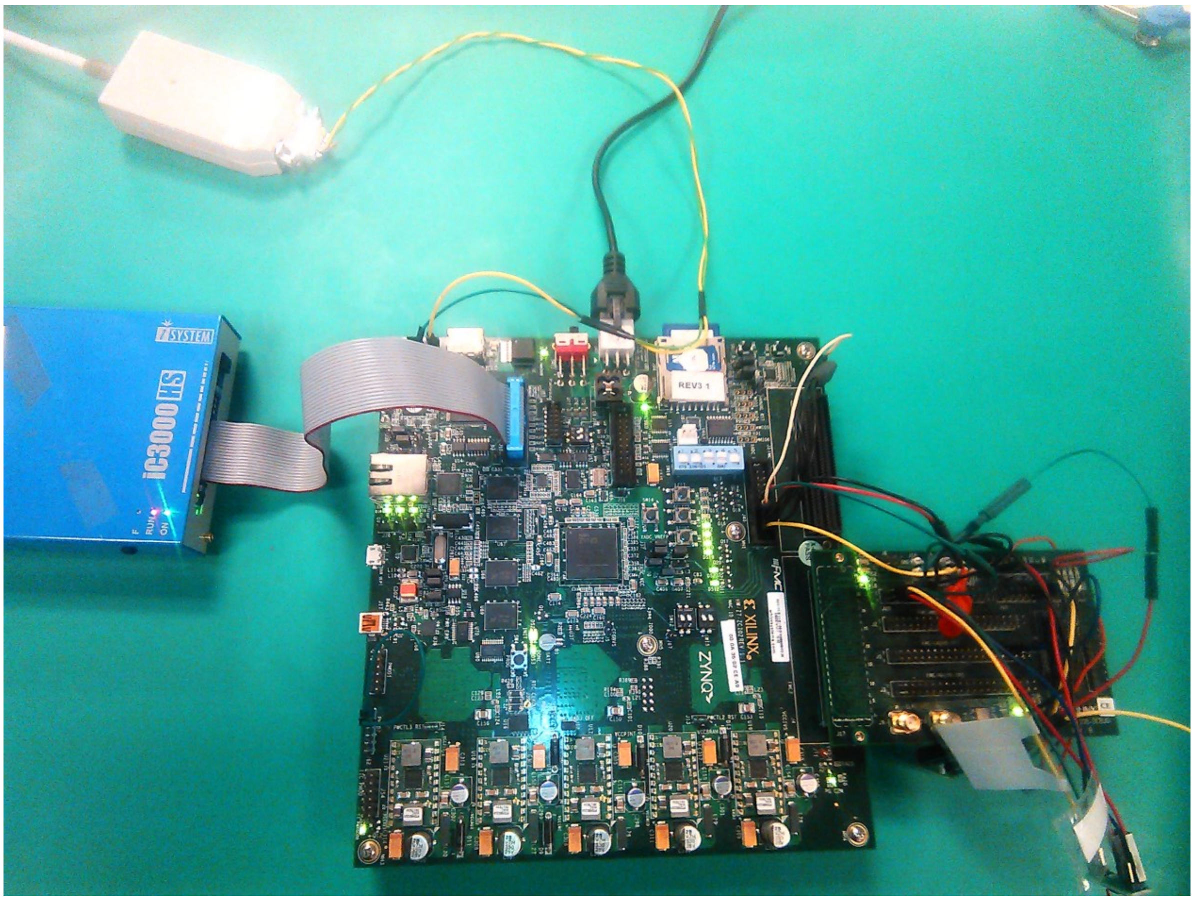


Feasibility Study of Implementing "*Reconfigurable Computing*" in *AUTOSAR* Environment

Master's thesis in Embedded Electronic System Design

VINODH RAJKUMAR GANESAN

Department of computer science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

Feasibility Study of Implementing
Reconfigurable Computing in AUTOSAR Environment

Examiner: PER LARSSON-EDEFORS

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Abstract

Technological advancements have always been a driving factor for growth in many domains. Competitive market and changing customer needs are a common scenario in the automotive sector. Many advancements in the recent past like the drive by wire system or the anti-lock braking system have become reality due to electronic control systems. With the increase in number of electronic control units within a vehicle, the standardization of vehicle's electrical architecture has gained significance. A standard that has been gaining recognition in the automotive sector over the years and that looks to be the future way ahead is the Automotive Open System Architecture (AUTOSAR).

We can expect future vehicles to have more electronics inside them and the electronic control units to be developed based on AUTOSAR standards. Advancements in communication technology have made it feasible to have data exchange between vehicles and by doing so, they have also increased the concerns on safety and security in vehicles. Further, development of various modern concepts like autonomous driving, etc., has pushed the automotive industry to gather lot of data and look for faster processing solutions. General purpose processors, which were previously used, alone are not competent enough to support the processing demands of these embedded applications and thus, wherever possible, hardware acceleration is being tried to aid the processor. But in doing so, the power usage and silicon area of the circuits increase, which is not viable. One way to solve this problem could be to introduce reconfigurable computing. *Reconfigurable Computing* is a form of computing in which the hardware configurations are modified during program execution. With this technique, the system becomes more flexible for the designers and by smartly selecting the hardware to be reconfigured, high throughput, less power usage and other system specific requirements can be attained. Another advantage could be after sale bug fixes like those which are possible on software today can also be extended to hardware.

Thus, if reconfigurable computing can be implemented within AUTOSAR, it will showcase the possibility to use this technology in vehicles. To understand the feasibility, a system that can dynamically and partially reconfigure an Advanced Encryption Standard (AES) encryption and decryption module in hardware according to diagnostic requests from the user was implemented using *Arctic Core* (an open source AUTOSAR software) for a Zynq® System On Chip (SOC). Results show the possibility of implementing such a system with few limitations. Further it could also be seen that a partially reconfigurable design would be more preferable than a fully reconfigurable solution as it could minimize the down time during reconfiguration of the system.

Keywords: AUTOSAR, Complex Device Driver, Reconfigurable Computing

Acknowledgements

I would foremost thank Mr. Johan Ekberg from Arccore AB for showing interest in the thesis idea without whom this work would not have been possible. I would like to express my sincere gratitude to Mr. Michael Lundell, my supervisor at Arccore AB and Mr. Mårtan Hilden for their guidance throughout this project. I am thankful to all the Arccore AB staffs for giving me a friendly environment and insights into various topics related to the thesis work, especially to Mr. Tomas Selldén and Mr. Staffan Johansson for sharing their expertise on the hardware.

My special thanks to professor Per Larsson-Edefors and my supervisor Mr. Sven Knutsson for their continuous support and patient reviews.

Vinodh Rajkumar Ganesan, Gothenburg, August 2016

Contents

List of Figures

List of Acronyms

AES	Advanced Encryption Standard
API	Application Program Interfaces
APU	Application Processor Unit
ARM	Advanced RISC Machine
AUTOSAR	Automotive Open System Architecture
Artop	AUTOSAR Tool Platform
ARXML	AUTOSAR XML
ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtended Interface
BSW	Basic Software
CAN	Controller Area Network
CDD	Complex Device Driver
CPU	Central Processing unit
DCM	Diagnostic Control Module
DDR	Double Data Rate
DID	Data Identifier
DMA	Direct Memory Access
DMAC	Direct Memory Access Controller
DRC	Design Rule Check
E/E	Electrical/Electronic
ECU	Electronic Control Unit
EMIO	Extended Multiplexed Input-Output
FPGA	Field Programmable Gate Array
FSBL	First Stage Boot Loader
GIC	General Interrupt Controller
GPIO	General Purpose input/output
GSM	Global System for Mobile communication
GUI	Graphical User Interface
hdf	Hardware Description File
ICAP	Internal Configuration Access Port
IDE	Integrated Development Environment
IO	Input Output
IP	Intellectual Property
ISO	International Organization for Standardization
JTAG	Joint Test Action Group

L1	Level 1
L2	Level 2
LED	Light Emitting Diode
LUT	Look Up Table
MIO	Multiplexed Input-Output
OBD	On-Board Diagnostics
OCM	On-chip Memory
OEM	Original Equipment Manufacturers
OSI	Open Systems Interconnection
PC	Personal Computer
PCAP	Processor Configuration Access Port
PDU	Protocol Data Unit
PL	Programmable Logic
PS	Processing System
RTE	Runtime Environment
RISC	Reduced Instruction Set Computing
SD	Secure Digital
SDK	Software Development Kit
SOC	System On Chip
SWC	Software Component
TCL	Tool Command Language
UDS	Unified Diagnostic Services
USB	Universal Serial Bus
VFB	Virtual Functional Bus
VHSIC	Very High Speed Integrated Circuit
VHDL	VHSIC Hardware Description Language
XADC	Xilinx ADC

1 Introduction

Automotive is a fast growing sector where the demands of customers are ever increasing and so is the need to adopt new technologies. A few decades ago, automobiles had simple electrical components but today most of the vehicles have advanced micro-controllers that run real time software[1] which can communicate within themselves and work collectively to improve the drive environment for the users. Several latest advancements like driver assistance systems, autonomous driving, vehicle to vehicle communication, etc., depend heavily on electronics and software. Also the amount of data stored and transferred within vehicles have increased in recent times. Modern day vehicles need advanced Electrical and Electronics (E/E) architectures suitable not only for handling technical challenges but that can also look into factors like passenger convenience, fulfill legal requirements, etc.[2]. A standardized environment for development of these E/E is thus inevitable in such a scenario to manage modifications with better quality. This leads the manufacturers to adopt an open software architecture called AUTomotive Open System ARchitecture (AUTOSAR) [3].

In-order to improve security and safety, it is expected that the Electronic Control Unit (ECU) inside vehicles have some reconfiguration capability [4]. It is well known that Field Programmable Gate Array (FPGA) have the special characteristic that they can be configured and reconfigured after being manufactured. Modern FPGAs have significantly more options like partial reconfiguration, which means that some connections inside the FPGA can be partly changed. Moreover partial reconfiguration can be done without switching the power off. Thus when FPGAs of automotive quality are coupled with AUTOSAR, such that the software in the system is capable of controlling the reconfiguration of hardware, there can emerge a wide range of benefits like hardware re-usability, after-sales modification of circuits and many more.

1.1 Background

AUTOSAR is a union formed by leading Original Equipment Manufacturers (OEMs) and the tier 1 suppliers in the automotive sector. Their objective was to define an open standard upon which automotive E/E can be built. AUTOSAR makes it mandatory that the software is developed in a layered architecture with standardized Application Program Interface (API) and thus ensuring portability and hardware independence for the OEM's software [5].

Reconfigurable computing is a widely researched topic in the high performance computing field. The significance of this approach is that it can combine the benefits of both general purpose processors and Application Specific Integrated Circuit (ASIC) [6]. When there is a computationally intensive task like signal processing or encryption/ decryption, it would be a good option to execute them in specialized hardware units rather than on general purpose processors. In doing so, not only the output is obtained faster but the processor is also allowed to execute other tasks thereby increasing the overall throughput of a system.

Modern System On Chip (SOC) have, within them, processor cores and configurable logic blocks and these even support partial reconfiguration [7]. The fact that these SOCs are available in automotive grades opens up the possibility to exploit the benefits of reconfigurable computing within automotive domain. As more vehicle manufacturers are leaning towards AUTOSAR, if reconfiguration of the hardware could be controlled through AUTOSAR software, it would be suffice to show that reconfigurable computing is feasible wherever needed in the automotive domain.

1.2 Aim

The goal of this thesis would be to show the possibility of implementing an ECU which can partially reconfigure an AES encryption and decryption hardware according to requirements, without affecting its other functionalities. The ECU's partial reconfiguration should be controlled by AUTOSAR software running on the same chip based on diagnostic requests from the user.

1.3 Limitation

The diagnostics implementation in AUTOSAR is broad. The standard supports a wide range of communication protocols and there are diagnostic services supported for each of them. To maintain simplicity, only requests of the below mentioned services through Controller Area Network (CAN) will be considered. The numbers refer to a service identifier and *service* here represents a request to the ECU from the user (or tester) in CAN message with these identifiers in specific locations.

- 0x23 Read memory by address
- 0x3D Write memory by address
- 0x27 Security Access

Although the implementation for the thesis is limited to these few services, the concept can easily be extended to other services and protocols supported by AUTOSAR.

2
Theory

This chapter gives a small introduction about different software concepts and the hardware used in the thesis.

2.1 Open Systems Interconnection

When computing systems need to be connected in a network, there must be a set of rules defined. The International Organization for Standardization (ISO) - Open Systems Interconnection (OSI) is a reference model based on which interconnection standards can be developed [8]. It consists of seven layers namely application, presentation, session, transport, network, data-link and physical as shown in Figure 2.1.

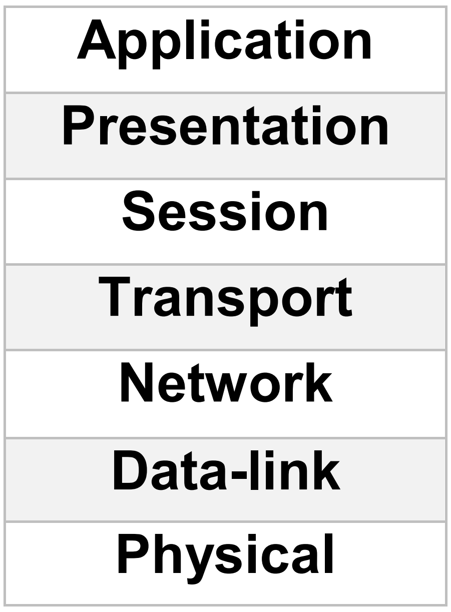


Figure 2.1: The OSI model

According to Mohammed M. Alani each layer handles data differently [9]. He further explains that if Protocol Data Unit (PDU) is considered the unit of data in a layer, then PDU for the physical layer is raw bits and the functionality of this layer will be to aid in the data transmission between data-link layers of the sender and receiver. The PDU in the data-link layer would be frames and the data link layer will have a wide range of tasks to perform like flow control, error detection, etc. The network layer's PDU is packet. This layer is useful in maintaining communication within a network and controls the routing of messages between networks. The next level is the transport layer with PDU as segment which is helpful in segmenting big data for transmission according to the bus or decoding several packets of data into segments for the above layers. Neither the session layer nor the presentation layer, which are the next two in hierarchy manipulates the data any further. While the former is involved in establishing communication sessions between different units, the latter manages the method in which the data is given to the application. Finally the application layer helps the user communicate with the network.

2.2 Automotive Open System Architecture

AUTOSAR is a union of OEMs and other E/E suppliers within the automotive industry [10]. The members of AUTOSAR are brought into a three tier structure namely the **Premium partners**, the **Development partners** and the **Associate partners** sharing different rights and responsibilities [11]. The union works on three major tasks [12],

- Defining the standardized work flow model called the *AUTOSAR methodology* that can aid in sharing of tasks during the development period
- Defining Software architecture
- Defining the different interfaces between the basic software and the application layer

The standard has been continuously evolving since the AUTOSAR release 1.0 in 2005. Such a standardized architecture is supported by OEMs in a view that in the future, their application softwares can be developed independent of the hardware. Also, because of this approach, software module(s) developed for one vehicle model can easily be reused in other models and application software designed to run on a particular ECU can be shifted without much difficulties to another.

AUTOSAR prescribes a layered software architecture. A piece of code with some functionality is called a module and there can be several modules within a layer. The architecture defines the limitations and extent of accessibility for different modules. In doing so, the impact of changes in the code is limited and new software modules can henceforth be added easily without affecting the quality of the other software. This directly reduces the need to test the already existing modules and in-turn can reduce the time to market and the cost of development for a new ECU.

Looking at the benefits, many of the automobile manufacturers have already started to develop their software based on AUTOSAR [13] and the standard is gaining acceptance world wide. The number of ECUs based on AUTOSAR is estimated to rise from 25 million in 2011 to 300 million in 2016 [14] and this figure directly explains the standard's significance.

2.2.1 AUTOSAR Methodology

In-order to achieve a model where software development can be done without the knowledge of underlying hardware, few procedures in the system development are to be followed and these are known as the "AUTOSAR Methodology" [15]. According to this, development begins by defining the overall functionality of the software in a vehicle. Then from these functionalities, requirements specific to the system, ECUs and different components of basic software are derived. The derived information is exchanged between different levels in a standard file format called AUTOSAR XML (ARXML). Since the flow is standardized, each part of the system can be developed separately and when adopted properly, different parts of the same software can be developed by various vendors. At some point when individually developed softwares are to be integrated, these ARXML will remain as the reference.

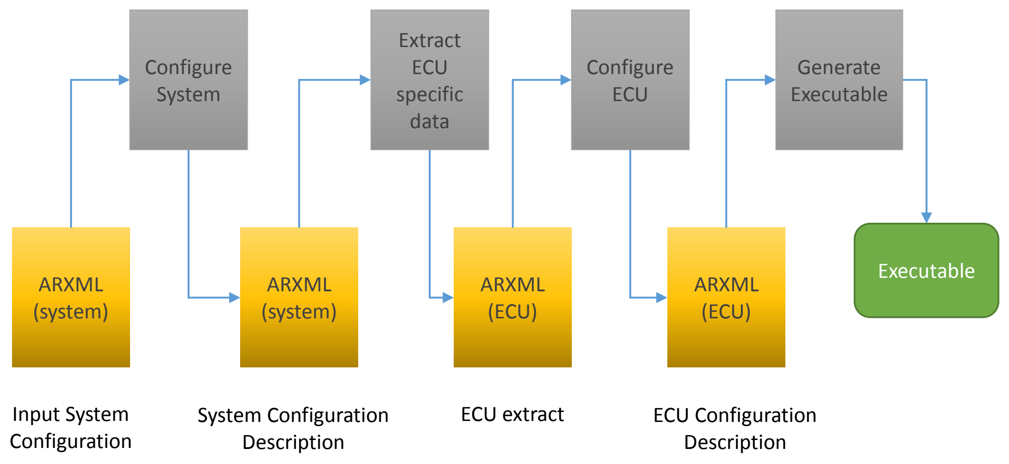


Figure 2.2: AUTOSAR Methodology

An overview of AUTOSAR methodology according to the standard [15] is shown in Figure 2.2. All the yellow boxes are either input or output of the system in ARXML format and the gray boxes are the tasks performed. As shown, the work flow begins with an ARXML file consisting of the system configuration details such as overall software component requirements in the system, ECU resources, bus topology, etc. Based on this input the *configure system* task allocates the application to ECUs, selects the bus topology and performs similar system level mappings and generates another ARXML files consisting all these details. This file now becomes the input to the next task in the work flow, *Extract ECU Specific Data* task, where details specific to individual ECUs are generated. An ARXML output named *ECU extract* gets generated and this serves as the input to the next task *Configure ECU*. During this task, based on the details from the ECU extract, finer details like BSW configuration, etc., needed for the development of software at the ECU level are derived and the output for this stage is the *ECU configuration Description* which serves as the base for the software development. The next process is the *Generate executable* task which generates the executable for a particular ECU.

2.2.2 AUTOSAR Software Architecture

The AUTOSAR software architecture mainly focuses on vehicle ECUs which use 16/32 bit micro-controllers and perform tasks at real time based on sensor inputs and other similar ECUs connected in networks [16]. Use of multi-core Central Processing unit (CPU)s is on the rise and the present AUTOSAR software architecture can be extended to support them.

As shown in Figure 2.3, for an ECU with single core, the software primarily consists of three layers namely

- **Basic software (BSW)**
- **Application layer**
- **Runtime Environment (RTE)** also referred as **Virtual Functional Bus (VFB)**

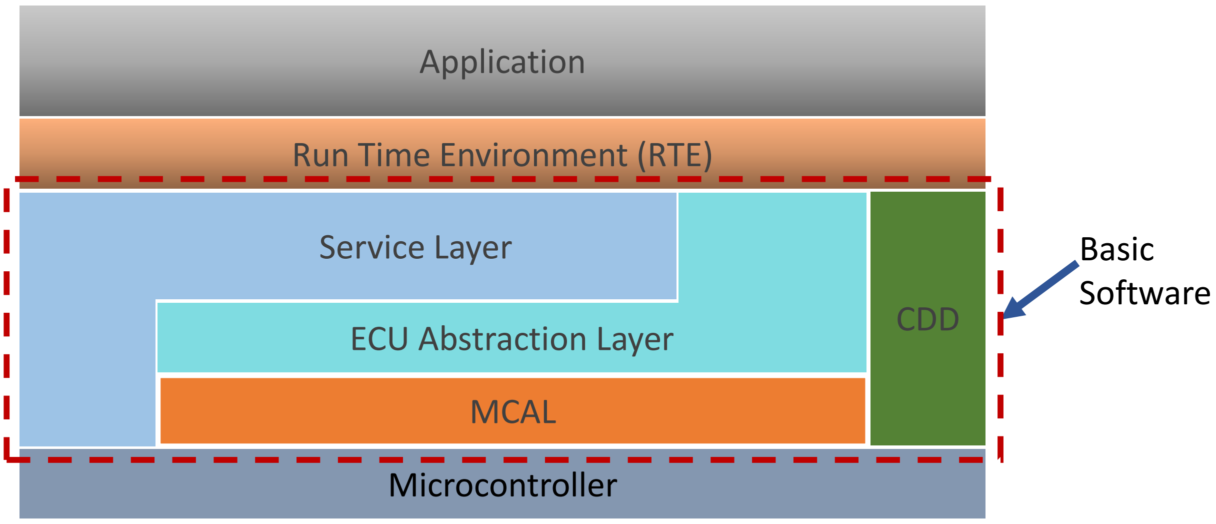


Figure 2.3: Basic structure of AUTOSAR

2.2.2.1 Application layer

During requirement analysis, all the basic tasks a system should perform are called functional requirements [17]. In the application layer modules performing algorithms to accomplish these functional requirements are placed. The code in this section does not need to know how the system is configured to achieve this. For example, a requirement for the system could be, on a particular condition it should switch on a lamp (indicators) and blink at a particular rate. The code in this section will only control the algorithm for switching on and off the lamp, but details like how this lamp is connected or which port in the micro-controller needs to be switched on and off to achieve this and how the time is calculated are all abstracted.

2.2.2.2 Basic Software

The BSW layer is made up of the service layer, ECU Abstraction layer and the micro-controller abstraction layer. These sections handle the requests from the application layer. As mentioned in the above example, the switching on and off of a lamp could be the request. The basic software layer has in it those finer details like how the lights are connected, which port(s) in the micro-controller needs to be switched, what is that value when assigned to the port will switch the light on/off, calculation of time according to the hardware oscillators that is used, etc., and thus help in achieving the functional requirements of the system.

2.2.2.3 Runtime Environment

The *RTE* links *BSW* with the *application layer*. The common references could be variables in memory. In events like hardware modifications, now it becomes sufficient to modify only the RTE and the application layer code can largely be maintained unaffected.

2.3 Diagnostics

Software inside a car has multiple functions. One among them is to monitor the functioning of various units in the vehicles. This feature is called diagnostics and they help the technicians, manufacturers and vehicle owners to know crucial details of various sub-systems [18]. The ECUs inside a vehicle are interconnected for data exchange. They could be connected in the same or different networks. AUTOSAR describes two major diagnostic standards namely

- **Unified Diagnostic Services (UDS)** [19]
- **On-Board Diagnostics (OBD)** [20]

The former is based on ISO standard *ISO14229-1* while the later is based on ISO standard *ISO15031-6*. They lie in the application layer of the OSI model.

To read/write information from ECUs, software allows users to send messages in a pre-defined format. These predefined messages are called diagnostic requests and are characterized by a unique number called *Service Identifiers (SID)*. This distinguishes which service is being requested by the user from the ECU. The information sent back by the software is called a diagnostic response and will also posses a corresponding number. If the request message was supported by the ECU and if the software was able to collect the required information then the ECU sends back the read data with a valid response code to the user. In-case the request is not valid or if the ECU had not been able to process the request within a given time, it sends back a negative response.

2.4 Controller Area Network

The Controller Area Network **(CAN)** is a serial communication protocol within a bus network [21]. It is a widely used protocol in automotive industry for communications between ECUs because of its cost effectiveness in comparison with other protocols for a bit rate of 1 Mbs^{-1}.

2.5 AES

Cryptography is widely used for data security during transmission and storage. Various algorithms exist to do this. The National Institute of Standards and Technology (NIST) [22] is a federal agency within the U.S. Department of Commerce involved in defining standards. In the year 2011, they announced the *Rijndael algorithm* [23] as the **Advanced Encryption Standard (AES)** [24].

2.6 Reconfigurable Computing

The process of using or creating an algorithm to get a desired output is called computing [25]. In electronics, computing is achieved with the help of circuits. If these fundamental circuit configurations are changed within the architecture during run-time, for various reasons like speeding-up the computation, etc., then we obtain reconfigurable computing. Thus this form of computing is heavily dependent on the underlying hardware. Reconfigurable computing is a widely researched topic in the high performance computing domain [26].

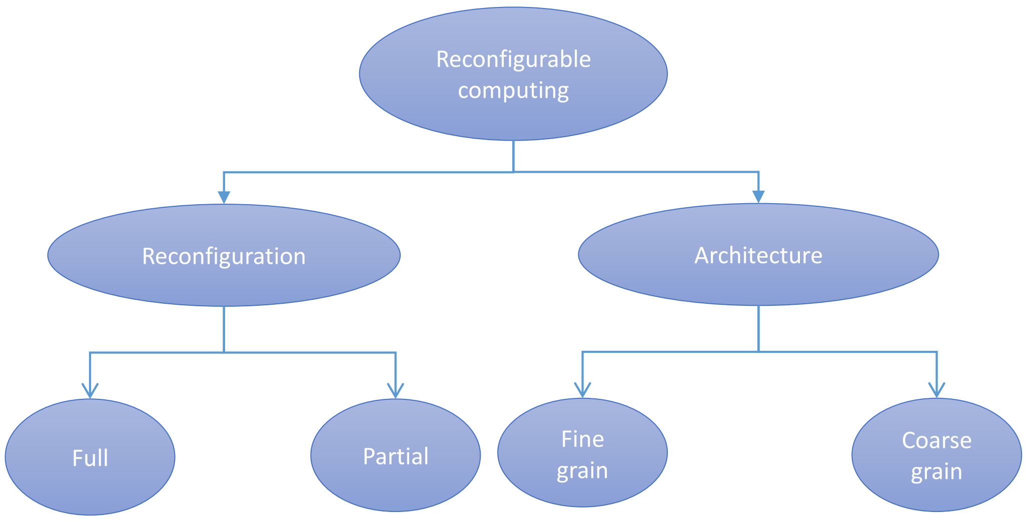


Figure 2.4: Reconfigurable Hardware types

Based on the extent of reconfiguration on circuits, a system can be classified as fully or partially reconfigurable system. During computation, if the whole hardware is to be reconfigured, then the system is called a fully reconfigurable system and on the other hand, if the system has the capability to reconfigure only a few parts of its whole circuitry, then it is called a partially reconfigurable system. Another way of classifying these systems is the smallest level of reconfiguration attainable called granularity. A system where the reconfiguration could be extended to the smallest unit available, like those in FPGAs, are called fine grain systems and those, in which a big circuitry using few basic units is replaced by another big circuit, are called coarse-grained reconfigurable systems [27].

2.7 Zynq®

The growth in computing requirements within embedded systems has opened up the need for using high performance computing techniques. Zynq® is a SOC from Xilinx with a general purpose processor cores and a FPGA. Further, Zynq® SOCs are marketed as different variants like 7z010, 7z020, etc., and in this thesis work 7z020 variant is used.

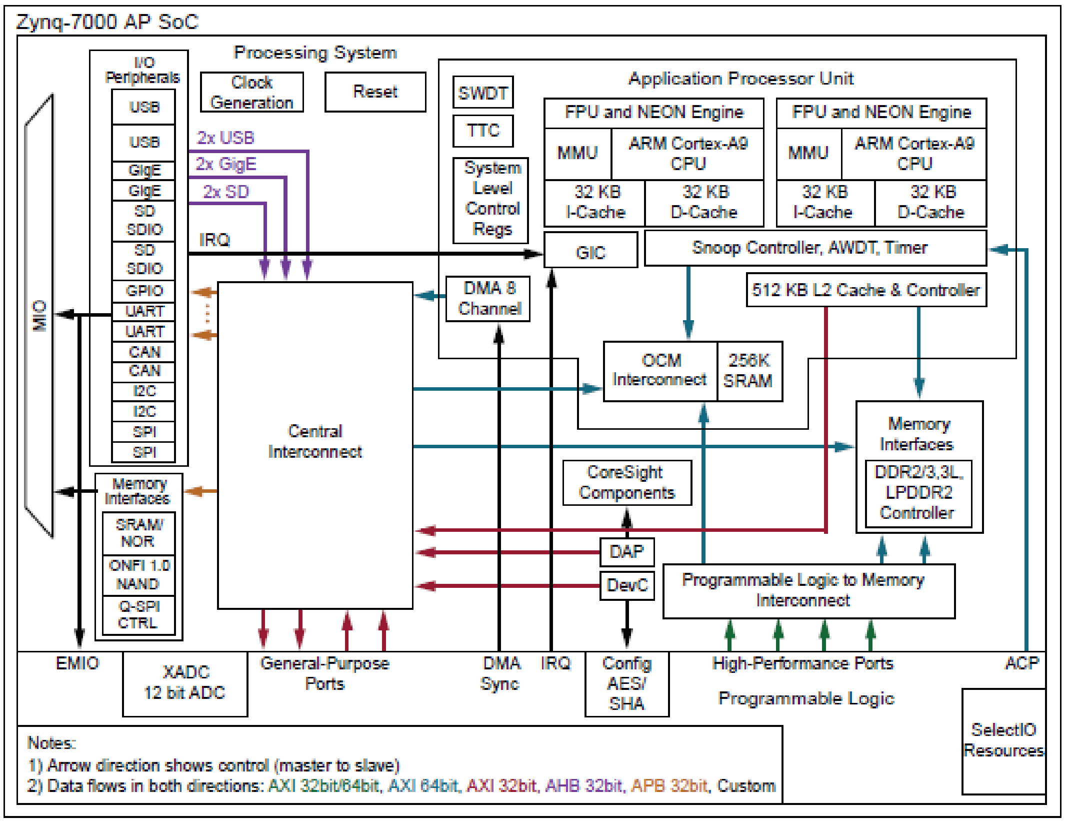


Figure 2.5: Block Diagram of Zynq®

Source: Zynq-7000 All Programmable SoC Technical Reference Manual UG585 (v1.10)

The section holding processing cores and supporting circuits is called a processing system (PS) and the section with FPGA and reconfiguration supporting circuits is called a programmable logic (PL). The block diagram of the SOC can be seen in Figure 2.5. Based on the technical reference manual for Zynq® from Xilinx [28], only those units which are of interest to this thesis, are mentioned below.

2.7.1 Processing System

The PS is divided into four major sections namely the **Application Processor Unit (APU)**, **memory interfaces**, **input-output peripherals** and **interconnects**. The APU comprises of two ARM Cortex™ A9 processing cores with 32 KB instruction and data Level 1 (L1) caches. It also has a 512 KB sharable Level 2 (L2) cache with snoop control units to maintain coherency between the L1 and L2 caches. In addition to these the PS also contains 256KB of on-chip memory. The Direct Memory Access (DMA) controller that controls the data transfer between memory and PL/PS independent of processor interventions, the General Interrupt Controller (GIC) that triggers or masks ARM defined interrupts and the watch dog timers are all part of the APU.

The memory interfaces supports Double Data Rate (DDR), quad-SPI, NAND technologies with special controllers. The input-output peripherals within PS consists of configurable General Purpose input/output (GPIO)s, a Secure Digital (SD) controller, CAN controllers, etc., In the interconnects section, there are central interconnects that can connect PL to the external peripherals, OCM interconnect that can connect PL and central interconnect to the OCM and PS-PL interfaces with Processor Configuration Access Port (PCAP) that support CPU controlled reconfiguration of PL and Extended Multiplexed Input-Output (EMIO).

2.7.2 Programmable Logic

The PL section within a 7z020 device is based on Xilinx's Artix®-7 FPGA logic. It comprises of the device configuration module (devc) that helps configuring or reconfiguring the FPGA from the PS logic. It also has the Xilinx ADC (XADC) that helps in monitoring voltage and temperature inside the device are present also available in addition to the FPGA logic within the PL section. Interconnections between the PL and the PS is supported through the Advanced eXtended Interface (AXI). The major resources available in the 7z020 FPGA logic can be seen in Table2.1.

Table 2.1: Composition of FPGA in 7z020 device

Sl No.	Resource	Size
1	Logic Slices	13300
2	Look-up Tables	53200
3	Memory Resources Block RAM	560 KB 140
4	I/O bank count	4

3 Methods

In this chapter, the software and hardware selection criteria and the intended feature addition as of the thesis work are discussed.

3.1 Software and Hardware Selection

The initial work in the thesis was to find a platform to start the work. It involved selection of the hardware and software to be used. The software to be used should be in compliance with AUTOSAR and the hardware to be chosen needs to have capability for both running an AUTOSAR software and supporting reconfigurable computing.

3.1.1 AUTOSAR Software

The AUTOSAR consortium works only on defining standards for the software and does not force the implementors to develop their solutions using any particular method. This strategy is followed in-order to encourage competition and innovation [2]. As a result, there are 76 vendors (as of April 08, 2016) who can deliver AUTOSAR based products [29]. It is also important to note that these products are mostly proprietary.

Arccore AB [30], a Swedish company is one of the leading vendors in the automotive software market. Their product *Arctic core* comprises a complete embedded platform based on AUTOSAR. Further *Arctic core* is released under both GNU General Public license v2 and a commercial license making it best suitable for the thesis work.

3.1.2 Hardware

As explained in section 2.7, Zynq® is a System On Chip (SOC) from Xilinx with a unique design. It has a PS with two ARM® processing cores and a PL that can be used to implement hardware designs. Zynq®has hardware that can support partial reconfiguration of the PL. Arctic core already supports Zynq® and thus it became the natural choice for this work.

3.2 Intended System

As mentioned earlier *arctic core* supports Zynq® architecture, but only to the extent that AUTOSAR software can be run on its PS. There is no option to control partial or complete reconfiguration of PL with PS.

With the focus to showcase the capability of AUTOSAR software controlled reconfigurable computing, a simple system whose hardware configuration in PL can be modified between 128 bit AES encryption or decryption algorithm according to diagnostic requests from the user that will allow users to write into or read from a limited range of memory addresses after establishing a secured session is intended. The AES will be memory mapped and the software will know this address. The logic of the system is shown in Figure 3.1.

- The AUTOSAR software must know the initial hardware settings in the PL and it should keep monitoring for the diagnostic requests from the user.
- When there is a diagnostic request from the user requesting security access (service 0x27), there will be a number sent to the user as response known as *Seed*.
- If the user replies to that *Seed* with a matching predefined number called *Key*, the software will enable access to the previously inaccessible memory regions. This access will be available to user only for a fixed amount of time and the ECU identifies this as a secured session.
- If needed the secured session can be extended for a predefined time by a tester using *tester present* requests.
- When the secured session is established and if a correct memory read request (service 0x23) for a particular address in the memory is sent, the software will check for the configuration available in PL. If the present circuit is for AES encryption, then the software should reconfigure the hardware to AES decryption first and then try to read the data from it. Otherwise the software should directly read the value from AES. The AES unit, in-turn should read the corresponding memory location. When the memory address requested is accessible, the software should respond to the tester with a positive response along with the read data (in decrypted format). In all other cases there should be a negative response.
- Similarly, when the secured session is established and if a correct memory write request (service 0x3D) for a particular address in the memory is requested, the software will check for the configuration available in PL. If the present circuit is for AES decryption, then the software should reconfigure the hardware to AES encryption initially and then try to write into it. Otherwise the software should directly try to write into AES. The AES ciruit will then write the data into the corresponding memory location. When the address requested is accessible and write is performed, the software should respond with a positive response to the tester confirming that the write was successful. In all other cases there should be a negative response.

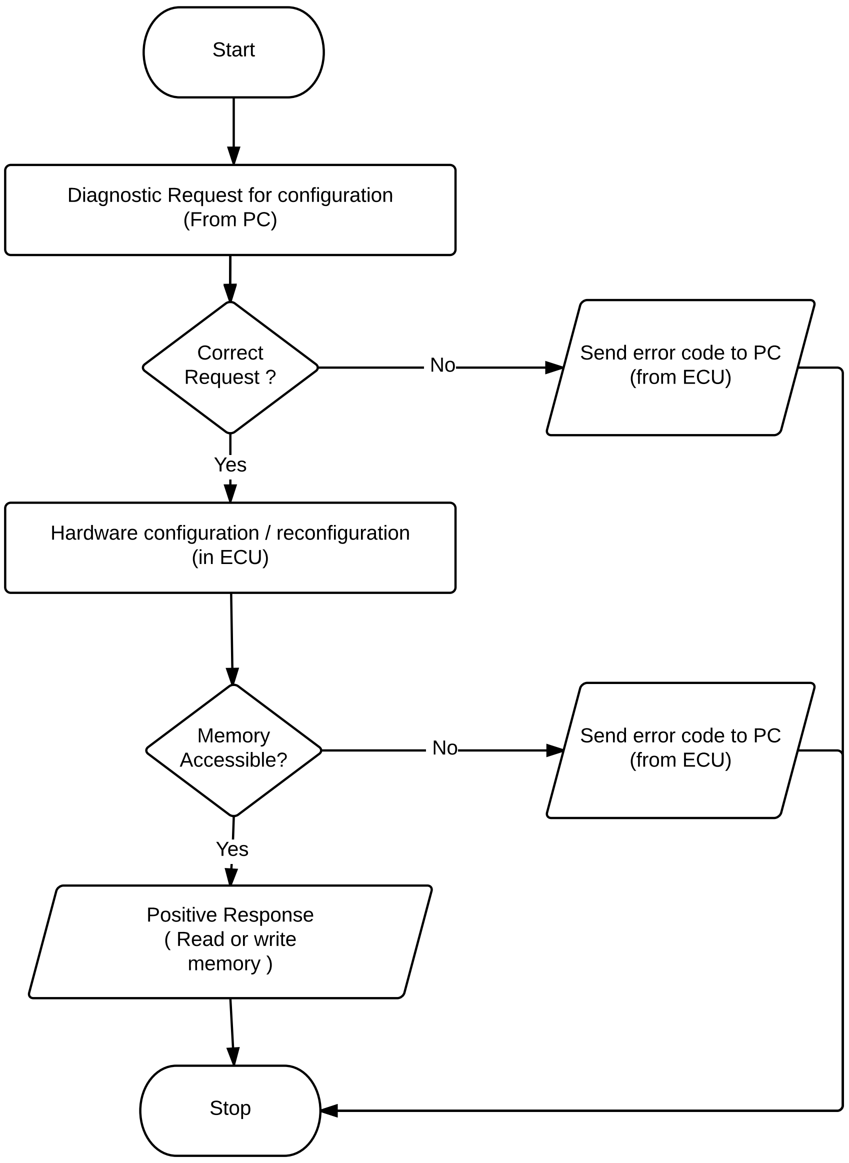


Figure 3.1: Intended Design Flow

Implementation of the above idea can be separated into hardware and software development. While the software part involves configuring an AUTOSAR ECU with necessary functionalities, the hardware part involves designing 128-bit AES encryption and decryption units. The procedures followed are described in detail in the following chapters.

4
Software

The software design started with understanding the basics of the AUTOSAR architecture, learning how *arctic core* implements this and usage of *arctic studio*. It also involved exploring AUTOSAR methodology and understanding the usage of tools for this purpose. As a part of the software development two major points were to be considered.

- Firstly, the software should be capable of supporting CAN diagnostics and it should respond to only specified requests.
- Secondly, there should be a logic within the software that can verify the present hardware configuration in the *Zynq®*'s PL and perform a reconfiguration if needed.

The chosen AUTOSAR software (*arctic core*) is capable of running on different hardware and it can also support CAN diagnostics. But it has to be configured according to the project requirements. Further to bring in reconfiguration of hardware, the software needed a Complex Device Driver (CDD). In this chapter the details of tools used in software designing, the need for a CDD, various considerations and strategies adopted in CDD development along with the reasons for adopting such a flow are explained in detail.

4.1 Development method

AUTOSAR defines a standardized way of defining and sharing tasks during software development. As a result of such a methodology, simultaneous development of various software components becomes possible. Further multiple vendors can be involved in the development of these components. In this section, an introduction to the *AUTOSAR development method* and the application of it from the thesis perspective is presented.

4.1.1 Design Strategy

In the case of this thesis work, the focus is at the ECU level and the objective is quite straight forward that the designed software must respond to selected diagnostic requests from the tester and then if needed control dynamic reconfiguration of *Zynq®* hardware. Arccore AB's *Arctic Core* software is used as the basic software. *Arctic Core* is compliant to AUTOSAR 4.0 and has in it all the basic components of an AU-

TOSAR software. Further, *Arctic Core* provides basic support for the *Zynq®* SOC. However at present, the software does not have the capability to exploit *Zynq®*'s reconfigurable computing feature. Thus for the proposed system, software design should involve

1. Configuation of *Arctic Core* such that it responds to needed diagnostic requests.
2. Additional code in *Arctic Core* that allows it to support reconfiguration of *Zynq®*'s PL.

4.1.1.1 Simple Example

According to AUTOSAR standards, to begin ECU development, an ECU extract that contains information regarding all the basic configurations required at the ECU level is needed. When *Arctic Core* software is downloaded, it contains a set of example configurations. The ***Hello World*** is one such simple example for a basic ECU and it consists of a configuration ARXML file (HelloWorld_Generic.arxml), resembling an ECU's requirement derived from a full system. It also has an application module that can control the blinking of one or more Light Emitting Diodes (LEDs). In-order to simplify things, except for a few features most of the others are disabled using the configuration file. The *Hello World* example can be run on various boards after generating executables with corresponding configuration files for different hardware. So, the example configured for *Zynq®* hardware was chosen to be the base for this thesis work.

4.1.1.2 Configuring *Arctic Core* for diagnostic requests

The Diagnostic Control Module (DCM) exists in the communication layer of the AUTOSAR architecture and is responsible for monitoring diagnostic requests and permitting the ECU to service these requests based on the present session and diagnostic states [31]. Analysis showed that, within the present software configuration, it would be suffice to configure only the DCM to achieve services for all the intended diagnostic requests.

4.1.1.3 Choice of a Complex Device Driver

When there is a need to introduce new concepts in AUTOSAR, a non standardized software entity called Complex Device Driver (CDD) can be used. The CDD, as any other software component can interact with basic software modules of the AUTOSAR with few restrictions as mentioned in [32]. The concept of reconfigurable computing requires quick access to hardware and further it is not standardized by AUTOSAR. Thus a CDD that can monitor and control reconfiguration was added to the design.

4.2 Design

The requirement to configure the DCM module was straight forward. But the design of a CDD is not standardized by AUTOSAR and the Zynq® architecture supports hardware reconfiguration in different ways. As a result, several solutions were found feasible to achieve the intended system. During the design of the CDD, two major questions were to be answered.

1. Where should the CDD be placed in the design?
2. How should the CDD control the hardware reconfiguration?

4.2.1 Location of CDD

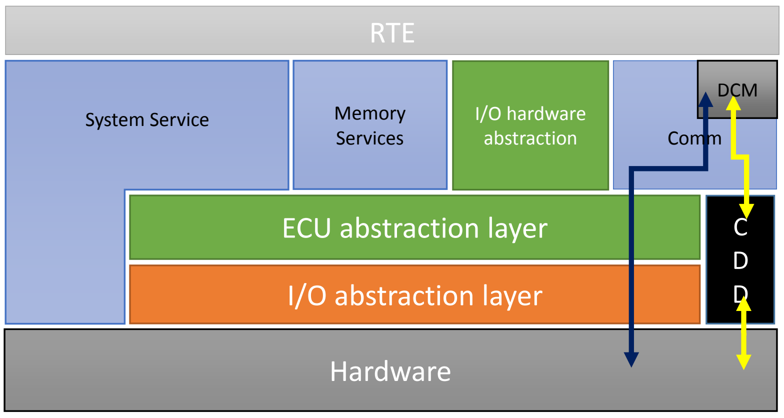
Based on requirement, the CDD should control reconfiguration of hardware based on diagnostics which is handled by DCM. This means there should be a way to establish communication between CDD and DCM. To achieve this, three different configurations were considered and these are shown in Figure 4.1. The Arrows indicate the flow of information or requests.

Option (a) shows the possibility of implementing the CDD within Basic Software (BSW). Under this design the communication between CDD and DCM is direct.

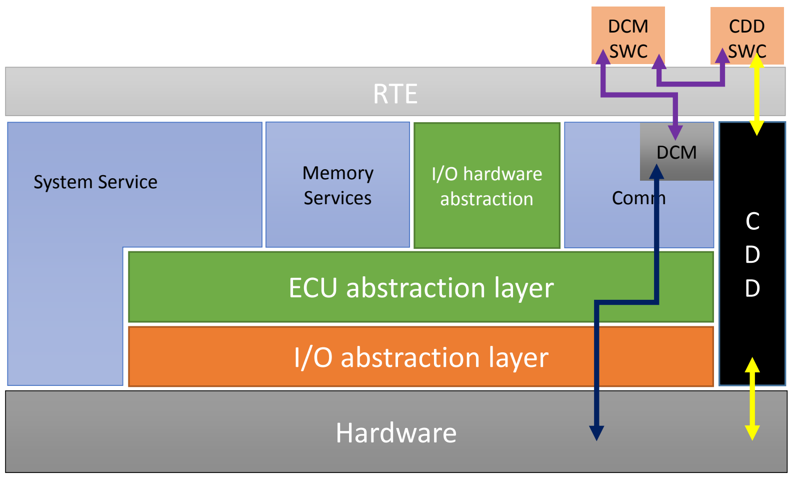
Option (b) shows a design to implement the CDD as a BSW and design a software component for both CDD and DCM. The communication between CDD and DCM will in this case be through Runtime Environment (RTE) with the aid of implemented Software Component (SWC)s .

Option (c) shows a design with CDD as a single SWC. There is also a DCM software component to be implemented. The communication between CDD and DCM here will also be through the RTE.

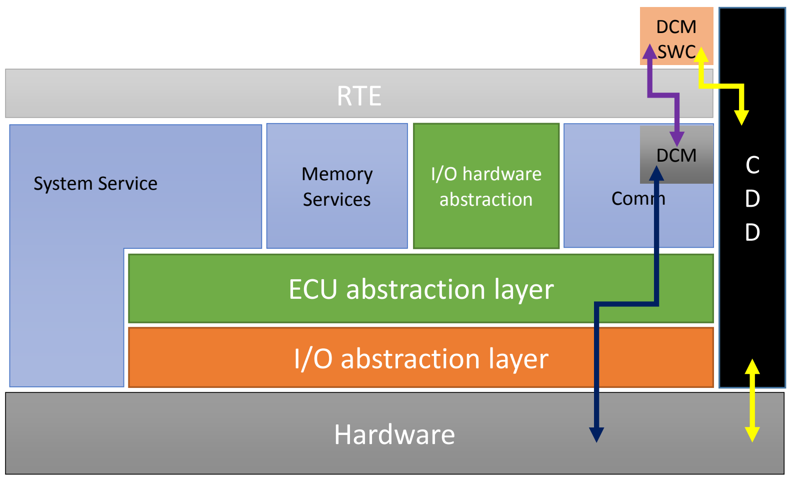
Under the given designs, option (a) will only allow BSW modules to communicate with CDD. The software components have no direct access to the CDD and this can limit those software components willing to control reconfiguration and thus was not chosen for implementation. Options (b) and (c) can be designed to allow both BSW modules and SWC to control reconfiguration. With simplicity in focus, option (c) was chosen for implementation.



(a) CDD entirely in BSW



(b) CDD as two modules, one in BSW and the other as a software component



(c) CDD entirely as a software component

Figure 4.1: Possible locations for CDD

4.2.2 Control of reconfiguration

As mentioned before *Zynq®* architecture has two Cortex-A9 CPUs, Direct Memory Access (DMA), etc., and there are various ways to control the PL reconfiguration. According to [28], reconfiguration can be controlled through either ICAP or PCAP, but for the reconfiguration to be controlled by the processor, PCAP must be used.There are other ways of doing this, like establishing a reconfigurable port in the hardware which on receiving interrupts from processor can begin reconfiguration.

4.3 Tools Used

During software design, *arctic studio* from Arccore AB and *winIDEA* from iSYSTEM were widely used. These tools are proprietary. Details regarding version and their usage are discussed in this section.

4.3.1 Arctic Studio

Arctic studio [33] is a collection of different tools presented as a single package by Arccore AB. To define it in simple words, it is an Integrated Development Environment (IDE). Software development, integration, etc., can be performed in the same tool. The version of *Arctic Studio* used in the project was 10.0.0. Functioning of this tool from the perspective of project is shown in Figure 4.2.

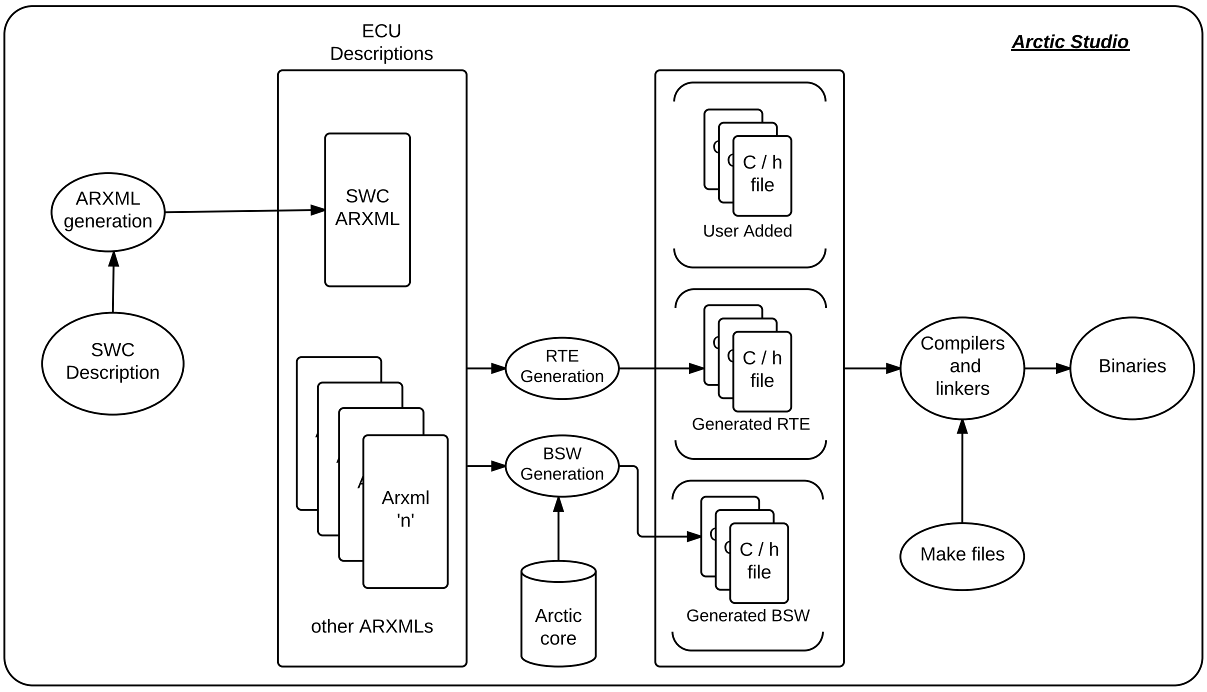


Figure 4.2: Arctic Studio work flow

The user's primary objective is to supply the tool with all the configuration details in the form of ARXMLs. To populate the information about SWC, each SWC is briefed using *ARText*. *ARText* is a framework defined by AUTOSAR Tool Platform (Artop) for modeling Software Component (SWC)s [34]. The external ports of a SWC and the connections between the considered module and other SWCs are listed out during this modeling. After individually defining all the software components, the descriptions are converted into ARXML files. The functionalities of SWC are coded separately with the help of text editors within *arctic studio*. The defined SWCs are instantiated into the design with help of another tool called *RTE editor*.

The next step is to configure the BSW. The functionalities of the BSW are coded in *Arctic Core* , but are very generic. Thus they must be configured with respect to the design needs. ECU configurations are made available to the tool in ARXML format and are referenced by another tool within the *arctic studio* environment called the *BSW editor*. Once all the configurations are present, the files for both BSW and RTE are generated with a single button click. The files are generated in *.c and *.h format.

Finally, the tool also contains a build system specific to all supported hardwares with reference to required compilers and linkers. With the help of build system executables (binaries) can be easily generated.

4.3.2 winIDEA

WinIDEA is an IDE from iSYSTEM [35]. The build used in the thesis work was 9.12.125. It was used extensively along with *iC3000 HS*, a debugger unit [36] and the connection setup is shown in Figure 4.3. As explained in the previous section *arctic studio* was used to develop required software, but it cannot be used to run programs on the hardware. Thus winIDEA along with *iC3000 HS* were useful in understanding the software flow and also for on-board debugging.

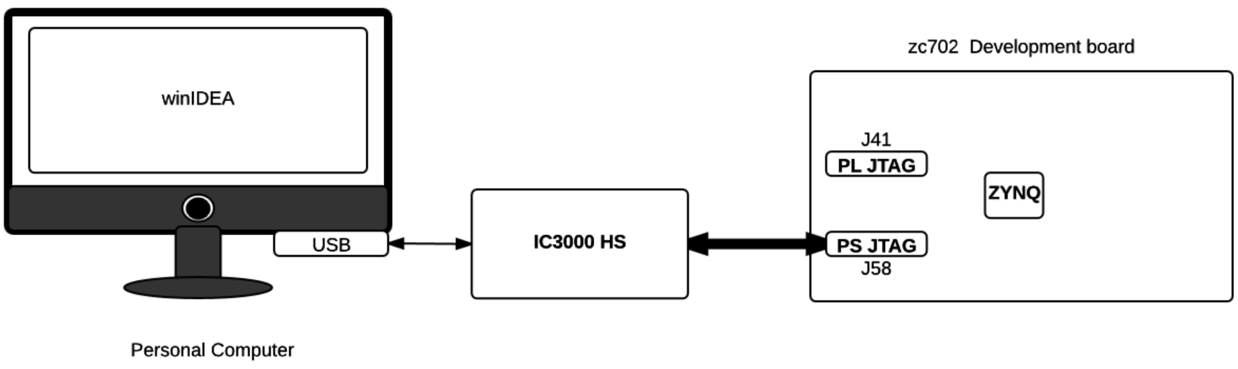


Figure 4.3: Programming Zynq® with winIDEA

Zc702 development board has two Joint Test Action Group (JTAG) connectors that allows programming *Zynq®*'s PS (through pins J58) and PL (through pins J41) separately. Thus *iC3000 HS* with a USB port and a Joint Test Action Group (JTAG) port was used to bridge the connection between the Personal Computer (PC) and J58 of zc702.

4.4 Implementation

The implementation started with analyzing the Hello World example. The various components present and the configuration of DCM module were studied. The CDD was then added to the design.

4.4.1 Hello World

The *Hello World* example with the *Arctic core* release contained a simple ECU configurations necessary to run a software on the hardware with the communication ports enabled. The example could support different boards and thus the configuration available for the *Zynq®* hardware was selected. The executable was generated with the help of *Arctic studio* and it was loaded on to the hardware with then help of *winIDEA*. The executable could periodically blink the LEDs in the zc702 board. Further when it was connected to CAN simulation tool (*Busmaster*) [37] and if random messages were sent with CAN id 0x01 , the loop back program available with it responded by sending back the same sent data in another message with CAN id 0x02.

4.4.2 Diagnostic Control Module

The next step was to configure the *Hello World* so that it responds to the necessary diagnostics. To achieve this analysis showed that only the DCM module's configuration needed change. So the corresponding *ARXML* was opened in *BSW editor* within *arctic studio.* The configuration was done at four levels.

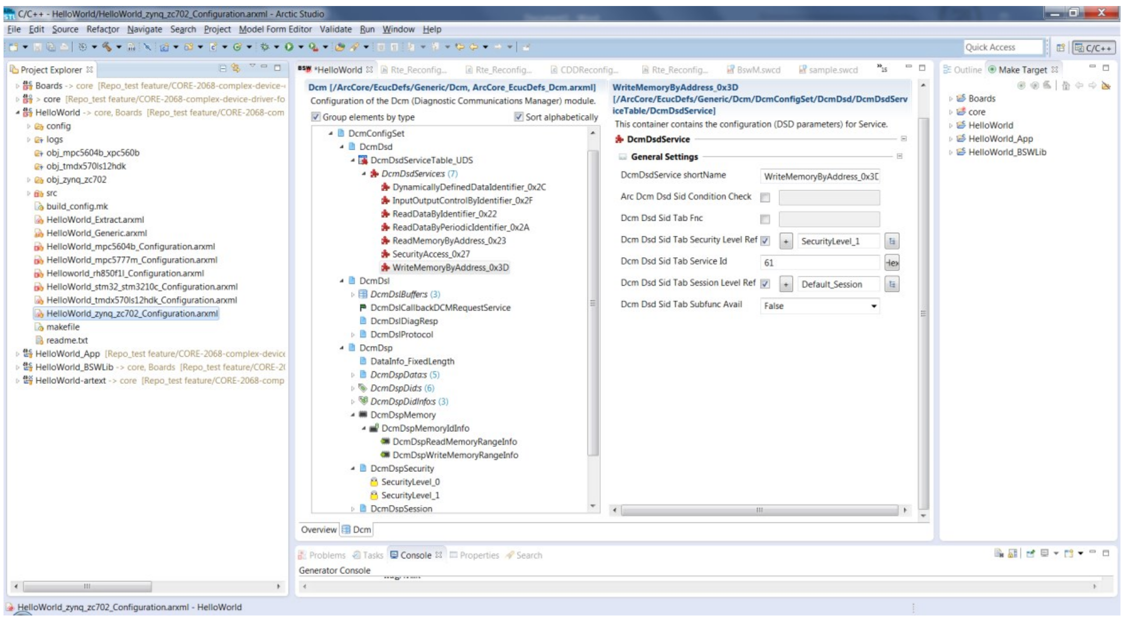
1. **DcmGeneral:** Basic configurations like the support for error detection, maximum dynamic identifiers to be supported can be configured here.
2. **DcmDsd:** The list of services supported by the DCM, for e.g service 0x22: Read Data Identifier (DID) can be configured.
3. **DcmDsl:** The parameters controlling the session layer of DCM, like the list of protocols to be supported, etc., are listed here
4. **DcmDsp:** The finer details of the services supported by the DCM, for example if the module supports memory read services, then details like the memory range accessible, the security session in which the ECU should remain to access these memory addresses can be configured here.

To achieve the desired functionality, the DCM must support services 0x27, 0x23 and 0x3D; Further it should be capable of establishing a secured session in which a limited range of memory is accessible for read and write. The configurations required with respect to individual services are listed in table 4.1.

Table 4.1: Intended services and needed changes

Diagnostic Service	**ID**	**Needed Changes**
SecurityAccess	0x27	a. Add service to supported list b. Additional security session support
ReadMemorybyAddress	0x23	a. Add service to supported list b. choose accessible memory range b. choose accessible security session
WriteMemorybyAddress	0x3D	a. Add service to supported list b. choose accessible memory range b. choose accessible security session

By default only one security level "`SecurityLevel_0`" was supported by the *Hello World* example. This means that the ECU is always in that security level. In addition to this another level `SecurityLevel_1` was added in the **DcmDsp** section. Services 0x23, 0x27, 0x3D were added to the `DcmDsdServiceTable_UDS` of the **DcmDsd** section. The security level needed to access the services 0x23 and 0x3D were set to `SecurityLevel_1`. With the services and security sessions defined, the next step is to configure the memory range accessible using services 0x23 and 0x3D. A new *DcmDspMemory* entry was added in the **DcmDsd** section where memory read and write range information were set. The security level required to access these memory addresses were also set to `SecurityLevel_1`. The screen shot of the BSW editor in *arctic studio* after configuration is shown in Figure 4.4.


C/C++ - HelloWorld/HelloWorld_zynq_zc702_Configuration.arxml - Arctic Studio

Project Explorer
- Boards -> core [Repo_test feature/CORE-2068-complex-device-
- > core [Repo_test feature/CORE-2068-complex-device-driver-fo
- HelloWorld -> core, Boards [Repo_test feature/CORE-2068-com
  - config
  - logs
  - obj_mpc5604b_xpc560b
  - obj_tmdx570ls12hdk
  - obj_zynq_zc702
  - src
  - build_config.mk
  - HelloWorld_Extract.arxml
  - HelloWorld_Generic.arxml
  - HelloWorld_mpc5604b_Configuration.arxml
  - HelloWorld_mpc5777m_Configuration.arxml
  - Helloworld_rh850f1l_Configuration.arxml
  - HelloWorld_stm32_stm3210c_Configuration.arxml
  - HelloWorld_tmdx570ls12hdk_Configuration.arxml
  - HelloWorld_zynq_zc702_Configuration.arxml
  - makefile
  - readme.txt
- HelloWorld_App [Repo_test feature/CORE-2068-complex-devic
- HelloWorld_BSWLib -> core, Boards [Repo_test feature/CORE-2
- HelloWorld-artext -> core [Repo_test feature/CORE-2068-comp

Dcm [/ArcCore/EcucDefs/Generic/Dcm, ArcCore_EcucDefs_Dcm.arxml]
Configuration of the Dcm (Diagnostic Communications Manager) module.
☑ Group elements by type ☑ Sort alphabetically

- DcmConfigSet
  - DcmDsd
    - DcmDsdServiceTable_UDS
      - *DcmDsdServices* (7)
        - DynamicallyDefinedDataIdentifier_0x2C
        - InputOutputControlByIdentifier_0x2F
        - ReadDataByIdentifier_0x22
        - ReadDataByPeriodicIdentifier_0x2A
        - ReadMemoryByAddress_0x23
        - SecurityAccess_0x27
        - WriteMemoryByAddress_0x3D
  - DcmDsl
    - *DcmDslBuffers* (3)
    - DcmDslCallbackDCMRequestService
    - DcmDslDiagResp
    - DcmDslProtocol
  - DcmDsp
    - DataInfo_FixedLength
    - *DcmDspDatas* (5)
    - *DcmDspDids* (6)
    - *DcmDspDidInfos* (3)
    - DcmDspMemory
      - DcmDspMemoryIdInfo
        - DcmDspReadMemoryRangeInfo
        - DcmDspWriteMemoryRangeInfo
    - DcmDspSecurity
      - SecurityLevel_0
      - SecurityLevel_1
    - DcmDspSession

WriteMemoryByAddress_0x3D
[/ArcCore/EcucDefs/Generic/Dcm/DcmConfigSet/DcmDsd/DcmDsdServiceTable/DcmDsdService]
This container contains the configuration (DSD parameters) for Service.

DcmDsdService

General Settings

DcmDsdService shortName	WriteMemoryByAddress_0x3D
Arc Dcm Dsd Sid Condition Check	☐
Dcm Dsd Sid Tab Fnc	☐
Dcm Dsd Sid Tab Security Level Ref	☑ SecurityLevel_1
Dcm Dsd Sid Tab Service Id	61
Dcm Dsd Sid Tab Session Level Ref	☑ Default_Session
Dcm Dsd Sid Tab Subfunc Avail	False

Make Target
- Boards
- core
- HelloWorld
- HelloWorld_App
- HelloWorld_BSWLib

Generator Console

HelloWorld_zynq_zc702_Configuration.arxml - HelloWorld

Figure 4.4: DCM configuration in BSW editor

4.4.3 Complex Device Driver

The next stage in software implementation is to design and integrate a CDD module within the existing project. As decided earlier the design needs this module to be implemented as a SWC in such a way that it can communicate with other SWCs and control the hardware reconfiguration. The steps involved are

- Create a software component description
- Instantiating the newly added SWC.
- Generate RTE
- Add logic for the SWC.

4.4.3.1 Software Component Description

To add a software component to the present design, the tool must be made aware of the SWC design. This is done with the help of a file format called *Software Component Description*, in which the SWC's name and its interface to other SWCs are defined. This file is taken as a reference during RTE generation. The software components are maintained in a separate project (*Hello World* -artext). The software component description of the CDD can be visualized as in Figure 4.5. The CDD was given the name *Reconfigurable Hardware*. The design needs the CDD to know the present state of the ECU and initialize a secured session. For the former, the CDD must get the data from EcuM and for the latter it should communicate with DCM. In case the data from one SWC to the other is sent and if the module needs an acknowledgment such ports are implemented as a server - client port. In our design, the DCM module requires a secured session to be established and thus it is expected to send a request and wait for CDD to respond. Therefore this interface was described as a client-server where CDD is the server and DCM is a client. This can be seen in the diagram as a circled port at the CDD and a semi-circled port at DCM. For a normal data exchange from one port to another asynchronously, a sender - receiver port is used. Thus in the case of transfer of data between EcuM and CDD a sender (at EcuM) - receiver (at CDD) port was defined and they can be seen as ports with arrows showing the direction of data transfer.

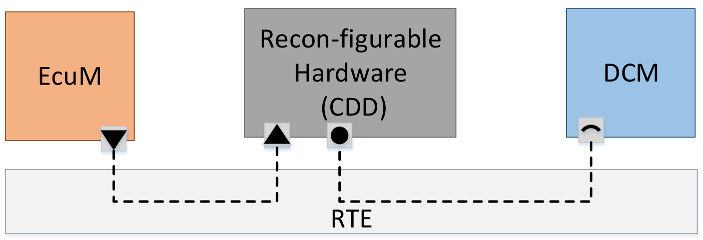
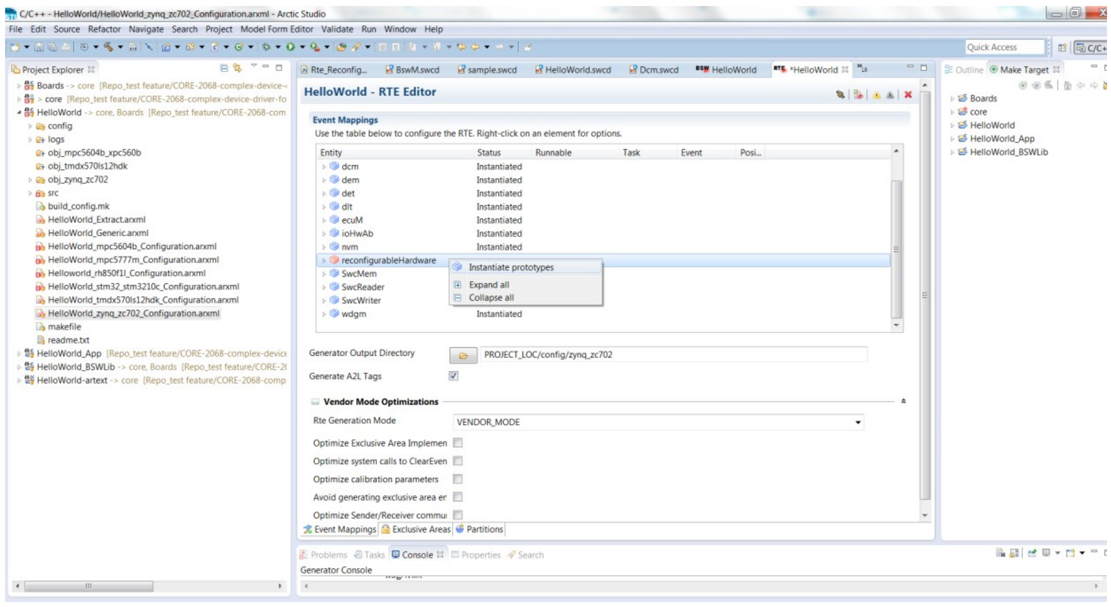


Figure 4.5: Software component Description CDD

4.4.3.2 Instantiate the newly added Software Component (SWC) and generating RTE

With the new SWC added for CDD, the ARXML needs to be regenerated and exported to the present project. When the new ARXML is viewed in arctic studio, the newly added component will be found uninitialized. The instantiation is done with the help of RTE editor as shown in Figure 4.6 and RTE was regenerated.



C/C++ - HelloWorld/HelloWorld_zynq_zc702_Configuration.arxml - Arctic Studio
File Edit Source Refactor Navigate Search Project Model Form Editor Validate Run Window Help
Rte_Reconfig... BswM.swcd sample.swcd HelloWorld.swcd Dcm.swcd HelloWorld *HelloWorld
Instantiate prototypes
Expand all
Collapse all
Event Mappings Exclusive Areas Partitions
Problems Tasks Console Properties Search
Generator Console

HelloWorld - RTE Editor

Event Mappings

Use the table below to configure the RTE. Right-click on an element for options.

Entity	Status	Runnable	Task	Event	Posi...
dcm	Instantiated				
dem	Instantiated				
det	Instantiated				
dlt	Instantiated				
ecuM	Instantiated				
ioHwAb	Instantiated				
nvm	Instantiated				
reconfigurableHardware					
SwcMem					
SwcReader					
SwcWriter					
wdgm	Instantiated				

Generator Output Directory: PROJECT_LOC/config/zynq_zc702

Generate A2L Tags: ☑

Vendor Mode Optimizations

Rte Generation Mode: VENDOR_MODE

Optimize Exclusive Area Implemen ☐

Optimize system calls to ClearEven ☐

Optimize calibration parameters ☐

Avoid generating exclusive area er ☐

Optimize Sender/Receiver commu ☐

Figure 4.6: Instantiation of CDD

4.4.3.3 Algorithm within CDD

The algorithm can be seen in Figure 4.7. The CDD is designed such that it continuously waits for a diagnostic seed request (from service 0x27). If such a request is received, a seed will be sent and a key will be expected from the user. In case of a correct response, the user will be allowed to access the read and write memory by address services (0x23 and 0x31) otherwise a negative response is sent. When a write memory by address service is requested with memory address value between 0x01 to 0x03, the data from the user is taken as input. The reconfigurable module is checked for encryption block. If it is not available, then it is partially reconfigured and then the input data is considered the input data to the AES module and the encrypted cipher data is stored in some secret location in memory. On the read data by memory service, if this data is requested by the user the CDD will copy this data from the secret location and feed to the decryption module (after reconfiguration if needed) and the decrypted data is sent as response to the user.

4.4.3.4 Partial Reconfiguration

To perform partial reconfiguration on Zynq® through PCAP the following steps were followed. This is in accordance with the step of procedure prescribed in technical reference manual.

1. Unlock the SLCR register
2. Disable the AXI interface
3. Disabling PS-PL level shifts
4. Poll the PCAP0INIT status for Reset
5. Disable the AES engine
6. Enable the pcap clock
7. Disable PS-PL level shifts
8. Initialize DMA transaction with proper source address, destination address and size of data to be transfered and the starting location of the data
9. Wait for the programming to complete
10. Check FPGA configuration
11. Wait for reconfiguration to complete
12. Check FPGA configuration
13. Enable the AXI interface
14. Enabling PS-PL level shifts
15. Lock the SLCR again

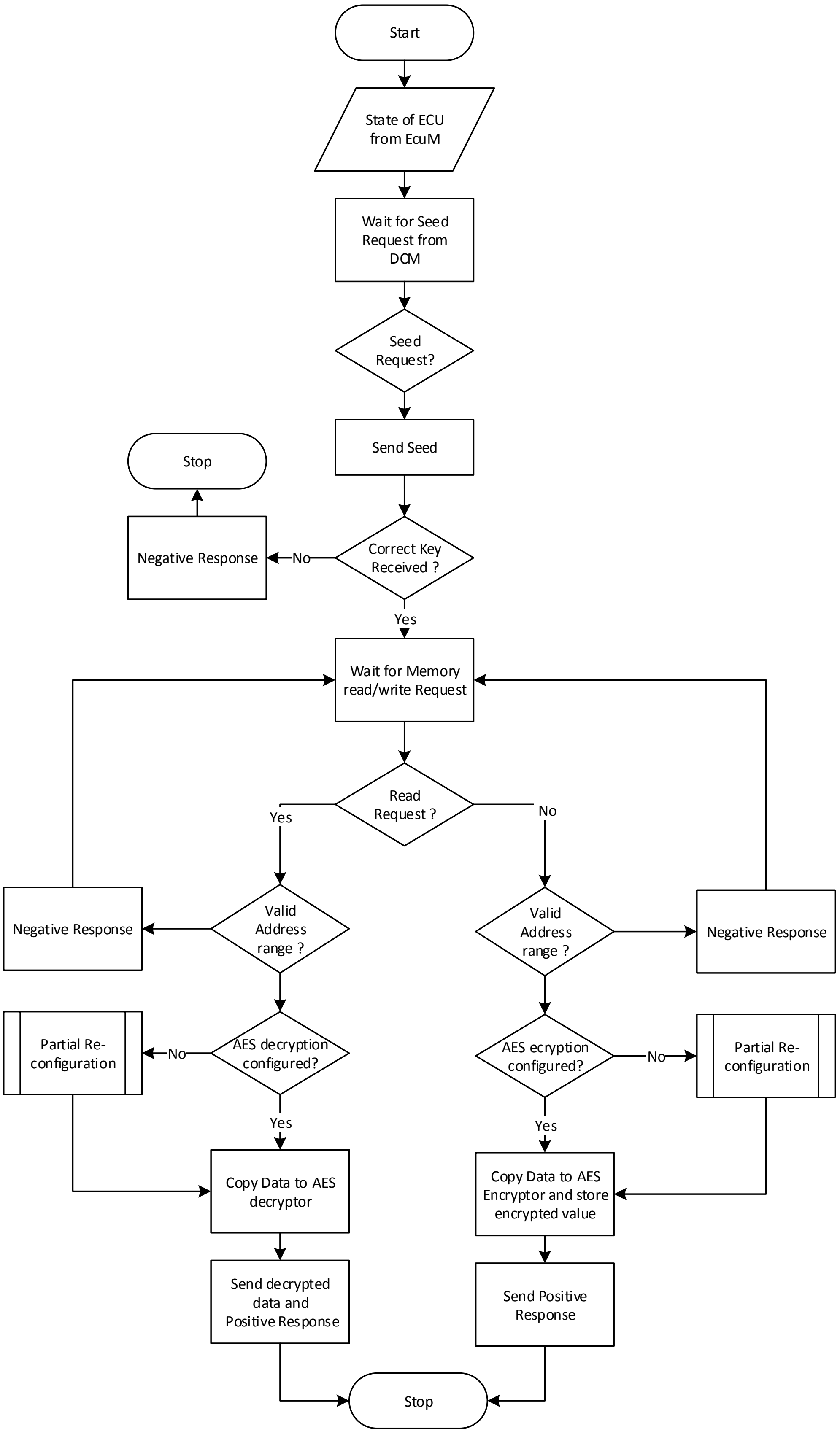


Figure 4.7: Algorithm for CDD

5 Hardware

As a part of hardware design an AES encryption and decryption module that can communicate with the CPU through an AXI bus was designed. There were already some hardware implemented in the company and interfaced to the processor on the AXI bus. This was kept unchanged and the new hardware was designed to be a second slave in the same bus. As per plan the system should be partially and dynamically reconfigured. Due to time constraints decision to use open source AES core was taken. The designing process involved

- Adopting an open source AES core.
- Modifying the core's design for reconfigurable computing
- Adding logic in the design to indicate the hardware configuration.
- Adding AXI interface.
- Packing the designed modules as IP cores and add them to the present design.
- Generate Partial reconfiguration binaries.

5.1 Design

"*Open source IP core library*" [38] released under *Creative Commons Attribution-Non Commercial-ShareAlike 3.0 Unported License* [39] was used. This core was chosen because the description was in VHDL, a familiar description language.

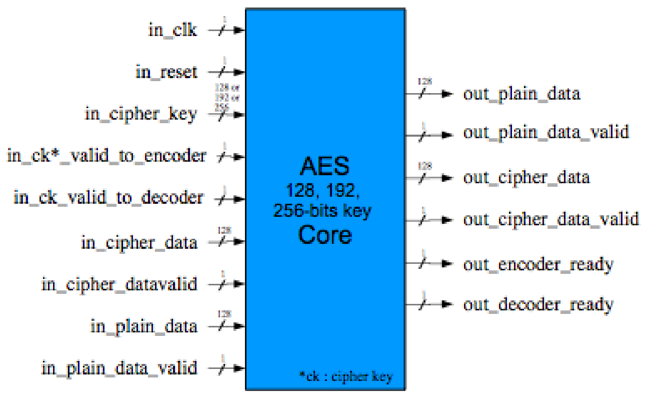


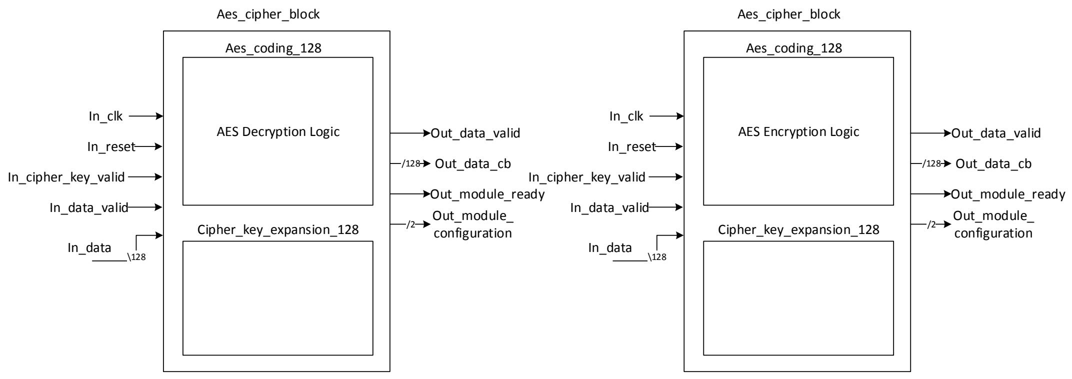
Figure 5.1: AES core

(source: http://www.rachiddafali.com/en/IPcorelibrary.html)

The core top design consisted of both the AES encryption and decryption logic and it was available in three versions corresponding to 128, 192 and 256 bits key. For the thesis, as the intended design is for a 128-bit AES, the core with 128 bit length key was used. The block diagram of the core is shown in Figure 5.1.

5.2 Encryption and Decryption

The first step was to separate the encryption and decryption logic and create two configurations. As the design is for a reconfigurable partition, the reconfigurable sections should have the same name and port configurations[40]. The components relating to encryption and decryption were separated and two projects were created. Both the projects had the top module with name *"aes_cipher_block_128"* and a component *"aes_coding_128"*. But the logic inside them was encryption on one and decryption on the other. A two bit output was added to the design and configured as 00 for decoder and 11 for encoder (A two bit output was chosen to support more reconfigurable logic in the future). This will give out the details of the inner existing configuration to the processor. The 128 bit cipher key value was embedded inside hardware as **0xFEDCBA9876543210FEDCBA9876543210**.



(a) Design with decoding logic **(b)** Design with encoding logic

Figure 5.2: Encryption and Decryption design with same top module Aes_coding_128

5.3 Integrating with present design

Zynq® supports interconnection between user designed hardware logic and the processor with AXI peripherals. So with the help of Xilinx Vivado® [41], AXI interfaces were generated considering that the device will be a slave when interconnected. Thus the design was updated with 10 32-bit slave registers. The input and output of the design were mapped to these registers and the clock will have to be connected to the AXI's clock. Each register can in turn be mapped to the processor with the help of memory. From the software perspective the hardware will just be a black box with inputs and outputs mapped to these registers. Again with the help of Vivado® 's IP packaging option both designs were packed and saved in different folders with the same name. It is to be noted here, if the designs are saved in the same folder it can cause duplication of Intellectual Property (IP) and Xilinx recommends the use of a **rebuilt** algorithm during this process (this can be chosen under the tool's synthesis settings).

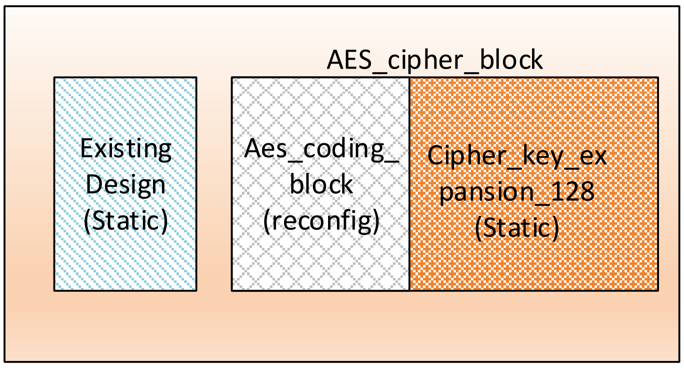

AES_cipher_block
Existing Design (Static)
Aes_coding_block (reconfig)
Cipher_key_expansion_128 (Static)

Figure 5.3: Static and Dynamic modules

Analysing the design, it was found that, only the Aes_coding_128 module was different in the encryption and decryption design. So the designs were synthesized after Aes_coding_128 block was set as an "out of contex module" in each design. On doing this Xilinx tools ensure there are no Input Output (IO) buffers added to these modules during synthesis [41] and only input and output ports will be taken as reference.

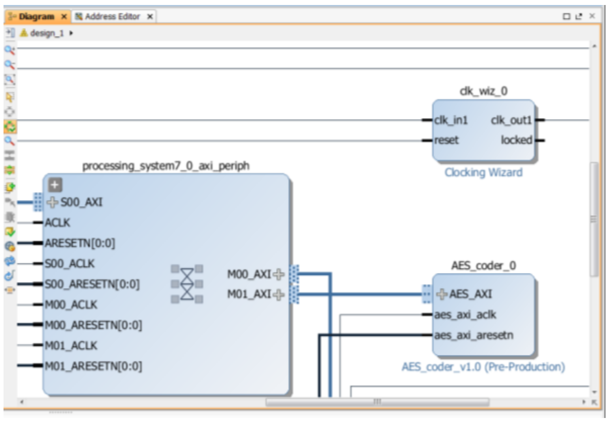


Figure 5.4: View of AES module connected to AXI interconnect in Vivado®

Two copies of the default already existing design were created. This design also had an AXI slave interfaced to the controller. With the help of the IP integrator option in Vivado® , the newly packed IP with a different configuration was added to the design through AXI interconnect block. This can be seen in Figure 5.4. In both the designs, memory address 0x43C10000 was set as the interfacing location.

This means the base register's address of the newly designed module will be visible to the processor at this memory location.

5.4 Generating Partial Reconfiguration bitstreams

The next step is to generate partial binary files that can be used to reconfigure the PL section. For this step, Xilinx's *Partial Reconfiguration* tool was used. This tool currently does not support the Graphical User Interface (GUI) and thus Tool Command Language (TCL) scripts were used to control the work flow. The steps involved performing a *bottom-up synthesis* and during this process, reference to [42] and [43] proved handy.

1. During the previous process, the design *"aes_cipher_block_128"* was synthesized after declaring the modules planned to be the reconfigurable as *out of context*. This is vital and the resultant synthesized design having only ports to reconfigurable blocks without any logic for them will be the base and the process at this point is saved as a check point. Let us refer this as **base check point**. The design had the constraint file to map the pin outputs.

2. Next, the reconfigurable module *"aes_coding_128"* with encryption and decryption logic were separately synthesized and the resultant two check points were also saved. Let us refer to these as **reconfig-encrypt check point** and **reconfig-decrypt check point**.

   It is to be noted here that, Vivado® uses different components to synthesize an efficient design. But all these components do not posses the reconfigurable capacity. One such is **BUFG** and this must not be present in the reconfigurable blocks. To avoid this component being used , the tool must be notified of this limitation in the beginning. This can be done by setting the number of **BUFG** limit to zero in the synthesis setting.

3. Then, the **base check point** was loaded into memory and the tool was then conveyed that the module *"aes_coding_128"* in the design is *reconfigurable* by setting the variable **HD.RECONFIGURABLE** to true for this module.

4. The next step in the process is to set area constraint for the reconfigurable region. Before this the **base check point** which is static only, has to be loaded with one reconfigurable module. It was recommended that if there are many reconfigurable modules, then the most complex among them should be loaded into the static design first. Among the reconfigurable modules in this work, the decryption module had more logic than the encryption module and thus **reconfig-decrypt check point** was loaded first and the design was saved as a check point. Let this be referred as **reconfig-init check point**.

   Though the implementation size of the reconfigurable designs are different, the work flow requires a common shared region. Thus it is always better to start with the bigger modules.

5. Now, when the *"aes_coding_128"* is checked, it can be observed to have circuit nets. This block is to be confined within a given area and the process is called *floor planning.* A region in hardware must be chosen where the reconfigurable module can be implemented and in Xilinx's terminology this is called a **pblock**. While choosing the size of the pblock, it should be ensured that the number of components available like Look Up Table (LUT), etc., is sufficient enough to implement the required circuit. The PL has different clock regions and the *pblock* selection though can share different clock regions, for better results the least clock boundry sharing should be looked for. After size selection, there is Design Rule Check (DRC) utility within Vivado® . This can be used to check the validity of the design so far specific to partial reconfiguration. This can be seen as a pink box in Figure 5.5. In the design, the *pblock* shares 4 clock regions.

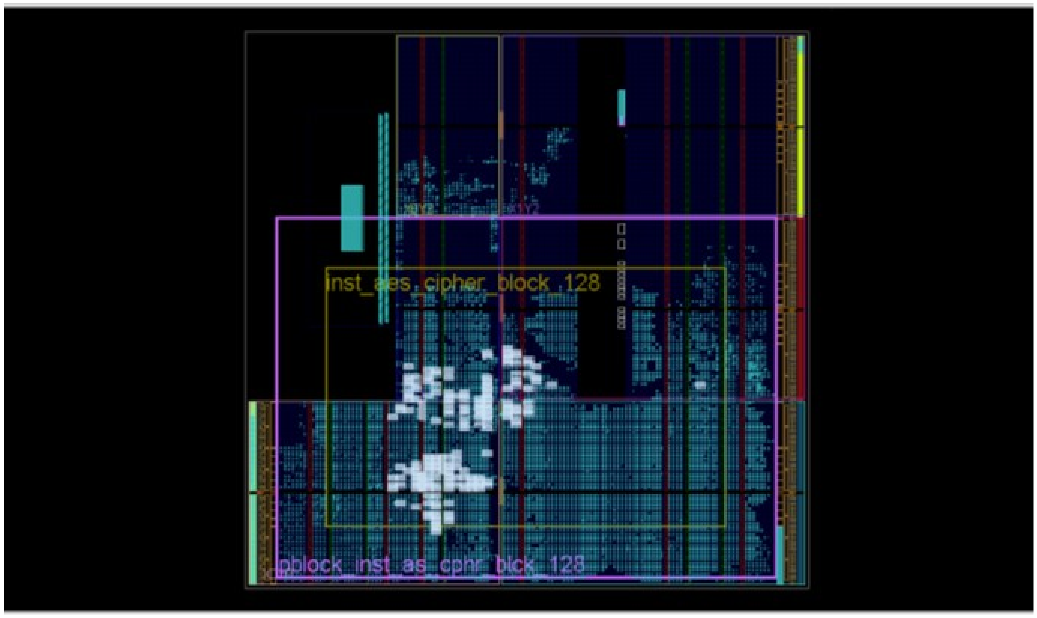


Figure 5.5: Placed design as viewed in Xilinx Design view

6. After the successful validation from DRC, the *pblock* region will be an area constraint and thus it is saved as FloorPlan.xdc file and is stored in a folder along with other constraints.

7. There is an option in the tool to enable a reset after partial reconfiguration on the reconfigurable region. When this is enabled, the circuits involving partial reconfiguration are reset once after the completion of reconfiguration

process. To enable this feature, variable RESET_AFTER_RECONFIG was set as true.

8. Next process in the flow is to place the design. After placement the resultant design can be seen in Figure 5.5. Timing constraints were not considered during the design. Some highlighted regions visible in the figure show the buffers that share the reconfigurable design.

9. After placement, it is time for routing. The output can be seen in Figure 5.6. The green part of the design show the wires. It can be seen they are concentrated on the left top of the design. This is because the processor interface is located over there. The resulting design completes the first iteration and the results are stored as a design check point.

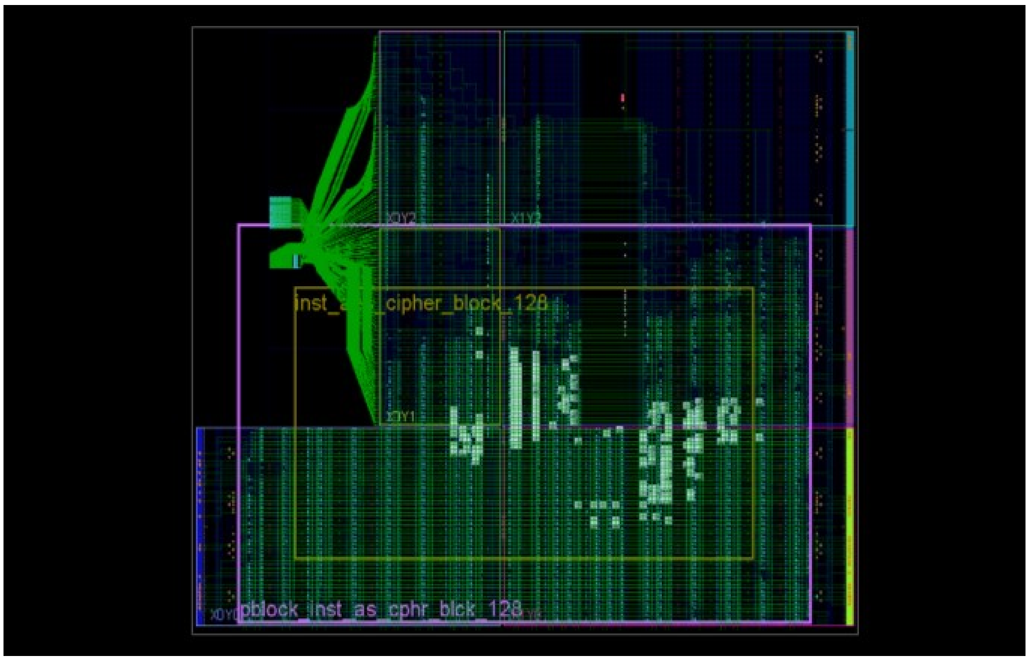


Figure 5.6: Routed design as viewed in Xilinx Design view

10. The next step is to implement the *reconfig-encrypt check point* into the circuit. But before doing this, the implemented circuit corresponding to the *reconfig-decrypt check point* should be removed. This is done by the tool when the design is asked to be updated by a black box in the place of a *reconfig-decrypt check point*. This can be seen in Figure 5.7. The routing for the reconfigurable block is taken off by the tool. This state of connections is saved by locking the design.

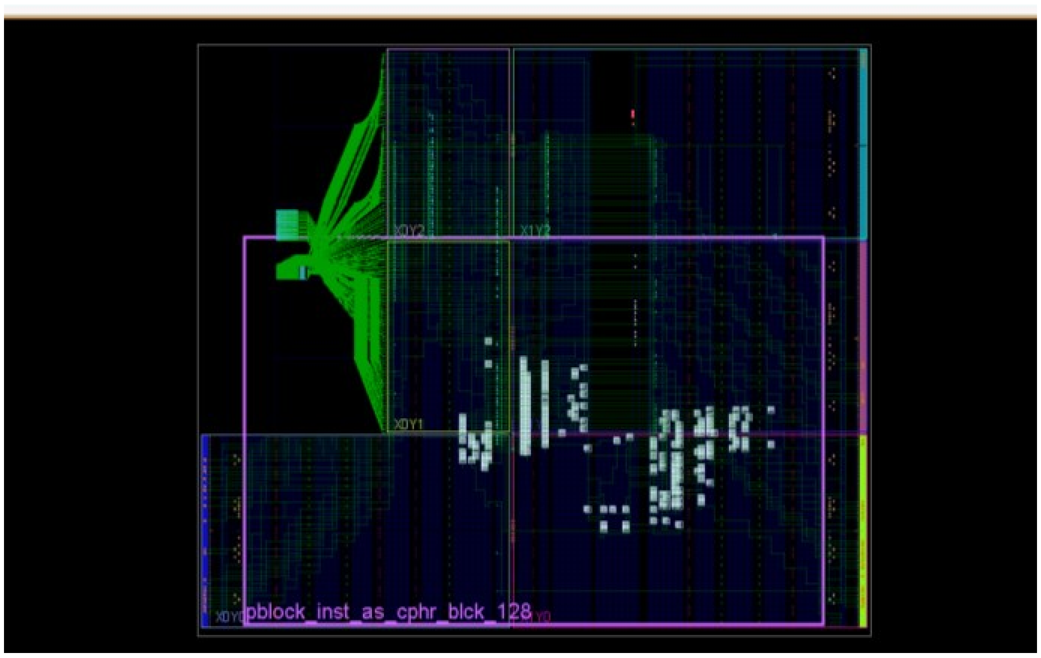


Figure 5.7: Design after resetting reconfigurable module as viewed in Xilinx Design view

11. As described earlier, a **reconfig-encrypt check point** configuration was loaded into the design. The resulting design is stored as a checkpoint for future use. The tool finds the reconfigurable connections and the connection is shown in yellow color.

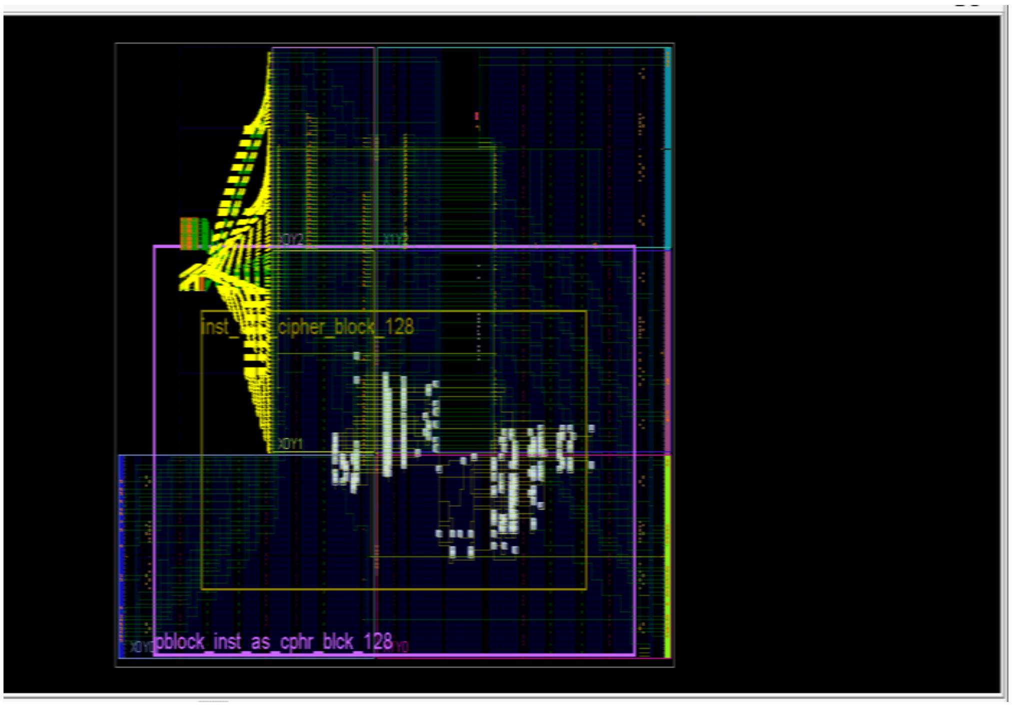


Figure 5.8: Design after writing the second reconfigurable module

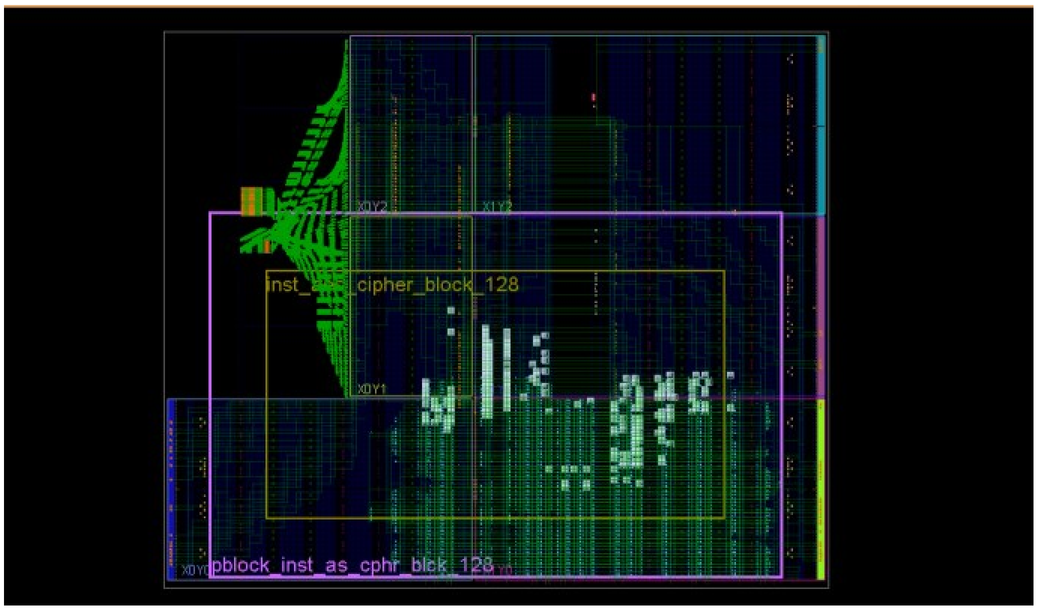


Figure 5.9: Second Routed design as viewed in Xilinx Design view

12. The new design is placed and routed as in the first design.

13. The final step is to generate the bit files As a result of these procedures 4 different bit files are generated; two that is for the whole design and two that represent partial reconfiguration. The bit files generated according to the requirement should use PCAP in the hardware to reconfigure PL. But the bit files now generated cannot do that. To enable this the "write_cfgmem" command must be run with the "-disablebitswap" option.

14. It is worth mentioning that if there is a need for a complete reconfiguration of the hardware through PCAP, then the generated bit file using the above said utility should change format.

The resource needed to be reconfigured can be inferred from the Table 5.1.

Table 5.1: Resource utilization between partial and full configuration

Sl No.	**Resource**	**Partial Configuration(%)**	**Full Configuration (%)**
1	Slice Registers	51	26.17
2	Look-up Tables	4.36	4.36
3	F7 Muxes	35.73	20.21
4	F8 Muxes	34.59	19.55

6
Booting

With both hardware and software developed, the next process is to port them on the hardware (SOC). The binaries generated using Arctic Studio and Vivado® can be ported to the ZC702 board in different ways. One easy way is to store the generated binaries for a specific configuration in a Secure Digital (SD) card and allow the board to automatically load the configuration during power up. To do this, the binaries must be stored in a format called **Boot image** and **Xilinx Software Development Kit (SDK)** was used for this purpose.

6.1 Zynq® Boot Process

The power-on sequence inside Zynq® is designed in a way that every time when the device is powered on, the PS section gets power first and one core, out of the two, starts executing the code stored within the boot ROM available.

- Carry on intialization of basic hardware and software parameters
- Initialize JTAG
- Look for boot strap pins (In this case it is configured for SD card)
- Look for the First Stage Boot Loader (FSBL). In case FSBL is present, start executing it

6.2 FSBL

The function of the FSBL is to initialize the board and to generate the FSBL, Xilinx SDK was used. The development of hardware using Vivado® is overwhelmingly dependent on the Zynq® architecture. The interaction of hardware to the processor and other basic parameter values are generated or are to be chosen by the designer. All these parameters are exported to Xilinx SDK as a Hardware Description File (HDF). When a boot loader project is chosen (available for easy FSBL generation inside the SDK), the configuration details are collected from HDF and an FSBL is generated.

6.2.1 Xilinx SDK

Xilinx SDK is an IDE from Xilinx which has many tools embedded inside, that aids software development for FPGAs. The release version used during the software development was 2015.4. Under the *Xilinx Tools* menu exists the **Create Boot**

Image tool. During the PL design, the memory mappings of the designed hardware with respect to the Zynq® CPU is generated and this file is called the **hardware definition file**. The hardware definition file (stored as boot loader), the software binary from Arctic Studio *(in *.elf format)* and the PL binary from Vivado® *(in *.bit format)* are placed in a list as input and the **boot image** (in **.bin* format) file was generated as output.

The FSBL is a simple C program and has initializing values for various registers. After initializing, in the case of an SD card, the generated FSBL has a file system probing algorithm, using which it probes through the card. If it finds any (*.bit) file located then the PL is configured with it. It then probes for other (*.bin) files within the SD card. If found they are loaded on to the memory.

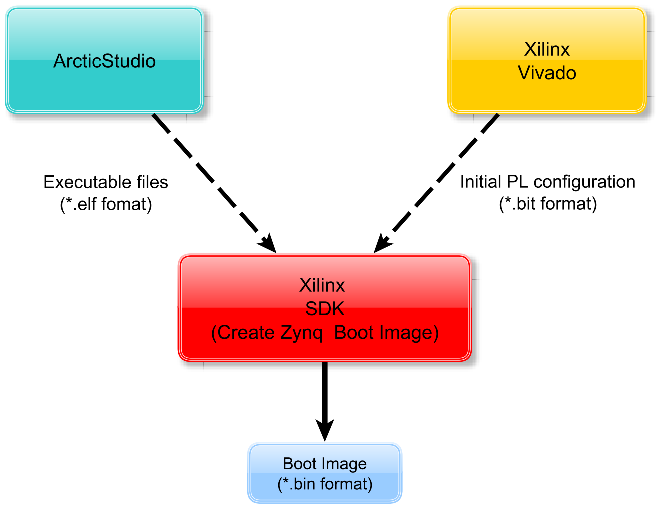

ArcticStudio
Xilinx Vivado
Executable files (*.elf fomat)
Initial PL configuration (*.bit format)
Xilinx SDK (Create Zynq Boot Image)
Boot Image (*.bin format)

Figure 6.1: Boot image creation

6.3 Boot Image

A Boot image is required on the SD card to execute FSBL. This is created with the help of *"Create Boot Image"*, a tool present within the Xilinx SDK. The order for the files to be present in the SD card can be defined in a file format (*.bif) and the tool uses this as reference to generate the boot image. As shown in Figure 6.2, the FSBL is placed first, followed by PL configuring bit file and then the two partial reconfiguring binary files with load address reference, stating where within the memory these files should be loaded were placed. If needed a *fail safe image* can be designed. This section also called the *golden image*, is the program section to be executed when the primary region has a problem. In this work, the *golden image* was not created. It is a must that the FSBL is available at the first place, followed by the PL configuring bit file. The other files can have their sequence changed.

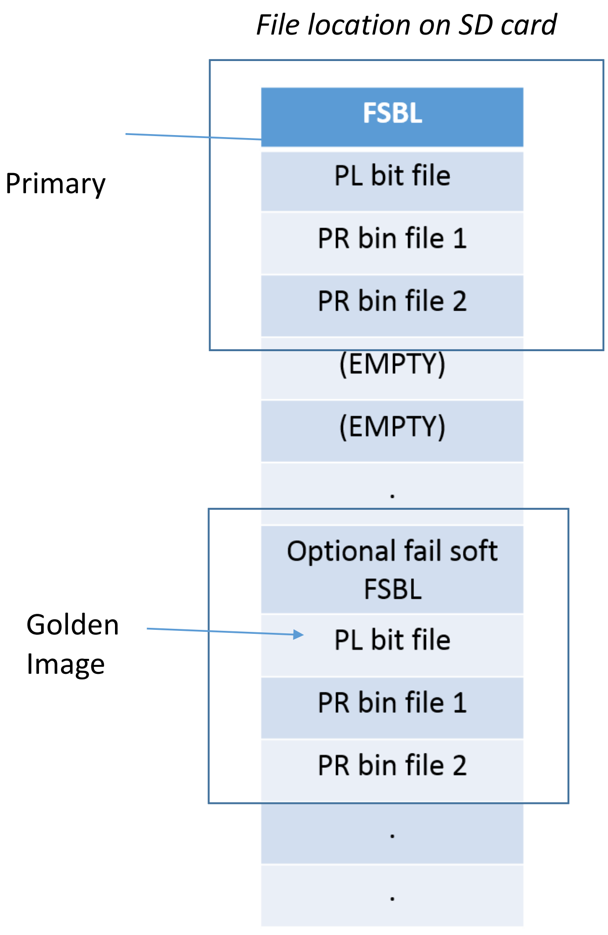
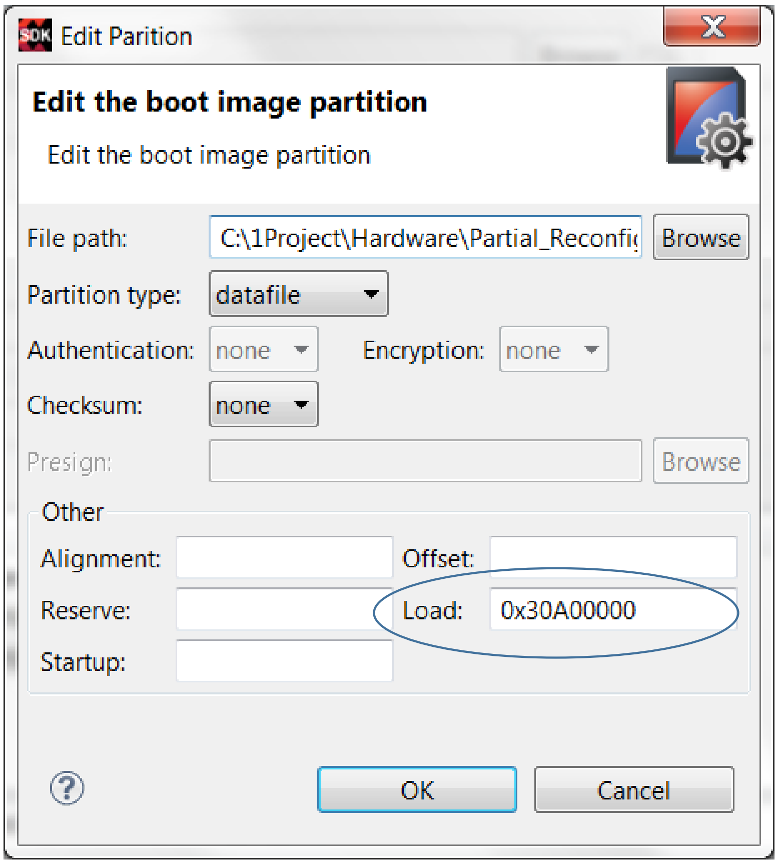


Figure 6.2: Boot image structure

There are many ways the partial binary files can be stored into memory. A simple way is to define the memory address into which the file should be loaded within the "Create Boot Image" tool as shown in Figure 6.3. The first PR image was loaded at 0x30A00000 and the second PR image at 0x30F00000.



Edit the boot image partition

Edit the boot image partition

File path: C:\1Project\Hardware\Partial_Reconfi

Partition type: datafile

Authentication: none

Encryption: none

Checksum: none

Presign:

Other

Alignment:

Offset:

Reserve:

Load: 0x30A00000

Startup:

Figure 6.3: Reference location of PR bit files as seen in Xilinx "Create Boot Image" tool

7
Test

The system after design should respond to diagnostic messages. In order to verify the functionality, a small test program was executed.

7.1 Test Setup

The test setup for the system is shown in Figure 7.1. The development board was connected to the computer through a CAN- USB converting tool. PS JTAG was connected and the memory location 0x43C10000 was monitored periodically by stopping the code execution. This was done because the tool was not able to get the real time memory values. Once stopped, the tool updates it with newer value in the memory

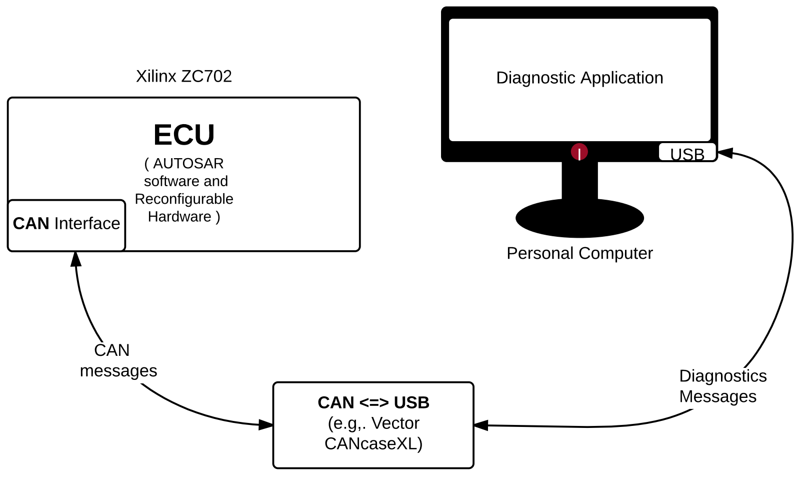


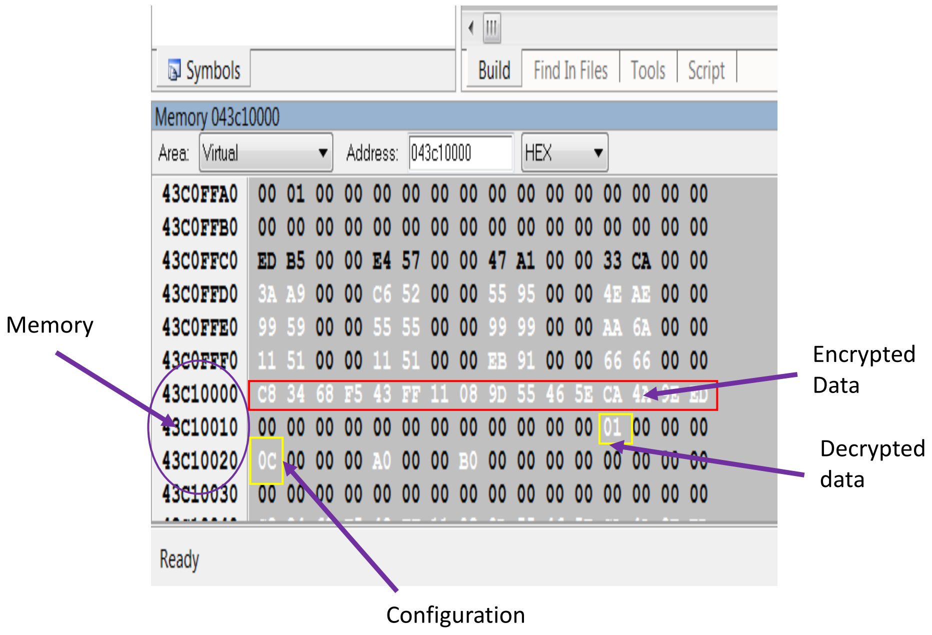
Figure 7.1: Test Setup

At the start of the program, encoder logic was present and this was ensured by reading memory location 0x43c10020, the register mapped to the configuration pins inside the designed AES module. The value was 0x0F, where the final 2 bits were of concern and it was 11 corresponding to the encoder.

7.2 Test Procedure

The diagnostic messages in the CAN format were sent from the computer through the diagnostic tools and correct response from the ECU and memory read/write were tested. The test sequence was as follows.

1. Seed requested through diagnostic command (0x27).
2. Obtained Seed from CDD as 0x09010203.
3. Had a hard coded logic to check for key to be 0x0A020304 and therefore it was sent as key. Obtained positive response indicating a security level check is over.
4. Diagnostic memory write by address was chosen for memory address 0x0001 and value 0x01 was written.
5. Positive response obtained indicating write was successful.
6. Diagnostic memory read by address was chosen for memory address 0x0001.
7. Positive response obtained with value 0x01 as response, indicating 0x01 was the decoded value.
   Applying break point on the program, the memory address was checked and the screen shot in Figure 7.2 shows that.



```
Memory 043c10000
Area: Virtual    Address: 043c10000    HEX

43C0FFA0  00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00
43C0FFB0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
43C0FFC0  ED B5 00 00 E4 57 00 00 47 A1 00 00 33 CA 00 00
43C0FFD0  3A A9 00 00 C6 52 00 00 55 95 00 00 4E AE 00 00
43C0FFE0  99 59 00 00 55 55 00 00 99 99 00 00 AA 6A 00 00
43C0FFF0  11 51 00 00 11 51 00 00 EB 91 00 00 66 66 00 00
43C10000  C8 34 68 F5 43 FF 11 08 9D 55 46 5E CA 4A 9E ED
43C10010  00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00
43C10020  0C 00 00 00 A0 00 00 B0 00 00 00 00 00 00 00 00
43C10030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Ready
```

Figure 7.2: Test result

The figure shows memory location 0x43c10020 to have configuration 0x0C indicating it now has a decryption logic inside it. Thus we can infer successful reconfiguration has taken place. Decrypted data 0x01 was in the slave register0 mapped to memory location 0x43c1000C. Thus decryption had also succesfully happened.

8
Discussion and Conclusion

During the thesis various possibilities for implementing reconfigurable computing were investigated and a system was designed. Some of the system's characteristics are worth of discussion.

8.1 Discussion

1. **Possibility of Partial Reconfiguration:**

   As per the design and the results obtained partial reconfiguration in AUTOSAR is very much possible.

2. **Timing:**

   The timing of reconfiguration depends on the size of the partial bit files. On observation it is fast as it uses DMA. But, in the design during the reconfiguration, interrupts are to be masked. This can result in long wait times for tasks of same priority as of the CDD. This can be a potential problem when safety critical tasks are stalled.

3. **Advantages of partial reconfiguration:**

   The resources that need to be reconfigured are reduced. This will in turn make the circuit quickly reconfigure and is thus preferable over fully reconfigurable solutions . But the effort needed for development of partial reconfigurable solutions is higher compared to the conventional solution and it requires additional tools.

8.2 Conclusion

With growing computing needs like image processing, hardware acceleration will be a major solution in the automotive domain in the future. Also there are security features which are enhanced if they are implemented on hardware. But implementing more hardware circuits make the system less flexible. Bug fixes in the hardware are not as easy to handle in software.

For all these problems reconfigurable computing can be a solution. It is proved from the thesis that reconfigurable computing is possible in AUTOSAR, but with few limitations and high dependency on hardware. Also if there is a possibility to choose between partial and complete reconfiguration, partial reconfiguration though needing a little more development time, is better.

Bibliography

[1] N. Zaman, *Automotive Electronics Design Fundamentals*, v1.0 ed. Springer, 2015.

[2] AUTOSAR. (2014) Basics. (Date last accessed: 16-May-2016). [Online]. Available: http://www.autosar.org/about/basics

[3] ——. (2014) Background. (Date last accessed: 16-May-2016). [Online]. Available: https://www.autosar.org/about/basics/motivation-goals

[4] F. SIT. (2008) E-safety vehicle intrusion protected applications. (Date last accessed: 04-Feb-2016). [Online]. Available: http://evita-project.org/index.html

[5] AUTOSAR. (2014) Key features. (Date last accessed: 04-Apr-2016). [Online]. Available: http://www.autosar.org/about/basics/key-features/

[6] M. Gokhale and P. Graham, *Reconfigurable Computing : Accelerating Computation with Field Programmable Gate Arrays.* Dordrecht: Springer, 2005.

[7] Xilinx. (2016) All programmable soc. (Date last accessed: 16-June-2016). [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html#productTable

[8] ISO/IEC 7498-1:1994(E), *Information Technology – Open Systems Interconnection – Basic Reference model: The Basic Model.* International Organization for Standardization, Geneva, CH, 1994.

[9] M. M. Alani, *Guide to OSI and TCP/IP Models.* Springer, 2014.

[10] AUTOSAR. (2014) Background. (Date last accessed: 16-May-2016). [Online]. Available: https://www.autosar.org/about/basics/background

[11] AUTOSAR. (2014) Current partners. (Date last accessed: 25-May-2016). [Online]. Available: http://www.autosar.org/partners/current-partners

[12] F. Helmut *et al.*, *Achievements and Explotation of the AUTOSAR development Partnership*, v4.2.0 ed. SAE convergence Congress, 2006.

[13] C. Hammerschmidt. (2011) Autosar core partners reveal roll-out plans. (Date last accessed: 25-May-2016). [Online]. Available: http://www.electronics-eetimes.com/news/autosar-core-partners-reveal-roll-out-plans

[14] AUTOSAR. (2014) Brochure. (Date last accessed: 25-May-2016). [Online]. Available: http://www.autosar.org/fileadmin/files/presentations/AUTOSAR_Brochure_EN.pdf

[15] AUTOSAR., *AUTOSAR-TR-Methodology*, v2.1.0 ed. AUTOSAR, 2014.

[16] AUTOSAR, *Layered Software Architecturep*, v3.2.0 ed. AUTOSAR, 2014.

[17] S. 4-A, *Systems Engineering Fundamentals.* Fort Belvoir, Virginia: Defense Acquisition University Press, 2001.

[18] OTAQ. (2015, March) On-board diagnostics. (Date last accessed: 03-Feb-2016). [Online]. Available: http://www.epa.gov/obd/

[19] ISO 14229-1:2013(E), *Road vehicles — Unified diagnostic services (UDS) — Part 1: Specification and requirements.* Case postale 56 • CH-1211 Geneva 20: International Organization for Standardization, Geneva, CH, 2013.

[20] ISO/DIS 15765-2, *Road vehicles — Diagnostic communication over Controller Area Network (DoCAN) — Part 2: Transport protocol and network layer services.* Case postale 56 • CH-1211 Geneva 20: International Organization for Standardization, Geneva, CH, 2013.

[21] R. B. Gmbh, *CAN Specification*, 2nd ed., Postfach 50, D-7000 Stuttgart 1, 1991.

[22] NIST. (2013) Nist general information. (Date last accessed: 11-Feb-2016). [Online]. Available: http://www.nist.gov/public_affairs/general_information.cfm

[23] J. Daemen and V. Rijmen, *Aes proposal: Rijndael*, 1999.

[24] FIPS197. (2001) Advanced encryption standard (aes). (Date last accessed: 11-Feb-2016). [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[25] Wikipedia. (2016) Computing. (Date last accessed: 16-May-2016). [Online]. Available: https://en.wikipedia.org/wiki/Computing

[26] ——. (2016) Reconfigurable computing. (Date last accessed: 16-May-2016). [Online]. Available: https://en.wikipedia.org/wiki/Reconfigurable_computing

[27] C. Bobda, *Introduction to Reconfigurable Computing: Architectures, Algorithms and Applications.* P.O. Box 17, 3300 AA Dordrecht, The Netherlands: Springer, 2007.

[28] Xilinx, *Zynq-7000 All Programmable SoC Technical Reference Manual UG585*, v1.10 ed. Xilinx ®, 2015.

[29] AUTOSAR. (2014) Autosar vendor id list. (Date last accessed: 08-Apr-2016). [Online]. Available: http://www.autosar.org/documents/vendor-id

[30] A. AB. (2016) Home. (Date last accessed: 09-May-2016). [Online]. Available: http://https://www.arccore.com

[31] AUTOSAR., *Specification of Diagnostic Communication Manager*, v4.2.0 ed. AUTOSAR, 2014.

[32] AUTOSAR, *Complex Driver design and integration guideline*, v1.1.0 ed. AUTOSAR, 2014.

[33] Arc-core. (2016) Arctic studio – the development tools. (Date last accessed: 09-Feb-2016). [Online]. Available: http://www.arccore.com/products/arctic-studio

[34] Artop. Artext - an autosar textual language framework. (Date last accessed: 10-Apr-2016). [Online]. Available: https://www.artop.org/artext/swcd

[35] iSYSTEM AG. winidea. (Date last accessed: 8-Apr-2016). [Online]. Available: http://www.isystem.com/products/software/winidea

[36] iSYSTEM. AG. Ic3000. (Date last accessed: 8-Apr-2016). [Online]. Available: http://www.isystem.com/products/software/winidea

[37] RBEI and E. GmbH. Busmaster. (Date last accessed: 8-May-2016). [Online]. Available: https://rbei-etas.github.io/busmaster/

[38] R. DAFALI. Open source ip core library. (Date last accessed: 30-May-2016). [Online]. Available: http://www.rachiddafali.com/en/IP_core_library.html

[39] C. Commons. Attribution-noncommercial-sharealike 3.0 unported. (Date last accessed: 30-May-2016). [Online]. Available: http://creativecommons.org/licenses/by-nc-sa/3.0/legalcode

[40] Xilinx, *Vivado Design Suite User Guide - Partial Reconfiguration (UG909)*, v2016.1 ed. Xilinx ®, 2016.

[41] xilinx, *Vivado Design Suite User Guide - Synthesis (UG901)*, v2013.1 ed. Xilinx ®, 2013.

[42] Xilinx, *Vivado Design Suite User Guide - Partial Reconfiguration (UG909)*, v2016.1 ed. Xilinx ®, 2016.

[43] xilinx, *Vivado Design Suite Tutorial - Partial Reconfiguration (UG947)*, v2015.4 ed. Xilinx ®, 2015.