tested on an ADS7-V1 evaluation board. The tests included measurements for response times of automatic gain control, along with general functionality tests. The response time measurements were also compared against requirements given in IEEE 802.11 specifications.

1.1. The Structure of This Document

This thesis is partitioned as follows: Chapter 2 discusses the principles and main characteristics of digital radio-frequency front-end systems. Chapter 3 then describes the theory of embedded computer processors, with special attention toward soft processor cores. Chapter 4 describes the design of the implemented soft processor subsystem, and the main tools used in the design and implementation work. Chapter 5

presents observations from the system implementation and experiments, while Chapter 6 comments on these observations and suggests possible directions for further experimentation and development. Finally, Chapter 7 provides a summary of the thesis.

2. DIGITAL RADIO

The control system discussed in this thesis is designed to be embedded within an FPGA-based digital radio frequency (RF) front-end. To establish an understanding on the purpose and functionality of the RF front-end, some principles of radio transceiver architectures are discussed here.

The following chapters discuss RF front-end architectures primarily from the perspective of the radio receiver. This approach is taken since receiver and transmitter front-end architectures are mostly similar. The differences between the two arise from the fact that transmitters only need to process locally generated signals, whereas receivers must contend with noise and interference affecting the receiving antenna [2].

2.1. The Radio Frequency Front-End

The RF front-end is generally considered to consist of all the radio transceiver components between the antenna and the baseband module. In a receiver this includes all the filters, low-noise amplifiers (LNAs), and mixers, which convert the RF signal down to lower frequencies. The RF front-end is generally considered to be the most critical part of a radio receiver. It determines the trade-offs in overall system performance, power consumption, and physical size. [3]

The following subchapters present some of the RF front-end architecture alternatives described in literature. These architectures are generally classified by the number of incorporated *intermediate frequency* (IF) stages.

2.1.1. Single-Conversion Superheterodyne

A simple *superheterodyne* (often abbreviated "*superhet*") receiver consists of the following functional blocks, in order of signal flow from the antenna:

1. Selector filter
2. RF amplifier
3. RF-to-IF Mixer, with input from local oscillator (LO)
4. IF filter
5. IF amplifier
6. Demodulator
7. Post-amplifier

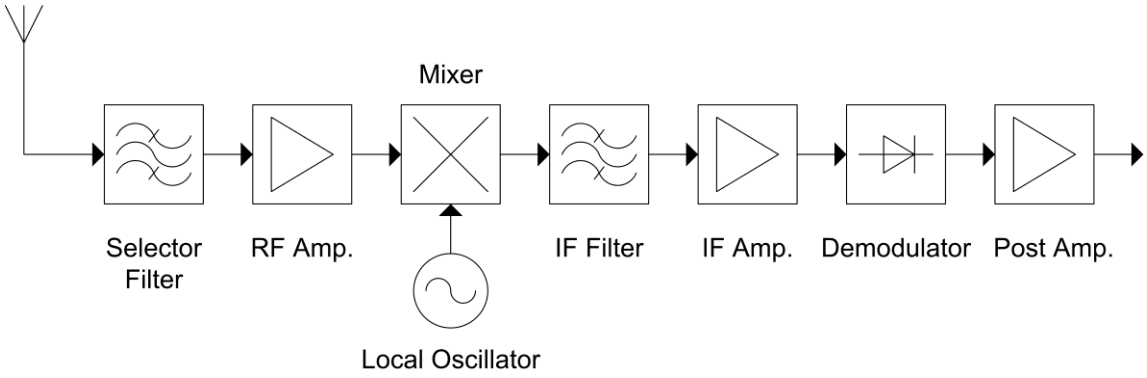


Figure 1. Functional block diagram of a single-conversion superheterodyne radio receiver.

The receiver frequency is tuned by varying the frequency of the local oscillator (LO). [4]

The RF-to-IF mixer stage generates an intermediate frequency signal with two receive frequencies. These two frequencies are mirrored around the LO frequency, both separated by the IF. The undesirable receive frequency is known as the *image frequency*. [4]

2.1.2. Multiple-Conversion Superheterodyne

In the superheterodyne receiver described above, an unwanted image frequency is generated and must be suppressed. In early superhets, solving this problem entailed a compromise between image suppression and selectivity. [4]

A second IF stage can be added into the superheterodyne receiver described above by repeating stages 3 – 5. This twofold heterodyning solves the problem of suppressing the unwanted image frequency without compromising selectivity. The resulting architecture is a *multiple-conversion superheterodyne* architecture with two IF stages. [4]

2.1.3. Direct Conversion

If the LO frequency is allowed to approach the receive frequency, the intermediate frequency will consequently approach zero. This means the signal will be mixed directly into the baseband frequency. The IF stage can thus be omitted, resulting in a *direct conversion* or *homodyne* architecture. The term *zero-IF* is also used. This architecture has become the most common choice for modern digital radios [11]. [4]

2.2. The Digital Front-End

Digital RF circuitry is considered one of the most prominent developments in radio technology in the last decade. Until the mid-1990's, RF transceiver circuit architectures had been analog-intensive, incorporating digital components only in the baseband parts. Since the late 1990's, these architectures have been successfully implemented with CMOS (Complementary Metal-Oxide Semiconductor). As CMOS

process technology has advanced at the pace of Moore's law, digital gate density – and consequently, digital processing performance – has doubled every 18–24 months. Analog CMOS performance, on the other hand, has been degrading after each gate density increment. Due to the complexity of interactions between transceiver components, it is very difficult to practically compensate for the degradation with digital logic. Hence, moving from the conventional transceiver architecture to a more digital one has been paramount to taking advantage of the continuous improvements in CMOS technology. [5]

The digital radio-frequency front-end (DFE) of a radio transceiver includes all the RF features that are realized with digital signal processing. These features, such as down-conversion and channel filtering, have previously been implemented with analog components. In addition to continued performance enhancement, a major motivation for replacing analog signal processing with the digital equivalent is to enable "soft" reconfiguration of the system. This feature is has enabled the concept of *software-defined radio*, which is discussed in the following chapter. [4] [6]

2.3. Software-Defined Radio

The software-defined radio (SDR) concept was originally introduced by Joe Mitola III in 1995. It described an idealized "software radio" architecture, where the receive/transmit antenna was connected directly to a set of analog-to-digital converters. This meant the received signal would be processed entirely using digital signal processing (DSP). The same applies to transmitted signals. It was acknowledged, that an actual radio device would still contain several analog components, such as RF conversion, RF distribution, anti-aliasing filters, and others. [2] [7]

No general, globally recognized definition exists for SDR [8]. One newer definition of software-defined radio is stated as "A radio in which some or all of the physical layer functions are software defined" [9] [10]. "Software-defined" is broadly understood as capability of supporting different waveforms through changes in the software or firmware of the device, without necessitating hardware changes. This means the hardware must be designed without waveform-specific components, at least according to a strict interpretation. A radio with built-in circuitry for different waveforms would instead be classified as *software-controlled*. [2]

The high adaptability of SDR can increase the efficiency of system resource (spectrum, power) usage. One way to realize these improvements is *opportunistic spectrum sharing*, where the radio can adapt to the spectrum at any given time, without having to commit to a selected frequency band. Moreover, SDR enables seamless interoperation between incompatible radio sets by acting as bridge between them. The adaptability of SDR also means the same hardware can be utilized in multiple types of applications. This has the potential of improving economy of production as devices become more generic, enabling a larger scale of production. [2] [11]

Because of its potential for interoperability and adaptability for different purposes, SDR has become highly desirable for military and law enforcement users. Radios made to military specifications have longer lifespans than the civilian equivalents, remaining in operational use alongside newer models. Moreover, the different branches and services within militaries have very different requirements for radios, which results in a large variety of devices with different waveforms. [2] [7]

The most prominent disadvantage of SDR is unit cost, which is a limiting factor in high-volume, low-margin products. Another pertinent issue is higher power consumption. One cause for this issue is the implementation of digital signal processing algorithms using programmable devices instead of hard-wired ASICs. Another source is wideband components, especially wideband power amplifiers, which account for at least 70 % of total power in radios. A narrowband power amplifier can be twice as efficient as its wideband counterpart. [2]

The signal processing devices used in SDR implementations include field-programmable gate-arrays (FPGAs), digital signal processors (DSPs), and general-purpose processors (GPPs) [2] [7] [11]. FPGA-based SDRs are can achieve high data rates and high bandwidth without excessive power consumption, which makes them suitable for high-performance, real-time usage. [2] [7]

2.4. The DFE Architecture Considered in This Thesis

The purpose of this thesis work is to implement a microcontroller unit (MCU) for an FPGA-based digital front-end of an SDR transceiver. A system level diagram of the DFE component is presented in Figure 2. The hardware platform for the system is described in Chapter 4.1.

The main features of the architecture are the MCU and the data path block, which incorporates interfaces to the baseband system and the RF daughterboard. All these features are implemented in the reprogrammable logic of the Virtex-7 FPGA. Several parallel transmit and receive data paths are included. Within each receive/transmit data path are common signal processing blocks performing common DFE tasks, such as channelization and digital down/up-conversion. No intermediate frequencies are incorporated, meaning the transceiver has a direct-conversion architecture.

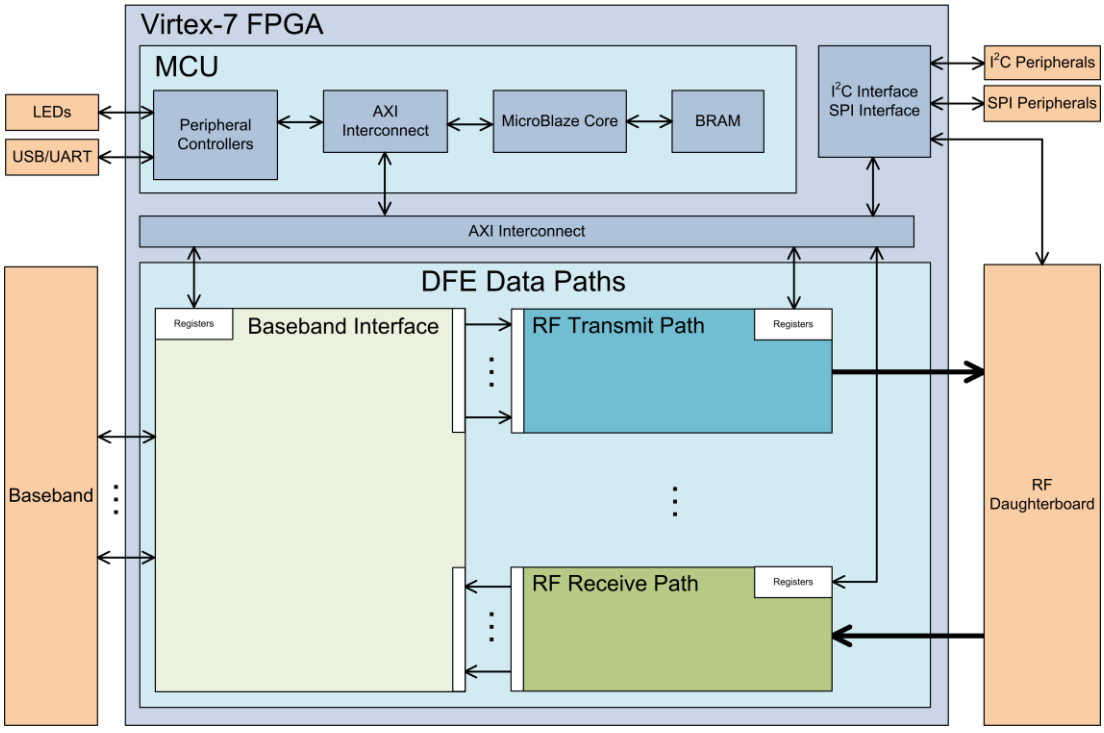


Figure 2. System level DFE block diagram.

2.5. Related work

2.5.1. WARP

The Wireless Open Access Research Platform (WARP) is a programmable wireless transceiver platform for network prototyping. In the most recent version (v3), the platform is built around a Xilinx Virtex-6 FPGA, which facilitates software-defined radio implementations. [14]

The WARP v3 board has two integrated RF interfaces. Each RF interface has a 40 MHz bandwidth, and is equipped with a 12-bit 100 MSp/s analog-to-digital converter (ADC), and a 12-bit 170 MSp/s digital-to-analog converter (DAC). A 50 Ω SMA (Subminiature version A) jack is provided for connecting an RF cable or antenna to each interface. [14]

An IEEE 802.11 OFDM PHY reference design, which is available on the project website, provides an example of a software-defined physical layer implementation. It incorporates two MicroBlaze soft processor cores with separate block random access memories (BRAMs). The "low" processor is responsible for low-level MAC distributed coordination function software. This software handles MAC-PHY interactions, as well as time-critical intra-packet state. Meanwhile, the "high" processor executes all top-level MAC code and other high-level functions. This includes constructions of non-control packets for transmission, and handshaking between network nodes. Both MicroBlaze cores are clocked at 160 MHz. The "low" core utilizes 64 kB of BRAM, while the "high" core utilizes 256 kB.

2.5.2. USRP

The X310 is an SDR platform designed for wireless system design and deployment. It is a 3rd generation device in the Universal Software Radio Peripheral (USRP™) device family from Ettus Research. The X300 is a very similar platform, the only major technical difference being the lower capacity of the incorporated FPGA chip. [15]

The X310 and X310 both incorporate a Xilinx Kintex-7 family FPGA. The specific device model is XC7K410T, whereas the X300 equivalent is XC7K325T. The board has two 14-bit 200 MSp/s ADCs, and two 16-bit 800 MSp/s DACs. Two RF daughterboard slots are included, covering frequencies up to 6 GHz with up to 120 MHz baseband bandwidth. [15]

The N300 and N310 are earlier, 2nd generation USRP platforms. Both platforms incorporate a Xilinx Spartan-3A DSP FPGA. As in the X Series, the main difference between the two platform versions is the higher capacity FPGA of the N310. The N300/N310 board has two 14-bit 100 MSp/s ADCs, and two 16-bit 400 MSp/s DACs. [16]

The FPGA firmware for the X300 and X310 incorporates an open-source ZPU soft processor core running at 166.667 MHz, with 32 kB of program BRAM. A ZPU core is also included in the N300/N310 firmware with a 50 MHz clock rate (divided from the 100 MHz main clock). In the N300/N310 case, 32 kB of BRAM is allocated in two 16 kB parts, of which one is reserved for a bootloader. [17]

A paper from 2013 describes an IEEE 802.11a/g/p compliant receiver based on the USRP N210 platform, paired with an XCVR2450 RF daughterboard. The receiver was implemented using the open source GNU Radio signal processing framework, with no

custom changes to the FPGA firmware. The receiver supports up to 20 MHz of bandwidth. [18]

An automatic gain control (AGC) feature was later added to the receiver design. The gain calculation was implemented directly on the FPGA, which was necessary to meet the real-time requirements defined for AGC in the IEEE 802.11 specifications (8 µs for a/g, and 16 µs for p). The actual signal scaling occurs on the MAX2829 transceiver chip, which is incorporated into the RF daughterboard. [19]

3. ARCHITECTURE OF EMBEDDED SYSTEMS AND MICROCONTROLLERS

This part presents some of the main principles of embedded system design, including real-time systems and microcontrollers. Soft-core processors and field-programmable gate array (FPGA) technologies are discussed in greater detail, as they are most relevant to the implementation part of this thesis.

3.1. Introduction to Embedded Systems

Embedded systems exist in a huge number of applications – in cars, household appliances, televisions, phones and almost every other electrical device. They have become a vital part of our everyday lives. [20] [21] [22] [26]

An embedded system can be defined loosely as a device that includes a programmable computer, but is not intended to perform general-purpose computation [20] [21]. Since this concept currently permeates such a vast number of engineering market segments, there is no precise definition that could encompass all of them [22].

The performance of embedded systems is commonly measured by the ability to provide real-time functionality. This means the system must produce a valid output for a given input before a specified deadline. This is in contrast to general-purpose computing, where the goal is to achieve "good enough" performance. [21]

Real-time embedded systems are often expected to run several tasks on a limited set of processor instances. The act of defining the run order of pending tasks is called *scheduling*. It is generally a difficult problem to form a scheduling for a set tasks so that all real-time constraints are met. The choice is often a compromise between conflicting criteria. [23]

The *round-robin* method is possibly the simplest common type of scheduling. It means tasks are checked for readiness one by one in predetermined order, and executed immediately when needed. This method is easy to implement and analyze in terms of timing guarantees. The problem with round-robin scheduling is that the maximum delay for task completion can be very long as each task may need to wait for all other tasks to complete. Moreover, the overhead arising from checking each task for readiness can become significant compared to actual task execution times if some tasks are rarely enabled. One simple way of decreasing service delays for urgent tasks is to check their readiness more than once per cycle, a method known as *static cycle scheduling*. [23]

The two methods described above are examples of *static scheduling policies*, where task execution order is determined before runtime. Meanwhile, with a *dynamic scheduling policy*, task execution order is determined at runtime as tasks become ready for processing. The execution order in dynamic scheduling is generally controlled with task-specific *priorities*. If tasks are activated at irregular intervals, dynamic scheduling can achieve much better processor utilization than static scheduling, assuming preemption is also used. Preemption means that a running task can be suspended before launching another task of higher priority, and continued after all higher priority tasks have been completed. [23]

3.2. Microcontrollers

A *microcontroller* is a microprocessor designed to act as an embedded control processor in a larger, non-computational system. The characteristics of typical microcontrollers are considerably different from processors designed for general-purpose computation. Instead of high performance, microcontrollers are optimized primarily for low cost and compactness. Input and output (I/O) capabilities are usually more critical than processing power. Data paths are often narrower – 16, 8 or even 4 bits have been used. Furthermore, microcontrollers must be designed to withstand the same environmental conditions as the systems they are embedded to. [20] [24]

A microcontroller chip normally consists of a processor core, memories for program instructions and data, and peripheral components, such as I/O controllers and timers. Like any digital circuit, a microcontroller needs a power source and at least one clock signal, which may be generated on-chip. [20]

3.3. Soft Processor Cores

Soft-core processors are generic processor models that can be customized for specific applications and synthesized for different implementation technologies, such as ASICs and FPGAs. In contrast, hard processor cores are made available in a fully implemented, "hardwired" form. If a core is described in a manner that is specific to an FPGA's structure, such as a placed and routed block of configurable logic blocks (CLBs), it is known as a *firm* core. [25] [26]

Soft-core processors are highly flexible, allowing a large degree of customizability for application-specific requirements. By definition, soft cores are technology-independent, which means they can be realized in different ASIC and FPGA technologies with reasonable effort. This also provides the benefit of higher forward-compatibility since there is no need to redesign a soft core in order to migrate it to a newer implementation technology. [25] [26]

Hard-core processors can be fast and relatively efficient. However, a soft-core-based system can be adapted for the specific application in several ways, such as by modifying the implementation and complexity of the processors, and by changing the number of cores. The number of soft core instances is constrained only by the availability of circuit resources [25]. This is opposed to hard-core processors, where the lack of adaptability often leads to either wasted or insufficient performance. Moreover, when hard cores are embedded into an FPGA, their fixed location may complicate routing between the cores and the general-purpose programmable logic. Finally, the presence of hard cores in an FPGA specializes the product as a whole, narrowing its customer base. [1]

3.4. Abstraction Levels in Soft Processor Design

Soft processor cores are usually designed and distributed using *hardware description languages* (HDLs). HDLs are generally useful for describing the electronic parts of integrated circuits (ICs) and printed circuit boards. While some HDLs can describe analog electronics, only digital electronics are considered relevant in this thesis. The

main HDLs in use today are Verilog (IEEE–STD–1364) and VHDL (IEEE–STD–1076). [25] [26]

HDL-based design flows were introduced in the 1980s as schematic-based design had become too inefficient for describing increasingly complex ASIC designs. Early HDLs were largely specific to the EDA (Electronic Design Automation) tools used at the time. [25]

Digital circuits can be described on various levels of abstraction. The following three levels, from lowest to highest, are recognized in literature: [25] [27]

- Structural – switch level, gate level
- Functional – Boolean equations, register transfer level (RTL)
- Behavioral – programmatic expressions and constructs

Whereas early HDLs often supported only structural level models, current languages support all levels. The same language can thus be used in all stages of design and simulation, without any need for specialized tools. [27]

3.5. Tools for IP-based Processor Design

In the late 1980s, schematic-based design started to become too cumbersome in the face of increasing design complexity. While this problem has been addressed with HDLs, hierarchical schematics are still a useful design tool. In a hierarchical schematic, related components of the design can be grouped together into encapsulated blocks, which means the whole design can be visualized from a high-level, top-down perspective. [25]

IP-based design is a form of digital design, where existing intellectual property blocks are structurally connected together to form a whole system. In this type of design process, schematic tools are highly useful and widely available. Examples of IP-based schematic design tools include ISE (Integrated System Environment) and Vivado IP Integrator from Xilinx, and Qsys from Altera [28] [29]. To expedite IP integration, schematic tools may provide design assistance, such as design rule checking, and automated IP block instantiation and connections, which are included in the Vivado IP Integrator [30].

IP-based design tools may also provide a customization GUI for setting the parameters of instantiated IP cores. The GUI may provide guidance during the IP customization workflow, validating given parameter values, or display estimates of resource usage and timing before the actual synthesis flow is initiated. Figure 3 shows an example of a tabbed customization dialog of the MicroBlaze IP core from Vivado IP Integrator. The “General” tab lets the designer configure various features of the processor core instance, such as the inclusion of optional instructions. Other tabs provide controls for features such as such hardware exceptions, cache memories, the memory management unit (MMU), bus interfaces, and more.

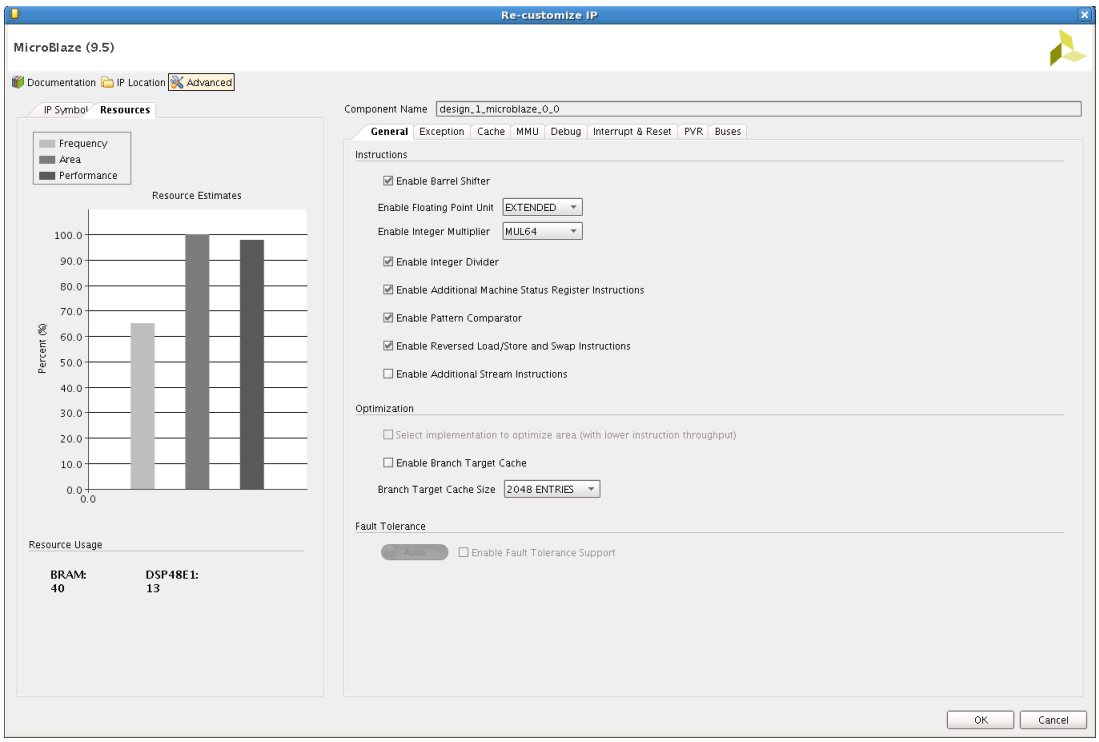


Figure 3. Screenshot from a customization GUI dialog in Vivado.

3.6. Hardware/Software Codesign and Coverification

When a programmable processor is included in an electronic system, the system designers must consider the choice of which features to implement as hardware components, and which ones as software tasks. The main goal is to choose the most cost-efficient implementation for each feature. [25]

Before HW/SW codesign, microprocessors had been used as board-level components for at least a decade. It was apparent that the advancement of integrated circuit technology, as stated in Moore's law, would enable chips to accommodate large systems with CPUs as integrated components. This entailed that system design would have to accommodate large predesigned CPUs, while also considering software as a first-class system component. [31]

The first HW/SW cosynthesis tools were born in the early 1990s. The CODES and CASHE workshops introduced several pieces of research on co-design, and HW/SW partitioning emerged from these as a first step for system modeling and algorithm design. HW/SW partitioning was formulated as the mapping of a design onto a target architecture comprised of CPU cores, ASIC cores, memory, and an interconnect structure between the various cores. [31]

An essential component of HW/SW codesign is cosimulation. As high-accuracy simulation requires substantial computational effort, mixed-level simulation was introduced to give designers the ability to select different detail levels for different components. This way enough input vectors can be executed to properly validate the design. [31]

The design flows of hardware and software are projected to converge in the future so that an entire system can be designed in a single step. Design tools can then be used

to generate the hardware processors and software programs implementing the whole system. [24]

3.7. Field-Programmable Gate Arrays

Field-programmable gate arrays (FPGAs) were introduced in the 1980s to address the gap between programmable logic devices (PLDs), which were highly flexible and configurable, and application-specific integrated circuits (ASICs), which could be used to implement more complex functionality. The first commercial FPGA chip was released by Xilinx Inc. in 1984 [32]. [25]

FPGAs are a prime platform for custom soft core based designs since soft cores can be implemented on the programmable fabric of any FPGA; there is no need for expensive and time-consuming CMOS production cycles. The main drawbacks of FPGA-based implementations compared to custom CMOS are significantly lower speed and higher chip area usage. A soft processor core implemented on an FPGA may take 18-26 times the area of a CMOS implementation, and have 17-27 times higher timing delays [33].

3.7.1. Architecture

FPGA architectures are fundamentally based on configurable logic blocks (CLBs), input/output (I/O) blocks, and programmable interconnects which are used to route signals between logic and I/O blocks. CLBs are also known as logic array blocks (LABs) in Altera's nomenclature. Clock circuitry is also needed to drive the clock inputs of memory cells within the CLBs. This generally applies to all FPGAs, although different FPGA manufacturers have their own architecture variants. [27]

In addition to the aforementioned features, FPGAs usually incorporate other embedded structures or "macro blocks". Embedded random-access memory blocks (known as e-RAM or BRAM) is included in all current FPGAs, occupying significant portions of die area [32]. Some functions that are inherently inefficient when implemented in CLBs, such as multipliers, may be included as hard-wired computational blocks. Entire hard microprocessor cores may also be included, either within the FPGA programmable logic fabric, or in a separate "stripe" area. [2] [25]

Figure 4 below illustrates the general structure of FPGAs, as described in literature [25] [27] [32]. The diagram shows a hypothetical architecture comprised of general programmable logic, specialized DSP blocks, and BRAM.

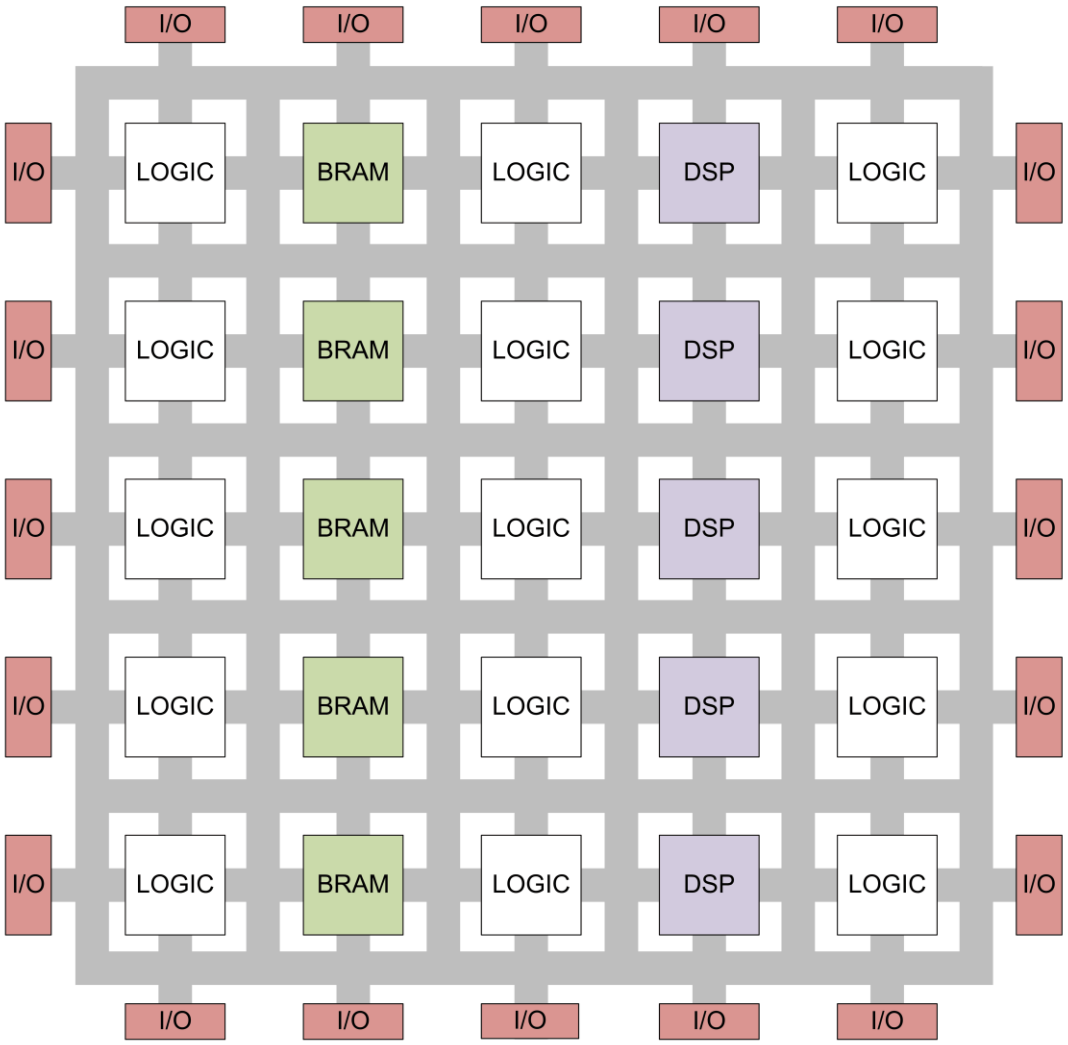


Figure 4. An example of the basic structure of field-programmable gate arrays.

One way to classify FPGA architectures is by comparing the sizes of their CLBs. A fine-grained FPGA architecture contains a relatively large number of small CLBs, while a coarse-grained FPGA has a smaller number of larger CLBs. A fine-grained CLB might accommodate a single logic gate with relatively few inputs, or a single memory element, such as a latch or flip-flop. Meanwhile, coarse-grained CLBs can contain several look-up tables (LUTs) for implementing larger logic functions, as well as several memory elements, multiplexers, and fast carry chain logic. [25]

Fine-grained FPGAs are particularly efficient for implementing glue logic and irregular structures, such as state machines. They are also highly suitable for implementing systolic algorithms. On the other hand, more coarse-grained architectures have the advantage of requiring less CLB interconnections, which often incur greater timing delays than the logic itself. [25]

Despite the advantages of fine-grained architectures, FPGA vendors have generally decided to move toward coarser architectures. In FPGAs, the programmable interconnects are the dominant source of delays rather than logic, and coarser architectures require less interconnect-based routing between CLBs. Hence, coarse architectures provide higher overall efficiency. Experimental research has shown, that

a logic block with 4–9 LUT/flip-flop pairs can provide optimum efficiency in terms of delay as well as chip area [34]. [25] [27]

3.7.2. Implementation Technologies

The programmability (or configurability) of FPGAs indicates that their functionality can be customized after fabrication. This is made possible by the underlying programming technology, which is used to control the programmable switches within the FPGA chip. [32]

This subchapter presents the most commonly used programming technologies and some of their implementation details.

SRAM:

SRAM-based FPGAs contain static RAM as programmable elements. SRAM bits can be combined into a memory block or used as LUTs to implement arbitrary combinational logic functions. SRAM bits may also be used to control multiplexers, which are used to select specific logic within a CLB, or individual transistors. For routing configuration, SRAM bits can control individual transistors as well as multiplexers. [27]

SRAM is the most common technology used for implementing FPGAs for mainly two reasons: it is fabricated using standard CMOS processes, and it is indefinitely reprogrammable. The downsides of SRAM include volatility, which means the device needs to be reprogrammed after each power cycle. Another disadvantage is the relatively large size of SRAM cells. [32]

Floating-gate:

Floating-gate memory technologies are one alternative addressing the shortcomings of SRAM. Flash is one such technology, as are EPROM and EEPROM. The latter two are no longer widely used for various reasons. EPROM is not in-circuit programmable, and needs to be treated with a special ultraviolet eraser before reprogramming. Meanwhile, EEPROM occupies about twice as much chip area as EPROM. Flash therefore combines the advantages of both EPROM and EEPROM. [32] [35] [36]

Compared to SRAM, all floating-gate technologies have the disadvantage of increased power consumption due to the large number of internal pull-up resistors. Moreover, they all require more stages in the fabrication process. Finally, the number of reprogramming cycles for flash-based programmable logic is limited due to charge buildup. [25] [32] [35]

Anti-fuse:

Whereas a conventional fuse breaks a connection when the current flowing through it exceeds a certain level, an anti-fuse works in the opposite way – by forming a connection when the given current level is exceeded. This forms a basis for a non-reprogrammable FPGA technology. [27]

Anti-fuse FPGAs are suitable for applications where the device needs to be non-volatile and not susceptible to inadvertent changes from glitches or radiation.

However, other FPGA devices are also available in radiation-hardened packaging. [25] [27] [35]

Probably the most essential advantage of anti-fuse FPGAs is IP security. Retrieving the bitstream from the programmed anti-fuse FPGA can be effectively prevented by setting a special security anti-fuse, which blocks any further programming data transfers. Reverse-engineering a programmed anti-fuse device is considered effectively impossible. [25] [27]

Anti-fuse FPGAs have the advantage of lower power-consumption compared to SRAM-based FPGAs. Anti-fuses are also very area-efficient, although this advantage is partially offset by the need for large programming transistors. Anti-fuse FPGAs can theoretically reach much higher speeds than SRAM. However, this advantage is not very meaningful for actual designs since SRAM technology is much more common and highly standardized, and therefore more attention has been given to optimizing it. [25] [27] [32]

Since anti-fuse FPGAs are one-time programmable, they are unsuitable for development and prototyping. Moreover, some manufacturing defects cannot be detected before programming. For a one-time programmable FPGA, this means some devices (up to 10 %) must be discarded after programming. [25] [32]

Summary:

A summary of the essential characteristics of practically relevant FPGA technologies is presented in Table 1. [32] [35] [36]

Table 1. Comparison of SRAM, Flash, and anti-fuse implementation technologies

	SRAM	**Flash**	**Anti-fuse**
Reprogrammable	✓	✓	
Volatile	✓		
Intrinsic IP security			✓
Transistors per storage element	6	1	None (Very small area)
Manufacturing process	Standard CMOS	Standard MOSFET	Modified CMOS
Switch resistance	~ 500 – 1000 Ω	~ 500 – 1000 Ω	20 – 100 Ω
Switch capacitance	~ 1 – 2 fF	~ 1 – 2 fF	< 1 fF
Failure rate after programming	None	None	Up to 10 %

4. IMPLEMENTATION

4.1. The Hardware Environment

The ADS7-V1 evaluation board from Analog Devices, Inc. forms the hardware platform for the digital front-end (DFE). The following features are included on the board: [37]

- Xilinx Virtex-7 FPGA (Part code: XC7VX690T3FFG1761–3)
- Two FMC-HPC connectors
- Ten 13.1 Gb/s transceivers per each FMC-HPC connector
- Two DDR3-1866 DIMMs
- USB 2.0 interface

The subsystem was implemented on the Virtex–7 FPGA. This device belongs to the Virtex line of devices, which is designed for maximum performance. The other two FPGA families in the Xilinx 7th series are Artix, which is optimized for low cost and power, and Kintex, which is a mid-range option between Artix and Virtex. All 7 series FPGAs are manufactured with a high performance, low-power (HPL), 28 nanometer high-k metal gate (HKMG) process technology. [38]

The FMC-HPC connectors were used for connecting a separate radio frequency (RF) interface board. This is in contrast to the WARP platform, which has two integrated RF interfaces [14]. The Virtex–7 FPGA is noted as superior to those incorporated in the WARP and USRP X310 platforms described in Chapter 2.5. It can thus be considered highly suitable for software-defined RF front-end implementations.

For standard software input/output, a UART interface core is included in the MCU subsystem. The external signal ports of this core are connected to an on-board USB-to-UART bridge via the FPGA I/O pins. Four UART signals are connected: data transmit, data receive, request to send (RTS), and clear to send (CTS).

For the purposes of the MCU implementation, the most central hardware features are the FPGA and the USB interface, including the USB-to-UART bridge.

4.2. Vivado Design Suite

The Vivado Design Suite from Xilinx is architected to increase overall productivity when designing, integrating, and implementing systems on 7th Series FPGAs, Zynq-7000 All Programmable SoCs, and UltraScale devices. It replaces the earlier Xilinx ISE Design Suite. [39]

All Vivado Design Suite tools include a native Tool Command Language (Tcl) interface. All commands and options available in the Vivado Integrated Design Environment, which forms the graphical user interface (GUI) for the Vivado Design Suite, are usable through Tcl. [39]

Vivado version 2015.1 was used over the course of this design project. All design work was done using the project-based workflow in the GUI.

4.2.1. Block design

The block design environment in the Vivado Design Suite features a hierarchical schematic editor. IP blocks are selected from an IP catalog, and parametrized using the window-based GUI. Some design constraints, such as clock frequencies, can also be defined in the block design GUI, and propagated automatically.

The design creation flow used to create the MCU submodule can be roughly described as follows:

1. Add a MicroBlaze core into the block design
2. Run block automation.
3. Add AXI peripherals and external ports
4. Run connection automation to create an AXI interconnect block
5. Create input ports for interrupts, and connect them to the AXI interrupt controller (via a Concat block).
6. Connect any remaining external ports.
7. Configure all the IP blocks as required.
8. Configure the memory map.
9. Set the MicroBlaze vector base address to point to the beginning of the instruction memory segment.

All the aforementioned steps can be repeated as needed to address new design requirements or to fix defects. The quick validation feature is also highly useful to check that the design is ready for the implementation flow.

4.2.2. Synthesis and Implementation Flow

When the design is fully connected, configured, and validated, a synthesis run can be launched. After a synthesized design has been completed, the *elaborated design* can be opened. This allows external ports to be mapped to FPGA I/O pins in the GUI. Once I/O has been appropriately configured for each external port in the design, an implementation run can be launched. Since there was only one clock source and no synchronous external I/O in the MCU design, there was no need to define constraints other than selecting the I/O standard and device pin for each external port.

After an implementation run was completed with successful timing, the last step left was to generate the bitstream. In this project, software was run directly from BRAM, which means the MCU BRAM section had to be initialized during configuration. This was achieved in Vivado by associating the software executable file with the bitstream. A tool named UpdateMEM is also available for merging the software executable into the bitstream.

4.3. The Implemented Subsystem

The implemented MCU subsystem constitutes the software-programmable control logic of the surrounding digital front-end (DFE). The subsystem consists of a single MicroBlaze core, a BRAM block for software instructions and data, several AXI peripheral cores, and miscellaneous cores. An external AXI4 Lite port is used to access control registers in the surrounding DFE system.

Figure 5 below displays a functional block diagram of the subsystem. Debug features have been omitted for clarity.

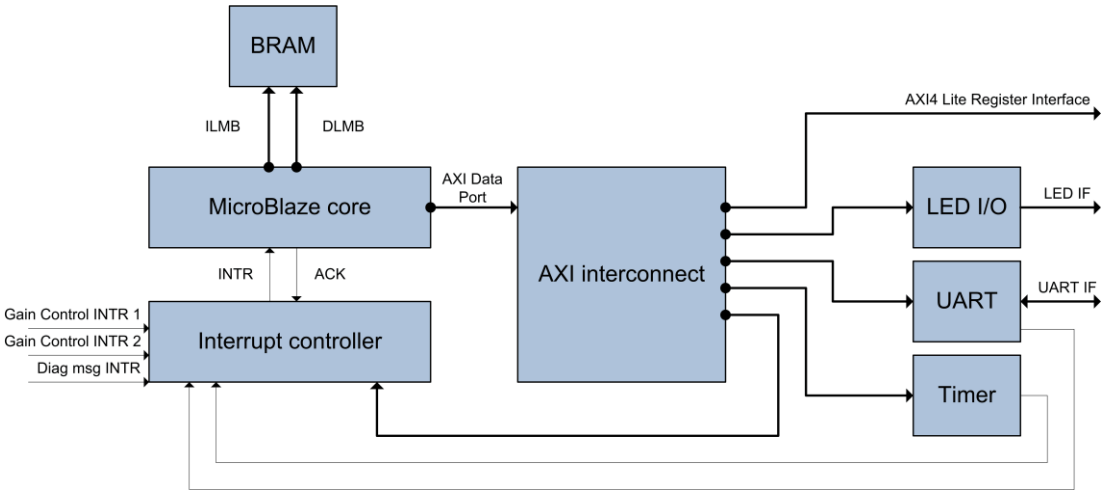


Figure 5. Block diagram of the MCU subsystem. The black dots (●) denote bus master ports.

4.3.1. Module Listing

Table 2 lists the IP cores of the MCU subsystem and their brief descriptions. The full block design diagram is enclosed in Appendix 1.

Table 2. IP cores and macroblocks in the MCU subsystem

Module name in the block design	IP name	Notes
microblaze_0	MicroBlaze	
microblaze_0_local_memory	*Macroblock*	512 kB true dual-port BRAM. Includes two BRAM controllers and three local memory bus IPs.
microblaze_0_axi_periph	*Macroblock*	Automatically generated. Includes an AXI crossbar and the necessary protocol converters.
axi_intc_0	AXI Interrupt Controller	
axi_timer_0	AXI Timer	General-purpose 32-bit timer
axi_led_gpio_0	AXI GPIO	LED
axi_uart_16550_0	AXI UART 16550	UART I/O interface
mdm_1	MicroBlaze Debug Module (MDM)	
rst_clk_125M	Processor system reset	Reset synchronizer
xlnconcat_0	Concat	Concatenate interrupt signals into a single bus

4.3.2. The MicroBlaze Core

The MicroBlaze embedded processor is a 32-bit reduced instruction set computer (RISC) soft core optimized for implementation in Xilinx FPGAs. It uses a Harvard memory architecture, meaning there are separate memory interfaces for instructions and data. [40]

The core instance included in the MCU block design (*microblaze_0*) was customized in most aspects to maximize performance. A full, 5-stage processing pipeline was used, and all optional instructions were enabled, except additional stream instructions since no stream interfaces were used. Caching was disabled, since BRAM is used for main memory. No virtual memory management was included, and memory fault tolerance features were disabled. Little-endian byte order was used in the MCU and throughout the rest of the DFE system. All available hardware exceptions were enabled.

The optional instruction set features used in the MCU include all MicroBlaze floating-point instructions. The native floating point format is IEEE 754-1985 single precision, which includes a single sign bit, an 8-bit biased exponent, and a 23-bit significand [40].

The external interface signals of the instantiated MicroBlaze core consist of two local memory buses (for instructions and data), an AXI4 data bus, and interrupt inputs. The core is configured for fast interrupt operation, which means interrupts are acknowledged to the interrupt controller (*axi_intc_0*) via direct signals.

4.3.3. The Local Memory

The Virtex-7 FPGA contains 1,470 BRAM blocks, arranged into 15 columns. The capacity is 36 kilobits per block. Each block can also be configured as two independent 16 kilobit blocks. [41]

A 512 kB BRAM section (exactly $7FFFF_{16} = 524287_{10}$ bytes) was instantiated for software program instructions and data. The local memory macroblock (*microblaze_0_local_memory*) contains a true dual-port BRAM instance along with two local memory bus (LMB) controllers. These correspond to the two LMB interfaces in the MicroBlaze core. Since the instruction and data LMB ports are connected to the same memory space, self-modifying software code can be executed on the MCU. An additional LMB data bus is connected to the MicroBlaze debug module (*mdm_1*).

4.3.4. The AXI Bus

A peripheral AXI4 data interface was added to the MicroBlaze core to facilitate memory-mapped access to control registers outside the MCU subsystem. The MCU also includes four built-in AXI peripherals: an interrupt controller (*axi_intc_0*), a general-purpose I/O interface (*axi_led_gpio_0*), a UART 16550 interface (*axi_uart_16550_0*), and a general-purpose timer (*axi_timer_0*). An automatically generated AXI interconnect macroblock (*microblaze_0_axi_periph*) connects the AXI master port of the MicroBlaze core to the peripherals. The MicroBlaze debug module (*mdm_1*) has a similar AXI master port, giving it access to peripheral registers.

4.4. Implemented Software Functionality

The following subchapters describe the features specified for the embedded bare metal software suite. “Bare metal” means the software can access hardware resources directly, without an operating system acting as an interface between hardware and software. Both interrupt-triggered and polled tasks are incorporated. Interrupts are serviced preemptively with static priorities, which are determined by the order of interrupt controller port assignments – index 0 has highest priority. (This input is connected to the *Dbg_Intr* port in the MicroBlaze core, see Appendix 1).

The software suite was implemented in the C programming language. The source code was organized in such way that device-specific parts are separated as clearly as possible from portable parts. Thus, most of the software suite could be compiled to a Linux desktop executable, which enables quick testing of all algorithmic functionalities.

The software was implemented using version 5.0 of the Xilinx standalone driver collection. These drivers implement the board support package (BSP) – a low level software library for accessing the MCU hardware. All drivers were statically linked, which means no separate runtime libraries were associated with the software executable file.

While register mapping for the peripheral cores in the MCU subsystem was handled in Vivado, mapping for DFE system registers was conducted with a third party tool flow. This flow involved IP-XACT (IEEE–STD–1685) compliant register map definitions, which were used as input for a register wrapper generator. The generator was used to produce C header files, which formed a *hardware abstraction layer* over the system registers.

4.4.1. Initialization and Module Resetting

After power-up or hardware reset, the various submodules of the DFE system need to be initialized by writing data to appropriate registers. This can be done interactively by writing commands to the MCU via the UART/USB interface, or by embedding a command sequence into the software executable. To connect to the UART/USB interface, device-specific drivers must be installed on the host computer.

4.4.2. Transmitter Gain Control

The DFE system, into which the MCU control system is embedded, includes automatic gain control (AGC) functionality for each transmission data path. Scaling coefficients for the data paths are computed in software upon interrupt activation. The MCU has several interrupt inputs so that each data path can trigger an interrupt independently. Calculations are executed in floating-point arithmetic, which means the software converts the fixed-point inputs into floating-point format, and the calculated coefficient is conversely converted back to fixed-point. The fixed-point formats are defined by the DFE hardware implementation. The scaling itself is realized as a simple multiplication, applied onto each frequency channel on the FPGA fabric. (A data path constitutes several frequency channels.)

The coefficient calculation is based on a given target signal power level (P_{target}), and the measured signal power ($P_{measured}$), which is obtained from power measurement

blocks located on each transmitter path after the scaling blocks. This means the AGC is implemented using a closed control loop. Each power measurement block also smooths the measured power value by averaging it over time. The scaling coefficients (C) are calculated according to the following formula:

$$C = \sqrt{\frac{P_{target}}{P_{measured}}} \tag{1}$$

The software algorithm also includes a register-configurable hysteresis feature, which determines whether the scaling coefficients need to be updated. The hysteresis condition is determined with the following algorithm:

1. Read the data path specific average power, scaling coefficient, and hysteresis threshold from system registers.
2. Calculate the unscaled average power by dividing the average power value with C^2.
3. Calculate the relative difference between the unscaled average power value and the target power value.
4. Calculate a new value for C, if the relative difference from step 3 is greater than the hysteresis threshold.

4.4.3. System Error and Warning Messages

To aid testing and maintenance, it is necessary to obtain information about the internal state of the DFE system. For this purpose, an interrupt-triggered software routine that collects information regarding the internal state was created.

After the routine is triggered by an interrupt, it reads a single system register via the AXI register interface. The bit fields in this register indicate the submodules that have experienced an error condition. If the register value is zero, no error condition has occurred. Additional registers for detailed submodule-specific information are also included. After the appropriate information has been obtained, the MCU transmits human-readable messages to a monitoring host terminal via the UART/USB interface, and resets all the relevant registers to zero to clear the error state.

4.4.4. Cooling Fan Control

To automatically adjust the rotation rate of cooling fans in the DFE system, the MCU software includes routines for temperature sensor reading and pulse width modulation (PWM) duty cycle (DC) calculation. The temperature is polled regularly, and the desired fan PWM duty cycle is calculated from the temperature reading according to a specified transfer function. The temperature sensor itself is an external peripheral, accessible via an SPI bus. The duty cycle, expressed as a percentage, is written into a memory-mapped register via the AXI interface. This register controls the actual PWM signal generator, which is implemented on the FPGA fabric.

The transfer function for the duty cycle (DC) calculation is a piecewise affine map, defined according to Equation 2. The parameters are defined in another memory-mapped register, which contains fields for the temperature range [T_{min}, T_{max}] (in ºC) and the duty cycle range [DC_{min}, DC_{max}].

$$DC = \begin{cases} DC_{min}, & \text{when } T < T_{min} \\ (T - T_{min})\frac{DC_{max} - DC_{min}}{T_{max} - T_{min}}, & \text{when } T_{min} \leq T \leq T_{max} \\ DC_{max}, & \text{when } T > T_{max} \end{cases} \quad (2)$$

The cooling fans may be equipped with tachometers that send pulses to the FPGA as the fans rotate. A pulse counter submodule is implemented on the FPGA with memory-mapped registers showing the RPM count. The MCU can monitor the actual RPM by reading these registers, which facilitates automatic handling of cooling fan failures. In the current software implementation, a warning is printed if an abnormal RPM is detected. An abnormal RPM reading is defined as one that deviates from a reference value by 30 % in either direction. The reference value is calculated from the PWM duty cycle setting using a linear approximation of the fan-specific RPM/PWM curve. The deviation threshold percentage was chosen to rule out practically observed variance in the RPM readings, and to allow some non-linearity in the RPM/PWM curve.

4.4.5. Interactive Usage

The software includes text-based command parsing facilities for interactive usage. The main purpose of command inputs is to manually access system registers. Other commands that influence the runtime state of the software were also included for testing and debugging purposes.

A register access command consists of three parts in one line of text:

1. One of the following letters: r, w, s, c, t
2. The register address: a hexadecimal number with 4 digits or less
3. The value to write: a hexadecimal number with 8 digits or less

The first part determines the type of operation. The letters in the first part stand for “read”, “write“, “set”, “clear”, and “toggle”, respectively. For read operations, the third part may be omitted. The three parts are separated with whitespace and all letters are case-insensitive. If a hash character (‘#’) is present, all subsequent characters on the same line are ignored. This allows comments to be inserted into a sequence of commands.

It is important to note that memory-mapped registers can be implemented with a large number of different access schemes. Moreover, several addresses can be associated with one physical register, so that several schemes can be used for that register. Therefore, one must be cognizant of the valid operations for any targeted address. A write operation, for instance, is obviously not valid for a read-only register. The operation types “set”, “clear” and “toggle” function on a “read-and-write-back” basis, and are thus valid only for read/write register access.

The register address is automatically word-aligned to 32 bits, i.e. the two least significant bits are treated as zeros regardless of the given address value. The address and value are both expressed in little-endian byte order.

The following are examples of valid register access commands:

```
w 4054 1
w 4058 F0BB1220
s 4058 0000FFFF
w 405c f0bb1220
c 405c ffff0000
r 4054
r 4058
r 405c
```

The three read operations should print the following values:

```
00000001
F0BBFFFF
00001220
```

To automate the initialization of the DFE system, it is practical to write sequences of initializing commands into a text file. The file contents can then simply be copied into a terminal emulator connected to the UART/USB interface.

4.4.6. Software Development Tools

The Xilinx Software Development Kit (SDK) is a software development environment for Xilinx embedded processors. Its features include a GNU-based compiler toolchain, JTAG debugger, and Flash programmer. Board support packages (BSPs) and drivers for Xilinx IP are also enclosed. The SDK is based on the open-source Eclipse platform, and incorporates the C/C++ toolkit. The SDK is available as a standalone installation and as a Vivado Design Suite installation package. [29]

The SDK was used for software development and BSP generation for the MCU subsystem. The JTAG debugger and Flash programmer were not used. The compiler toolchain incorporated in the SDK includes GNU-based compilers for the C and C++ programming languages. The compiler targeting MicroBlaze is called *mb-gcc*. Also included are the corresponding assembler (*mb-as*) and linker (*mb-ld*). [42]

The software build was configured to use the C99 standard of the C language. The highest optimization level (`-O3`) was used to maximize runtime performance.

4.5. Verification

The MCU subsystem design was entirely IP-based, so a conventional hardware verification testbench was not required. Basic verification of the block design was done with the “Verify Design” feature in Vivado before running the implementation flow.

The software features described in Chapter 4.4 were verified in two separate environments: the Linux development environment and the actual ADS7-V1 evaluation board. A separate software executable was compiled for each environment. The Linux build included several functional test routines for verifying that the software algorithms worked as specified.

5. EXPERIMENTS AND ANALYSIS

The MCU subsystem was synthesized and implemented both as a standalone design and as an integral part of the whole DFE design. The clock frequency was 125 MHz in both cases. In the standalone design, a "Clocking Wizard" IP block had to be added to the design to instantiate a single phase-locked loop (PLL), which was instantiated separately in the full DFE.

5.1. Implementation Results

Utilization figures from the fully integrated implementation run are presented in Table 3. Notice that some of the resource counts on the "Total" row are greater than the sums of module level counts. This is caused by resource instantiations on the MCU top level.

Table 3. Utilization results

Module	Slice LUTs	Slice Regs	F7 Muxes	F8 Muxes	Slices	LUTs as logic	LUTs as mem	BRAM tiles	DSPs
axi_intc_0	366	285			135	334	32		
axi_led_gpio_0	49	52			19	49			
axi_timer_0	334	216	1		114	334			
axi_uart16550_0	420	326			153	409	11		
mdm_1	1024	763	29		420	945	79		
microblaze_0	10150	10558	281	33	4337	9840	310	6	13
microblaze_0_axi_periph	709	511			290	668	41		
microblaze_0_local_memory	145	63			121	145		128	
rst_clk_125M	25	34			14	24	1		
Total	**13218**	**12808**	**311**	**33**	**5311**	**12744**	**474**	**134**	**13**

It is evident from the utilization figures that a clear majority of resources – over 80 percent – is reserved for the MicroBlaze processor core itself. This means any effort to decrease the resource utilization of the MCU should probably target the MicroBlaze core, which was originally parametrized to include a large part of its complete feature set, including extended debug features.

5.2. Maximum Clock Frequency

To estimate the maximum feasible clock frequency for the MCU subsystem, the standalone design was also synthesized and implemented with higher clock frequencies. The frequency was increased at 5 MHz increments until timing would fail after implementation. Similar experiments were not conducted with the full DFE due to the substantial computational effort required to implement it.

The highest clock frequency achieved with Vivado default synthesis and implementation strategies was 190 MHz. This exceeds the clock frequency of the WARP reference design presented in Chapter 2.5.1 by 30 MHz. This result was expectable since the Xilinx FPGA in the WARP v3 platform is from the older 6th series. 7th series FPGAs are used in the USRP X300 and X310, although they belong to the less performant Kintex family. The clock frequency for the ZPU processor core in the

USRP X300/X310 was about 166.7 MHz, which the MCU result still exceeds by over 22 MHz.

Several implementation settings were experimented with to push the timing margin further. This meant running the synthesis and implementation flow with different implementation strategies – predefined flow configuration profiles included in Vivado. First, synthesis was run with the *Flow_PerfOptimized_High* strategy. After that, a set of implementation runs was initiated with the following strategies:

Performance_RefinePlacement
Performance_ExploreSLLs
Performance_Explore
Performance_NetDelayHigh
Performance_Retiming
Performance_ExplorePostRoutePhysOpt
Performance_WLBlockPlacement
Performance_WLBlockPlacementFanoutOpt

No significant improvement was achieved with the experimented strategies.

When the number of BRAM tiles in *microblaze_0_local_memory* was doubled (resulting in 1 MB capacity), the highest successfully implemented clock frequency dropped to 170 MHz (with Vivado default strategies for synthesis and implementation.) This indicates that the routing associated with BRAM has a considerable impact on timing.

5.3. Software Build Results

The size of the entire software ELF file was 160 kB (163 883 bytes). The sizes of individual memory segments are presented in Table 4. The ELF file format also includes a header and a program header table, which is the reason for the larger size of the complete file compared to the sum of segment sizes [43].

Table 4. Software segment sizes

Segment	Description	Size in bytes
.text	Executable program instructions	54952
.data	Read-write data	1604
.bss	Uninitialized data (Total)	99636
	Stack	65536
	Heap	32768
	Other	1332
Sum		156192

To decrease the size of the ELF file, one simple option is to tell the compiler to optimize for code size. Changing the compiler optimization flag from `-O3` to `-Os` decreased the size of the .text segment by 1200 bytes. This decreased the size of the file by 0.7 %, which is insignificant for the purposes of decreasing BRAM utilization.

5.4. Software Gain Control Performance

No strict real-time latency constraint was specified for the gain control feature described in Chapter 4.4.2. Nevertheless, it is useful to gauge the real-time performance of the implementation for reference.

To measure the performance of the gain control calculation, a response time test was conducted on the actual hardware. A modified ELF file was compiled with invocations for starting and stopping the built-in timer during gain control calculation execution. The modified software also had an additional text-based command for invoking the calculation manually. The resulting timer values were printed through the UART/USB interface onto a terminal emulator while the DFE system ran on the ADS7-V1 board. Performance tests were run with various target power values while the measured power (a pre-set dummy value) remained constant. Hysteresis was disabled, which means the scaling coefficient was always updated.

Up to 1548 cycles were observed during the test run. This number needs to be adjusted to account for overheads caused by starting and stopping the timer. Moreover, the measured interrupt routine has a preamble and postamble, which are not included in the timer cycle count. The overhead incurred from starting the timer was 8 cycles, and the overhead for stopping the timer was 21 cycles. The preamble and postamble took 24 cycles each. A delay of 4 cycles is also incurred in the interrupt controller [44]. Further delays incurred in the data path gain control components are assumed to be no greater than 10 cycles. This sums up to $24 \times 2 - 21 - 8 + 4 + 10 = 33$ additional cycles that were not covered by the timer. Thus, the total delay of the gain control calculation routine estimated to be no greater than 1581 cycles. At a 125 MHz clock rate, this corresponds to $12.648 \approx$ **12.7 µs** in real time.

The result is about 59 % slower than the IEEE 802.11a/g latency requirement mentioned in Chapter 2.5.2 (8 µs). The calculation bypassed the hysteresis condition, which would increase the maximum response time further. Moreover, the calculation was performed for only one data path, whereas the referred AGC implementation incorporates two data paths, which means the worst case delay of the MCU implementation in a similar scenario would be almost 25.4 µs.

6. DISCUSSION AND FURTHER DEVELOPMENT

The software program running on the MCU was embedded into the FPGA configuration bitstream. This means any changes in the software require a new bitstream generation and configuration cycle. One way to improve on this is to embed a bootloader into the bitstream, and load the actual program code from a separate non-volatile memory module, such as an SD card. This could potentially decrease the time to develop and test new software versions on the target hardware. For perspective, it must be noted that a software update for a complete FPGA bitstream can still be done in minutes. This is considerably quicker than executing the full implementation flow for the FPGA design and generating a new bitstream, which could take several hours.

The amount of BRAM reserved for software purposes – 512 kilobytes – was more than sufficient for associated software image, which is discussed in Chapter 5.3. The amount is also larger than in any of the reference designs mentioned in Chapter 2.5. As BRAM is a specialized FPGA resource, placed among the configurable logic blocks of the device, large scale usage can cause difficulty with placement and routing. Halving the memory range to 256 kB should be a straightforward way to ease routing congestion and to free up BRAM for other uses, if necessary. Further BRAM downsizing would require optimization of the software memory footprint. A good place to start would be the uninitialized data (.bss) segment. The 32 kB heap memory range was not utilized at all in the final software revision, and stack memory could probably be downsized as well. Meanwhile, the size of the program instruction segment (.text) could be decreased by writing specialized versions of large library functions that have been linked into the software executable. One example of such optimization that has already been realized in the software program is `xil_printf()`, which is available in the software libraries bundled with Xilinx SDK. It is a downsized variant of the `printf()` function from the *stdio* standard library.

The UART output system was hindered by the lack of character FIFO state information, which is a known limitation of the instantiated UART IP block. This meant the processor had to wait for the FIFO to be empty before transmitting any subsequent characters. By giving the processor full read access to the state of the UART output FIFO software would no longer have to wait for each character to be transmitted, as long as the FIFO can accommodate all pending characters. This would require changes to the UART IP core.

The program running on the MCU was developed as bare metal software. Some tasks were interrupt-triggered, while others ran "in the background" when no interrupt routines were active. By adding a timer-triggered interrupt for background tasks (such as cooling fan control), the processor could enter sleep mode when no interrupts are pending. This could potentially decrease power consumption. Enabling sleep mode would require connecting the appropriate signals to the MicroBlaze *Wakeup* port.

The response time of transmitter gain control, which was discussed in Chapter 5.4, was shown to be longer than the reference implementation mentioned in Chapter 2.5.2. Several potential optimization strategies can be considered for improving the response time. The software subroutines involved in the calculation could also be optimized more for the MicroBlaze architecture – conversions between floating-point and fixed-point, for instance, could be implemented in a format-specific manner, at the expense of code portability. Moreover, the experiments discussed in Chapter 5.2 show that there is considerable room for increasing the clock rate. To meet the 8 µs latency

constraint with the measured number of cycles, the clock rate would have to be increased to $\frac{1581}{8\ \mu s} = 197.625$ MHz. The maximum experimentally demonstrated clock rate was 190 MHz, suggesting that a large part of the response time slack can be mitigated by simply increasing the clock rate. However, it is worth noting that 125 MHz was selected as the system-wide AXI clock frequency, and changing it may be unfeasible due to timing limitations in other subsystems of the DFE. Changing the MCU clock rate locally, on the other hand, would require clock domain crossings in the MCU AXI register bus. In that case, software performance scaling would not be entirely linear since accessing the system registers, which would remain in the slower clock domain, would cost additional clock cycles for the MCU. In fact, register access delay would increase, unless the local clock rate can be increased to an integer multiple of 125 MHz (250, 375 …).

To address the problem of servicing multiple data paths, a mutual synchronization scheme can be implemented in hardware to ensure that only one data path needs to be serviced at a time. Additional MicroBlaze cores may also be added into the design if a single core cannot service concurrent interrupts quickly enough.

As an additional comment on the gain control response time measurements, the number of cycles per calculation appeared to be dependent only on the input values; no unpredictable variation was observed. This is expectable, as the MCU subsystem has been designed in a manner that should result in relatively predictable runtime behavior in this scenario. Reasons for this predictability include bare metal software, BRAM as main memory, and absence of active interrupts during the test runs.

Also regarding the software implementation, the system error and warning routine (Chapter 4.4.3) was interrupt-triggered, although a polled routine would most likely be sufficient.

A newer revision of the MCU may incorporate a real-time operating system (RTOS) to enable a more advanced form of software task scheduling. This could be useful if more functional requirements are envisioned for the MCU. An RTOS may require virtual memory support to be enabled in the MicroBlaze processor core, as well as various changes to the software to adapt it to the new runtime environment.

7. SUMMARY

Digitalization has been a major trend in radio transceiver design since the 1990's. The replacement of analog-intensive circuitry with digital equivalents has enabled radio designers to exploit the continuous advancement of CMOS technology.

Radio transceiver architectures are classified by the number of distinct signal frequencies present in the signal path. A superheterodyne transceiver has at least one intermediate frequency (IF) stage, while a direct conversion or homodyne transceiver has no IF stages, which means the radio frequency (RF) signal is mixed directly onto the baseband in the receiver, and vice versa in the transmitter. The direct conversion architecture is the most common choice for modern digital transceivers.

This thesis has presented a soft-core processor based implementation of a control subsystem for a digital radio frequency front-end (DFE). In Chapter 2, the theory of radio front-ends was discussed, with focus on modern digital implementations. Chapter 3 then described the principles and practical considerations of embedded system design, especially pertaining to soft-core processors. Field-programmable gate arrays (FPGAs) were also described in detail, as they are a common implementation platform for soft-core based embedded systems.

The control subsystem was developed using the schematic-based design flow in Vivado IP Integrator, using Xilinx intellectual property (IP). The subsystem was designed around a single MicroBlaze processor core, with a set of peripherals implemented with other IP cores, and main memory implemented with block RAM. This design flow proved highly effective in realizing a control module with a programmable, general-purpose soft-core processor with soft real-time requirements. No HDL code had to be written manually as the design did not encompass any specialized, high-speed data processing, and the IP catalog in Vivado 2015.1 was sufficient for composing all parts of the design.

The whole DFE system was tested extensively on the ADS7-V1 board to verify its functionalities, including the tasks performed by the MCU. The response times of the automatic gain control calculation were also measured using an internal timer. Moreover, experimental synthesis runs were conducted for the MCU design to estimate its maximum feasible clock rate.

The goal of the thesis was to develop a soft-core processor system, and evaluate the correctness of the implementation. Automatic gain control calculation performance was also measured experimentally, but no maximum latency was specified explicitly. Another goal was to evaluate the IP-centric graphical design flow in Vivado.

Verification showed that all specified features were successfully implemented. The real-time performance of the automatic gain control calculation feature was also considered satisfactory for the current system, although its speed was insufficient to achieve the maximum latency of 8 µs, which was used as a baseline reference. Various opportunities for performance improvement were identified and discussed in Chapter 6. The graphical design flow was shown to be highly effective for developing soft-core processor based control modules for FPGAs. Creating and connecting the needed IP blocks was a quick and reasonably automatic process. Configuring the individual IP blocks was also partially automatic. Some pitfalls still remained, such as having to manually set the MicroBlaze interrupt vector base address after configuring the memory map. Once the hardware design was complete, appropriate software drivers were automatically configured to form a board support package (BSP) for the MCU.

8. REFERENCES

[1] Yiannacouras P., The Microarchitecture of FPGA-Based Soft Processors. MASc Thesis, Department of Electrical and Computer Engineering, University of Toronto, 2005.

[2] Grayver E., Implementing Software Defined Radio. Springer, 2013. ISBN: 9781441993311.

[3] Bowick C., Blyler J., Ajluni C. J., RF Circuit Design, Second Edition, pp. 185–188. Elsevier, 2008. ISBN: 9780080553429.

[4] Rudersdorfer R., Radio Receiver Technology: Principles, Architectures and Applications, Chapter 1. John Wiley & Sons, 2014. ISBN: 9781118503201.

[5] Staszewski R. B., Digital RF Architectures for Wireless Transceivers (Invited). 20th European Conference on Circuit Theory and Design, IEEE, pp. 429–436, Linköping, 2011.

[6] Hentschel T., Henker M., Fettweis G., The Digital Front-End of Software Radio Terminals. IEEE Personal Communications, Vol. 6, No. 4, pp. 40–46, August 1999.

[7] Mitola J. III, Software Radios: Survey, Critical Evaluation and Future Directions. Aerospace and Electronic Systems Magazine, IEEE, Vol.8, No.4, pp. 25-36, April 1993.

[8] Ulversøy T., Software Defined Radio: Challenges and Opportunities. IEEE Communications Surveys & Tutorials, Vol. 12, No. 4, pp. 531–550, Fourth Quarter 2010.

[9] Wireless Innovation Forum: Introduction to SDR. URL: http://www.wirelessinnovation.org/Introduction_to_SDR, retrieved on May 13th 2016.

[10] IEEE Std 1900.1-2008 – IEEE Standard Definitions and Concepts for Dynamic Spectrum Access: Terminology Relating to Emerging Wireless Networks, System Functionality, and Spectrum Management. IEEE Communications Society, October 3rd, 2008.

[11] Holland O., Bogucka H., Medeisis A., Opportunistic Spectrum Sharing and White Space Access: The Practical Reality, Chapter 1. John Wiley & Sons, 2015. ISBN: 9781118893746.

[12] Sabater J., Gómez J. M., López M., Towards an IEEE 802.15.4 SDR transceiver. IEEE International Conference on Electronics, Circuits, and Systems, Athens, December 12–15 2010, pp. 323–326.

[13] Khattab A., Camp J., Hunter C., Murphy P., Sabharwal A., Knightly E. W., WARP: A Flexible Platform for Clean-Slate Wireless Medium Access Protocol Design. Mobile Computing and Communications Review, January 2008, Vol. 12, No. 1, pp. 56–58.

[14] WARP Project. URL: http://warpproject.org, retrieved on February 29th 2016.

[15] USRP™ X Series, Ettus Research. URL: https://www.ettus.com/product/category/USRP-X-Series, accessed on April 26th 2016.

[16] USRP™ Networked Series, Ettus Research. URL: https://www.ettus.com/product/category/USRP-Networked-Series, accessed on April 26th 2016.

[17] USRP™ FPGA HDL Source, Ettus Research. URL: https://github.com/EttusResearch/fpga, accessed on April 22nd 2016.

[18] Bloessl B., Segata M., Sommer C., Dressler F., An IEEE 802.11a/g/p OFDM Receiver for GNU Radio. Proceedings of the Second Workshop of Software Radio Implementation Forum (SRIF '13). ACM, New York, NY, USA, pp. 9–16.

[19] Bloessl B., Sommer C., Dressler F., Power Matters: Automatic Gain Control for a Software Defined Radio IEEE 802.11a/g/p Receiver. IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), Hong Kong, April 26 – May 1 2015, pp. 25–26.

[20] Wilmshurst T., Designing Embedded Systems with PIC Microcontrollers: Principles and Applications. Elsevier Science and Technology Books, Inc., 2007. ISBN: 9780750667555.

[21] Wolf M., Computers as Components: Principles of Embedded Computing System Design, Third Edition. Morgan Kaufmann Publishers, 2012. ISBN: 9780123884367.

[22] Noergaard T., Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers, Second Edition. Newnes, 2013. ISBN: 9780123821966.

[23] Balarin F., Lavagno L., Murthy P., Sangiovanni-Vincentelli A., Scheduling for Embedded Real-Time Systems. Design & Test of Computers, IEEE, Vol. 15, No. 1, pp. 71–82, January – March 1998.

[24] Vahid F., The Softening of Hardware. Computer, Vol. 36, No. 4, pp. 27–34, April 2003.

[25] Maxfield C., The Design Warrior's Guide to FPGAs: Devices, Tools and Flows. Newnes, 2004. ISBN: 9780750676045.

[26] Tong J. G., Anderson I. D. L., Khalid M. A. S., Soft-Core Processors for Embedded Systems. 18th International Conference on Microelectronics (ICM), 2006.

[27] Zeidman B., Designing with FPGAs and CPLDs. Taylor and Francis, 2002. ISBN: 9781578201129.

[28] Quartus Prime Software, Altera Corporation. URL: https://www.altera.com/products/design-software/fpga-design/quartus-prime/overview.html, accessed on March 29th 2016.

[29] Vivado Design Suite User Guide – Embedded Processor Hardware Design (UG898), version 2015.4. Xilinx Inc., November 18, 2015. URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug898-vivado-embedded-design.pdf.

[30] UltraFast Design Methodology Guide for the Vivado Design Suite (UG949), version 2015.3. Xilinx Inc., November 23, 2015. URL: http://www.xilinx.com/support/documentation/sw_manuals/ug949-vivado-design-methodology.pdf.

[31] Wolf W., A Decade of Hardware/Software Codesign. Computer, Vol. 36, No. 4, pp. 38–43, April 2003.

[32] Kuon I., Tessier R., Rose J., FPGA Architecture: Survey and Challenges. Foundations and Trends® in Electronic Design Automation, Vol. 2, No. 2 (2007), pp. 135–253.

[33] Wong H., Betz V., Rose J., Comparing FPGA vs. Custom CMOS and the Impact on Processor Microarchitecture. FPGA'11, February 27 – March 1, 2011, Monterey, California, USA, pp. 5–14.

[34] Ahmed E., Rose J., The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 12, No. 3, pp. 288–298, March 2004.

[35] Kaeslin H., Digital Integrated Circuit Design. Cambridge University Press, 2008. ISBN: 9780511574276.

[36] Brown S. & Rose J., Architecture of FPGAs and CPLDs: A Tutorial. Design and Test of Computers, IEEE, pp. 42–57, Vol. 13, No. 2, 1996.

[37] ADS7-V1EBZ Overview, Analog Devices, Inc. URL: http://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/EVAL-ADS7-V1.html, accessed on October 20th, 2015.

[38] 7 Series FPGAs Overview (DS180), v. 1.17. Xilinx, Inc., May 27, 2015. URL: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

[39] Vivado Design Suite User Guide: Getting Started (UG910), version 2015.4. Xilinx Inc., November 18, 2015. URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug910-vivado-getting-started.pdf.

[40] MicroBlaze Processor Reference Guide (UG984), version 2015.4. Xilinx Inc., November 18, 2015. URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug984-vivado-microblaze-ref.pdf.

[41] 7 Series FPGAs Memory Resources – User Guide (UG473), version 1.11. Xilinx Inc., November 12, 2014. URL: http://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf.

[42] Embedded System Tools Reference Manual (UG1043), version 2015.4. Xilinx Inc., November 18, 2015. URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug1043-embedded-system-tools.pdf.

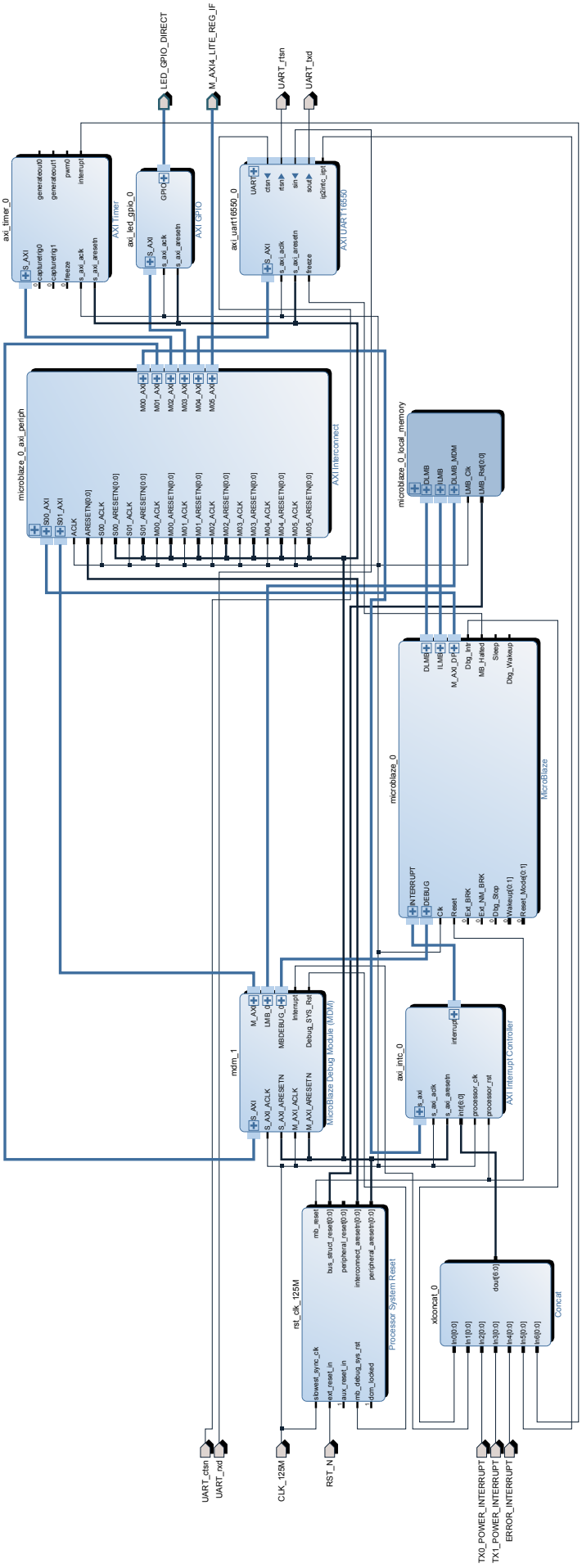
[43] Executable and Linking Format Specification, version 1.2. Tool Interface Standards (TIS) Committee, May 1995. URL: http://refspecs.linuxbase.org/elf/elf.pdf.

[44] AXI Interrupt Controller (INTC) – LogiCORE IP Product Guide (PG099), version 4.1, Xilinx Inc., April 6, 2016. URL: http://www.xilinx.com/support/documentation/ip_documentation/axi_intc/v4_1/pg099-axi-intc.pdf.

9. APPENDICES

Appendix 1. MCU block diagram

Appendix 1.


LED_GPIO_DIRECT
M_AXI4_LITE_REG_IF
UART_rtsn
UART_txd
axi_timer_0
AXI Timer
axi_led_gpio_0
AXI GPIO
axi_uart16550_0
AXI UART16650
microblaze_0_axi_periph
AXI Interconnect
microblaze_0_local_memory
microblaze_0
Microblaze
mdm_1
MicroBlaze Debug Module (MDM)
axi_intc_0
AXI Interrupt Controller
rst_clk_125M
Processor System Reset
xlconcat_0
Concat
UART_ctsn
UART_rxd
CLK_125M
RST_N
TX0_POWER_INTERRUPT
TX1_POWER_INTERRUPT
ERROR_INTERRUPT