ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

INSTRUCTION SET EXTENSION OF SOME PROCESSORS FOR SECURE IoT IMPLEMENTATIONS

SENIOR DESIGN PROJECT
INTERIM REPORT

Cihan Berk GÜNGÖR
Yusuf ÖNDEŞ
Talip Tolga SARI
Berkay UÇKUN

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

APRIL 2018

ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

INSTRUCTION SET EXTENSION OF SOME PROCESSORS FOR SECURE IoT IMPLEMENTATIONS

SENIOR DESIGN PROJECT
INTERIM REPORT

Cihan Berk GÜNGÖR
(040130017)

Yusuf ÖNDEŞ
(040130061)

Talip Tolga SARI
(040140007)

Berkay UÇKUN
(040120037)

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

Project Advisor: Prof. Dr. Berna Örs YALÇIN

APRIL 2018

We are submitting the Senior Design Project Interim Report entitled as “INSTRUCTION SET EXTENSION OF SOME PROCESSORS FOR SECURE IoT IMPLEMENTATIONS”. The Senior Design Project Interim Report has been prepared as to fulfill the relevant regulations of the Electronics and Communication Engineering Department of Istanbul Technical University. We hereby confirm that we have realized all stages of the Senior Design Project Interim Report by ourselves, and we have abided by the ethical rules with respect to academic and professional integrity .

Cihan Berk GÜNGÖR .....
(040130017)

Yusuf ÖNDEŞ .....
(040130061)

Talip Tolga SARI .....
(040140007)

Berkay UÇKUN .....
(040120037)

Project Advisor: **Prof. Dr. Berna Örs YALÇIN** .....

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

1. INTRODUCTION

In this graduation project, Instruction Set Extension (ISE) [1] is applied on four different 32-bit processor: Leon3 ref which is constructed with SparcV8 Instruction Set Architecture (ISA) [2], OR 1200 which is constructed with OpenRISC 1000 ISA[3],ARM Cortex-M0 DesignStart r2p0 which is constructed with v6-M Archtecture,----tolga----. Flexibility and efficiency can be achieved with the ISE on processors for different applications [4].

Security is the most important factor of the new generation Internet of Things (IoT) applications [5]. Various cryptography algorithms developed for secure communication or data storage at mobile phones, computers, wireless networks etc. [6], [7]. Because of the length of the key, and the number of the rounds, a software implementation of the block ciphers consume high rate of power and time when the device is working. Block ciphers also have high power consumption and timing problems, ISE is used on processors to implement block ciphers more efficiently [8]. There are several examples of block ciphers such as Advanced Encryption Standard (AES) [9], Tiny Encryption Algorithm (TEA) [10], Present[?]---tolga---berkay??---.

2. LEON3

2.1 Introduction

Leon3 is designed from Cobham Gaisler research that has made freely available a collection of open source IP cores and a design configuration environment for developing the Leon3 based Field Programmed Gate Array (FPGA) designs, collectively referred to as GRLIB IP library [11]. The block diagram of Leon3 is shown in Figure 2.1.

Security is the most important factor of the new generation Internet of Things (IoT) applications [5]. Various cryptography algorithms developed for secure communication or data storage at mobile phones, computers, wireless networks etc. [6], [7]. Because of the length of the key, and the number of the rounds, a software implementation of the block ciphers consume high rate of power and time when the device is working. Block ciphers also have high power consumption and timing problems, ISE is used on processors to implement block ciphers more efficiently [8]. Two examples of block ciphers are Advanced Encryption Standard (AES) [9] and Tiny Encryption Algorithm (TEA) [10].

In this study, Leon3 processor is implemented on Spartan6 FPGA board. A simple software is implemented and verified on board. A software that uses the Input / Output (I/O) unit of the board is also implemented and tested. Debugger for Leon3 processor is installed and tested. Arithmetic Logic Unit (ALU) is extended as it includes main operations of the AES block cipher. Instruction set of Leon3 processor is enlarged to use these operations.

Previously, Leon3 processor is used for analysis of its performance, memory [12]. It also used for image and video processing [13], [14]. Leon3 is used to implement AES block cipher without ISE and AES software is written and run on Leon3 [15]. ISE is implemented on different processor architectures for different purposes [16]. Moreover, ISE is implemented on Leon3 for block ciphers, but new instructions are not generated for each of the operations of the AES block cipher. In stead of this, encryption and decryption are implemented as two new operations [17].

In this study, ISE is used to implement AES block cipher. Each operation of the AES block cipher is added to instructions of the SparcV8. Three new added instructions are used in the software of the AES block cipher. Thus, required hardware addition of new instructions is reduced. It also reduces usage of the source of the FPGA on target board.

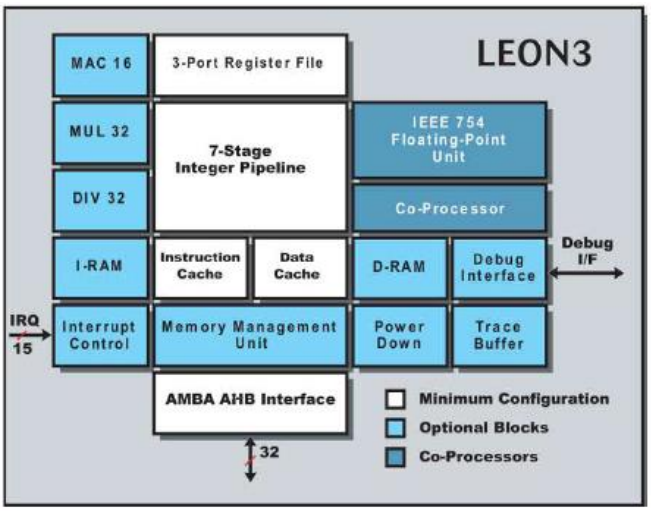


Figure 2.1 : Architectural Block Diagram of the Leon3 processor [10]

2.2 Leon3 Processor Architecture and SparcV8 Instruction Set Architecture

Leon3 is developed by a Cobham Gaisler company that provides open source IP library called as GRLIB and it contains all source codes of the Leon3 processor [18]. Leon3 is based on SparcV8 ISA and it developed with Very High Speed Integrated Circuit Hardware Description Language (VHDL). It supports designs upon 16 Central Processing Unit (CPU) [19]. It contains 7 pipeline stages and it supports several communication interfaces. It has integer unit (IU), general purpose registers, Random Access Memories (RAMs) and their controllers, Floating Point Unit (FPU), and Memory Management Unit (MMU) [18]. Some of the features of Leon3 given below;

- Leon3 is a part of a Gaisler IP Library (GRLIB), thus it can be changed easily. Because of this feature, it is useful for System on Chip (SoC) designs.
- Its performance is really high such that it can work with 400 Mhz clock frequency at 0.13 micrometer technology.

• It supports power down mode for other CPUs in multiple CPU designs. It can also switch clock signal. These features make Leon3 more energy efficient when compared to other processor types.

2.2.1 Gaisler IP library

Gaisler Integrated Peripheral (IP) Library (GRLIB) contains reusable IP cores that designed for SoC development. The IP cores are centered around a common on-chip bus and use a coherent method for simulation and synthesis. The library is vendor independent, with support for different CAD tools and target technologies [19].

2.2.1.1 Library organization

GRLIB is organized around VHDL libraries, where each major IP (or IP vendor) is assigned a unique library name. Using separate libraries avoids name clashes between IP cores and hides unnecessary implementation details from the end user. Each VHDL library typically contains a number of packages, declaring the exported IP cores and their interface types. Simulation and synthesis scripts are created automatically by a global makefile. Adding and removing of libraries and packages can be made without modifying any global files, ensuring that modification of one vendor's library will not affect other vendors. A few global libraries are provided to define shared data structures and utility functions. GRLIB also provides automatic script generators for different simulators and implementation tools.[18]

2.2.2 On-chip bus

The most of the IP cores will be connected to the common bus and their organization is made as all IPs around this common bus. The Advanced Micro-controller Bus Architecture (AMBA) 2.0 Advanced High Speed Bus (AHB) / Advanced Peripheral Bus (APB) bus has been selected as the common on-chip bus, due to its market dominance and because it is well documented and can be used for free without license restrictions [18]. The figure below shows an example of a Leon3 system designed with GRLIB. It can be seen from 0 that high bandwidth required IP cores connected to AHB, but other IP cores which are not required as fast as first ones are connected to APB [18]. AHB and APB are connected to each other via AHB/APB bridge.

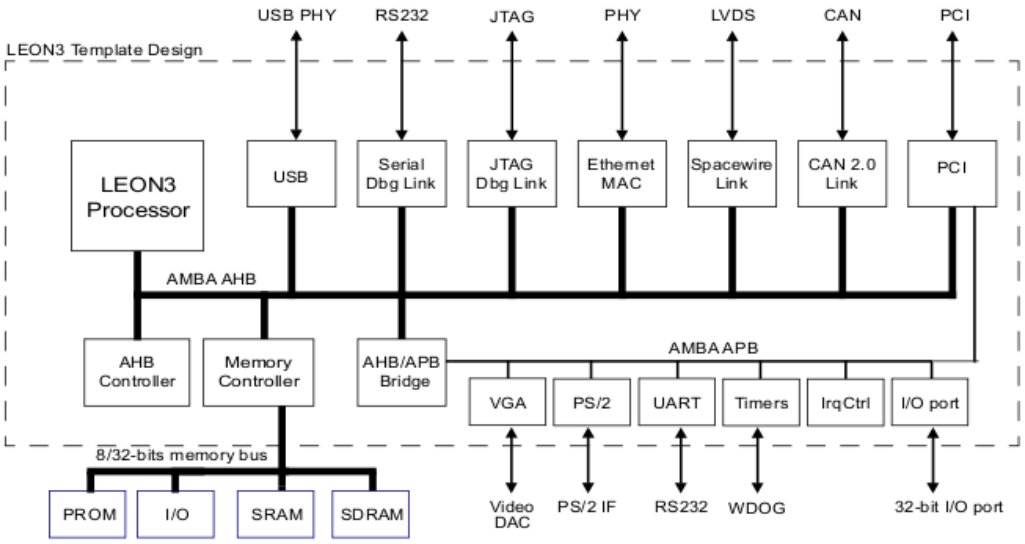


Figure 2.2 : General view of the Leon3 processor [18]

2.3 Gaisler Monitor

Gaisler Monitor (GRMON) is the debug tool that used with Leon processors and GRLIB oriented SoC designs. GRMON supports Universal Serial Bus (USB), Joint Test Action Group (JTAG), RS232, Ethernet, SpaceWire, and 32-bit Peripheral Component Interface (PCI) [20]. GRMON connected to the target device via one of these interfaces. All debug interfaces are acted as AHB masters on the target system with the debug protocol implemented in hardware. Because of this feature, debugging does not require any additional software. It can also detect hardware components of the target device and its addresses [20]. Figure 2.3 indicates conceptual view of the GRMON and Leon3 system that connected to it. C/C++ application or operating system can be loaded via GRMON. When GRMON connects to target device first time, it scans IP cores that system includes.

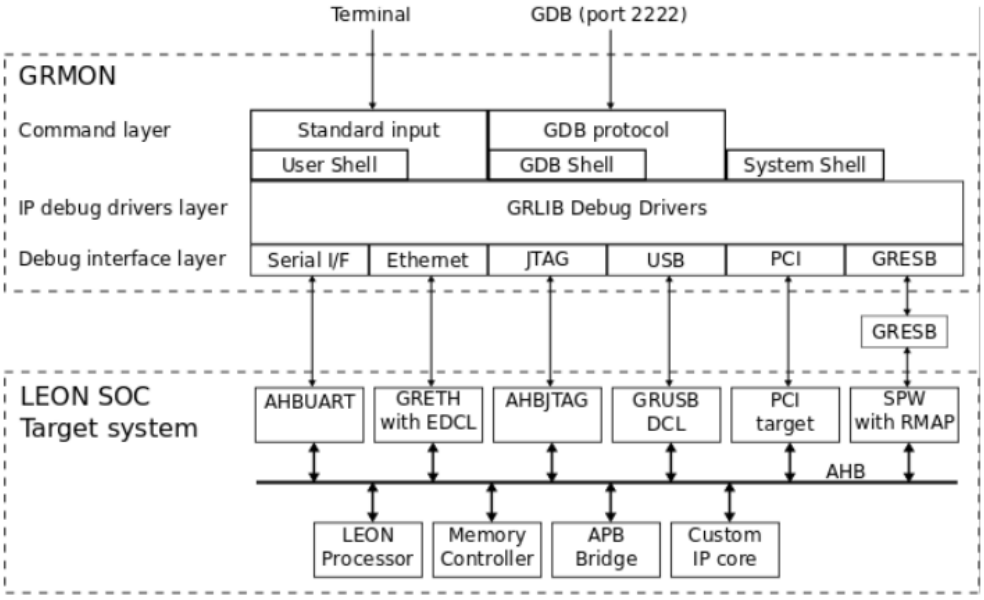


Figure 2.3 : GRMON Debug tool conceptual view [19]

2.4 Bare-C Cross Compiler

The Bare-C Cross Compiler (BCC) is developed for Leon3 processors. It compiles C and C++ applications and it supports Floating Point operations, SparcV8 multiply and divide instructions [21]. BCC is used to generate elf files which are loaded to the processor via GRMON. This cross-compiler is designed for Leon2, Leon3 and Leon4 processors. It is based on the GNU compiler tools, the newlib C library and a support library for programming Leon2, Leon3, and Leon4 systems. The cross-compiler allows compilation of C and C++ applications. Applications that are written with C programming language for testing or implementing of AES block chiper and complied with BCC2.

2.5 Atlys FPGA Board

The Atlys circuit board is a complete, ready-to-use digital circuit development platform based on a Xilinx Spartan-6 LX45 FPGA, speed grade is -3 [22]. The Atlys board has high-end peripherals including Gbit Ethernet, High-Definition Multimedia Interface (HDMI) Video, 128MByte 16-bit DDR2 memory, and USB and audio ports.

The Spartan-6 LX45 is optimized for high-performance logic and offers:

- 6,822 slices, each containing four 6-input LUTs and eight flip-flops
- 2.1Mbits of fast block RAM
- Four clock tiles (eight DCMs & four PLLs)
- Six phase-locked loops
- 58 DSP slices
- 500MHz+ clock speeds

The Atlys board includes Digilent's newest Adept USB2 system, which offers device programming, real-time power supply monitoring, automated board tests, virtual I/O, and simplified user-data transfer facilities [22].

In this study, configuration USB port, Basic I/O ports are used. HDMI Input / Output ports, Ethernet, DDR2, and other facilities of this board is not required and not used. General view of the Atlys board and used ports of it is shown in Figure 2.4

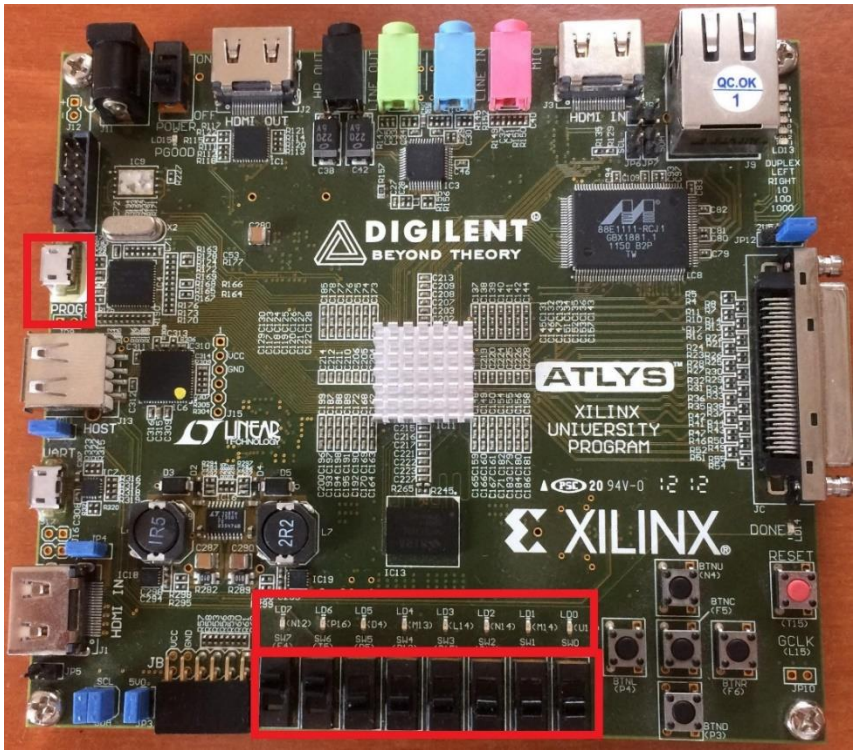


Figure 2.4 : General view of Atlys board and used ports

2.6 Advanced Encryption Standard

Rijndael algorithm had been developed by two cryptography expert Loan Daemen and Vincent Rijmen. In 2000, Rijndael algorithm is named as Advanced Encryption Standard (AES) [9] by National Institute of Standards and Technology (NIST) and it is approved as data encryption standard aimed to provide electronic data security.

Data Encryption Standard (DES) had been developed by IBM in early 1970s has 54-bit key [23]. Due to the theoretical weakness of it and new technologies on computer systems and integrated circuits, DES is seen as an unsecured algorithm in the 1990s. Thus, NIST started a competition for the development of new encryption standard and Rijdael algorithm is announced as the winner after 4 years.

2.6.1 Rijndael algorithm

Rijndael algorithm includes different cycles depend on the key length and key is recalculated with defined operations. These recalculated keys are used to generate encrypted data. AES is part of the symmetric key algorithms. The symmetric key is defined that same key is used to encrypt and decrypt data.

In this study, 128-bit version of Rijndael algorithm is used. This 128-bit data is rearranged as a 4x4 matrix which each cell represents 1-byte or equally 8-bit part of the 128-bit data. This matrix called “state” [9]. It is stated that number of cycles of the AES algorithm is defined with the length of the key. Security of data or in other words reliability of the encryption is increased by the increasing number of cycle. However, the number of operations and required memory to store key is increased in the meantime. Relationship between number of cycles and length of the key is given in Table 2.1.

Table 2.1 : Relationship between number of cycles and key length [23]

	Data (Block Size)	Key Length	Number of cycles
AES-128	128-bit	128-bit	10
AES-192	128-bit	192-bit	12
AES-256	128-bit	256-bit	14

Flow diagram of the AES block cipher is shown in Figure 5. In this diagram, Plain Matrix and Encrypted data are 4x4 matrices. Cycles can be defined by loop and number of cycles can take one of the values that mentioned before. It can be seen that MixColumn operation is not required for last cycle.

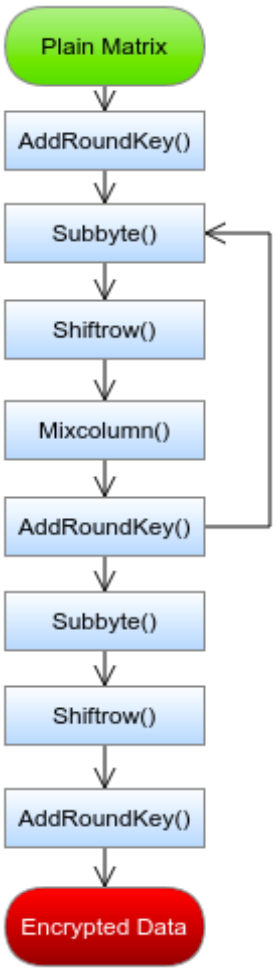


Figure 2.5 : Flow diagram of AES block chiper

2.6.2 Encryption

AES provide encryption of the 128-bit data on 4x4 matrix named plain or state. Thus, input data of the AES should be converted to matrix form. This conversation is shown with the following example. For example, input data is “32 43 F6 A8 88 5A 30 8D 31 31 98 A2 E0 37 07 34” in hexadecimal, a plain matrix of it is shown in Table 2.2.

Table 2.2 : Example Plain matrix

32	88	31	E0
43	5A	31	37
F6	30	98	07
A8	8D	A2	34

This matrix is used to generate encrypted data with the SubByte, ShiftRow, MixColumn, and AddRoundKey.

2.6.2.1 SubByte operation

SubByte is a nonlinear operation that conducts independent calculations on each byte (cell) of the plain matrix. Each cell in the plain matrix is changed its value to new value that determined from S-Box. S-Box is a special 16x16 matrix. S-Box matrix is shown in Table 2.3.

Table 2.3 : S-Box matrix for AES block chiper [24]

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 2.6 shows the example plain matrix and output matrix of SubByte operation.

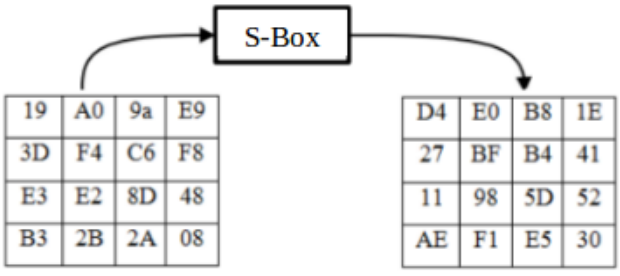


Figure 2.6 : Example of S-Box operation [24]

2.6.2.2 ShiftRow operation

In ShiftRow operation, the first row of a plain matrix is not shifted. Second row is shifted by 1 cell, third row is shifted by 2 cell, and fourth row is shifted by 3 cell. These shift operations are done to left. Example ShiftRow operation are shown in Figure 2.7.

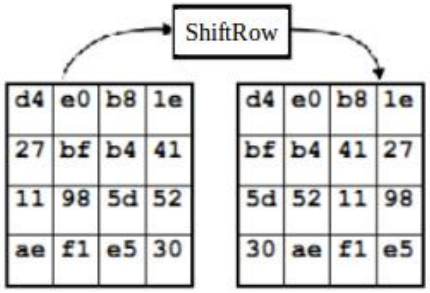


Figure 2.7 : Example of ShiftRow operation [24]

2.6.2.3 MixColumn operation

Firstly, rows of the plain matrix are used to generate columns of the plain matrix. Then, calculations are done on these columns with the matrix operation in Figure 2.8. As a result, output columns of the MixColumn operation is generated. Figure 2.9 indicates the example MixColumn operation.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

Figure 2.8 : Definition of the matrix operation for MixColumn [24]

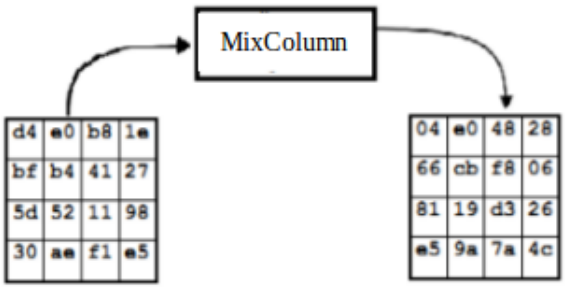


Figure 2.9 : Example of the MixColumn operation [24]

2.6.2.4 AddRoundKey operation

AES Algorithm requires different keys for each round and generation of these different keys is described next part. AddRoundKey operation is represented addition operation on plain matrix and generated key matrix. This addition is a bitwise XOR operation. Thus, key generation requires a defined number of cycles and each round key calculated from previous keys.

2.6.3 Key generation

Round keys are different from each other and each of them is generated one of these cases. These cases are on the fly key expansion that calculation of round keys is done with encryption at the same time. Other case is pre-computed key schedule.

For the key generation, previous round key is used. Firstly, previous column and 4th previous column are threated XOR operation. In Figure 2.10, T operation is used to calculate first column of the round key. T operation includes ShiftRow, SubByte, and XOR with the Rc(x) vector value. Figure 2.11 shows operations that are used for T operation. Rc(x) vector values that are associated with the round number are also listed in Table 2.4

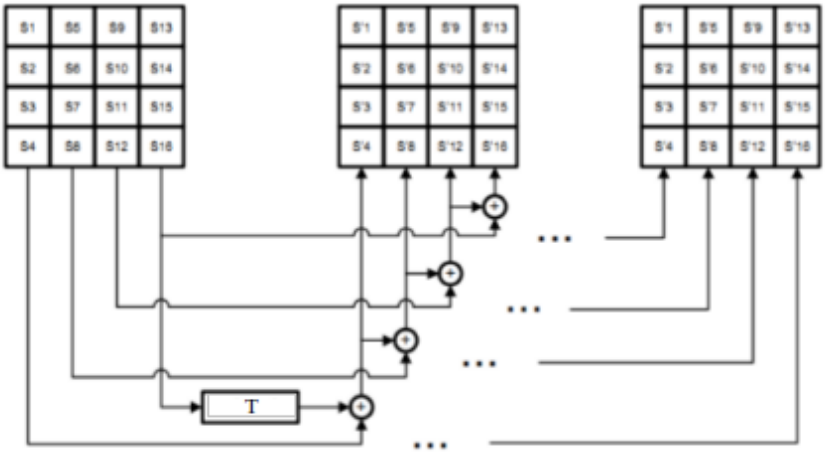


Figure 2.10 : General description of the key generation [24]

Table 2.4 : Value of the Rc(x) vectors [24]

Round Number	Rc(x) Value
1	01 00 00 00
2	02 00 00 00
3	04 00 00 00
4	08 00 00 00
5	10 00 00 00
6	20 00 00 00
7	40 00 00 00
8	80 00 00 00
9	1B 00 00 00
10	36 00 00 00

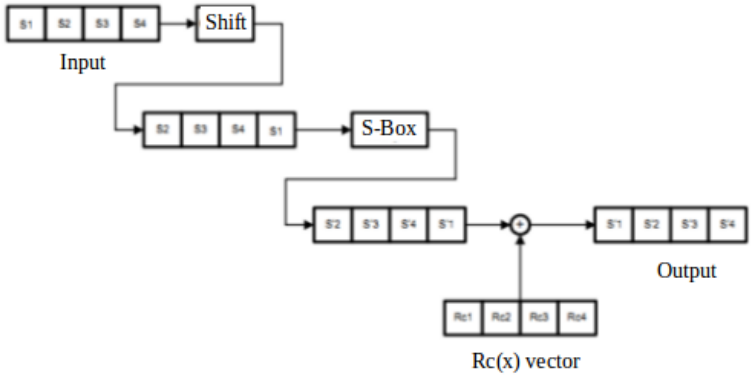


Figure 2.11 : Description of the T operation in key generation [24]

2.7 Implementation

2.7.1 Implementation of Leon3

Implementation of Leon3 can be conducted on Linux operating system or Windows with Cygwin tool. For this study, Linux operating system is selected and Ubuntu 16.04 is installed.

Firstly, one of the Xilinx ISE or Vivado development environments should be downloaded and installed. These tools can be obtained with free WebPack license from Xilinx website [25]. Xilinx ISE Design Suite 14.7 is used for this study. A folder named Xilinx should be created before starting the installation. Write and Read permissions should also be given to this folder before installation. Because ISE installation is started with `/opt/Xilinx` directory as the location of the installation. Read/Write permission are given with the following command.

```
sudo chmod 777 /opt/Xilinx
```

When Xilinx ISE installed, cable drivers are not installed via installation phase. Thus, cable drivers should be installed after Xilinx ISE installation.

After installation of Xilinx ISE and cable drivers is completed, GRLIB Library file is downloaded from Cobham Gaisler website [26]. This file should be ended with "tar.gz". "grlib-gpl-2017.3-b4208" version of GRLIB is used for this study. This file can be installed in any location on the host system via "`tar xvf`" command. In the executed file, GRLIB manual and other supporting documents can be found in doc folder. Xilinx ISE bin folder should be added to PATH environment with the export command or changing the "environment" document in `/etc` directory in Ubuntu. GRLIB and Xilinx ISE directories should be added to environment variables. Setting up the environment variables and new Paths via "`export`" command must be rewritten when new terminal opened. Examples of export commands;

```
export XILINX=/opt/Xilinx/14.7/ISE_DS/ISE
```

```
export GRLIB=/home/cihan/Bitirme/Leon/grlib-gpl-2017.3-b4208
```

```
export PATH=$PATH:$XILINX/bin/nt
```

These environment variables can also be set via environment document in `/etc` folder. When new environment variables and Paths have added this document and it runs after addition not requires repeated calling of "`export`" command. Thus, changing the "environment" document should be preferred. The document should be opened with root permissions. An example command to open this document shown below.

```
sudo nano /etc/environment
```

When document is opened, new environment variable is added as follows,

```
XILINX="/opt/Xilinx/14.7/ISE_DS/ISE"
```

```
GRLIB="/home/cihan/Bitirme/Leon/grlib-gpl-2017.3-b4208"
```

New PATH variables are also added at the end of the PATH variable section with ":". Example of this shown below.

```
:/opt/Xilinx/14.7/ISE_DS/ISE/bin/nt"
```

To finalize these steps changed `/etc/environment` document must be run. For this, user should enter the `/etc` directory and run the following command.

```
. environment
```

After these variables are set, it is time to go into the GRLIB file and implement Leon3 for the desired board. GRLIB supports many FPGAs such as Spartan3, Spartan6, Virtex2, Virtex4, Virtex6, Zynq etc. [18]. In this study, Atlys board that have Spartan6 XC6SLX45-CSG324C FPGA is used. After the target board is selected, the folder that contains required design files opened like the following command.

```
cd Bitirme/Leon/grlib-gpl-2017.3-b4208/designs/leon3-digilent-atlys
```

In design folder, Graphical User Interface (GUI) of the configuration file is opened with "`make xconfig`" command. This GUI provides a user friendly interface of the configuration file of Leon3 design. User can be changed several quantities of the Leon3 design such as Integer Unit (IU), Debug Support Unit (DSU), Memory Management Unit (MMU), Floating Point Unit (FPU), SDRAM controller, Ethernet, Universal Asynchronous Receiver-Transmitter (UART), Timer etc. If the configuration is wanted to save, save and exit button can be used. If another configuration file wants to be loaded, any configuration file that arranged before can be used. Example view of the configuration GUI is shown in Figure 2.12

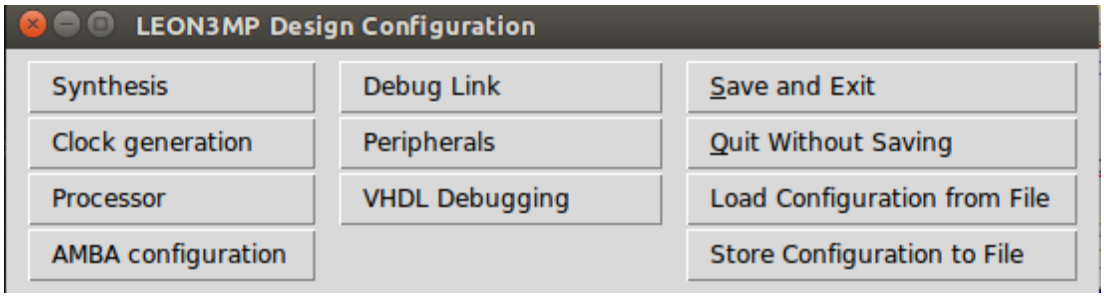


Figure 2.12 : Example view of the main page of configuration GUI

After configuration is done, required scripts generated by "`make scripts`" command. This command will create an XML project file (TOP.xise). Before using "`make ise`" or "`make ise-lauch`" the executable of the Xilinx ISE-14.7 should be indicated with the source command. Example of this operation shown below.

```
source /opt/Xilinx/14.7/ISE_DS/settings64.sh
```

When executing "`make ise-launch`", this XML will be used to launch the ISE project manager. Synthesis and place&route can also be run in batch mode (preferred option) using "`make ise`" for the XST flow and "`make ise-synp`" for synplify flow. Many Xilinx FPGA boards are supported in GRLIB, and can be re-programmed using "`make ise-prog-fpga`" and "`make ise-prog-prom`". All possible targets of the "make" command are listed in Figure 2.13.

Make target	Description
ise	Synthesize and place&route design with XST in batch mode
ise-prec	Synthesize and place&route design with Precision in batch mode
ise-synp	Synthesize and place&route design with Synplify in batch mode
ise-launch	Start project navigator interactively using XST flow
ise-launch-synp	Start project navigator interactively using EDIF flow
ise-map	Synthesize design with XST in batch mode
ise-prog-fpga	Program FPGA on target board using JTAG
ise-prog-fpga-ref	Program FPGA on target board with reference bit file
ise-prog-prom	Program configuartion proms on target board using JTAG
ise-prog-prom-ref	Program configuartion proms with reference bit file
install-unisim	Install Xilinx UNISIM libraries into GRLIB
remove-unisim	Remove Xilinx UNISIM libraries from GRLIB
install-secureip	Install Xilinx SecureIP files into GRLIB
remove-secureIP	Remove Xilinx SecureIP files from GRLIB
install-unimacro	Install Xilinx UNIMACRO files into GRLIB (requires install-unisim)
remove-unimacro	Remove Xilinx UNIMACRO files from GRLIB
install-unisim_ver	Install Verilog version of UNISIMS into GRLIB
install-xilinxcorelibs_ver	Install Verilog version of Xilinx CoreLibs into GRLIB
install-secureip_ver	Install Verilog version of SecureIP into GRLIB (secureip_ver)

Figure 2.13 : Possible targets of the make command [18]

With the "`make ise`" command, synthesize and place&route are done. Moreover, ISE project manager will be launched with "`make ise-launch`" command and synthesize, place&route are done from ISE project manager. This process continues for 30 minutes long. Bitstream that is used to program target device is generated after synthesize and place&route finished. Atlys board can be programmable with the IMPACT tool of the ISE project manager via program input of it.

2.7.2 Bare-C Cross Complier installiation

Bare-C Cross Complier is used to design C/C++ application and it generates **elf** files are loaded into the target device. BCC has its own assembler and disassembler [21]. The latest version of BCC can be downloaded from Cobham Gaisler's website [27].
In order to recompile BCC from sources, "automake-1.11.1" and "autoconf-2.68" is required [21]. Thus, automake-1.11.1 and autoconf-2.68 commands must be checked. If they are not installed, installation is done with following commands. This installation should be done with root permissions. Required commands for the installation of "automake" and" autoconf" is given below.

```
sudo apt-get install autoconf
```

```
sudo apt-get install automake
```

After obtaining the compressed tar file for the binary distribution, uncompress and untar it to a suitable location. The Linux version of BCC has been prepared to reside in the `/opt/bcc-2.0.2-gcc/` directory. Thus, downloaded tar file should be moved to `/opt` directory. Before the movement of "tar" file, read/write permissions of `/opt` directory must be given. If it is not given, tar file does not be moved to `/opt` directory. After these stages, user should change the current directory to `/opt`. Then, the distribution can be installed with the following command:

```
tar -C /opt -xf /opt/bcc-2.0.2-gcc-linux64.tar.xz
```

After the compiler is installed, add `/opt/bcc-2.0.2-gcc/bin` directory to the executables search path (PATH) and `/opt/bcc-2.0.2-gcc/man` directory to the

manual page path (MANPATH). This can be done with the "`export`" command or change the `/etc/environment` document as described before.

The binary installation of BCC contains the following sub-directories:

bin/	Executables
doc/	GNU, newlib and BCC documentation
man/	Manual pages for GNU tools
sparc-gaisler-elf/	SPARC target libraries, include files and LEON BSP
sparc-gaisler-elf/bsp/	Board Support Packages for LEON systems
src/	Various sources, examples and make scripts
src/examples/	BCC example applications
src/libbcc/	libbcc source code and make scripts

The following tools are installed with BCC:

sparc-gaisler-elf-addr2line	Convert address to C/C++ line number
sparc-gaisler-elf-ar	Library archiver
sparc-gaisler-elf-as	Cross-assembler
sparc-gaisler-elf-c++	C++ cross-compiler
sparc-gaisler-elf-c++filt	Utility to demangle C++ symbols
sparc-gaisler-elf-cpp	The C preprocessor
sparc-gaisler-elf-g++	Same as sparc-gaisler-elf-c++
sparc-gaisler-elf-gcc	C/C++ cross-compiler
sparc-gaisler-elf-gcov	Coverage testing tool
sparc-gaisler-elf-gdb	GNU GDB C/C++ level Debugger
sparc-gaisler-elf-gprof	Profiling utility
sparc-gaisler-elf-ld	GNU linker
sparc-gaisler-elf-nm	Utility to print symbol table
sparc-gaisler-elf-objcopy	Utility to convert between binary formats
sparc-gaisler-elf-objdump	Utility to dump various parts of elf files
sparc-gaisler-elf-ranlib	Library sorter
sparc-gaisler-elf-readelf	ELF file information utility
sparc-gaisler-elf-size	Utility to display segment sizes

sparc-gaisler-elf-strings	Utility to dump strings from elf files
sparc-gaisler-elf-strip	Utility to remove symbol table

The source code for the BCC 2.0.2 toolchain is distributed in an archive named "bcc-2.0.2-src.tar.bz2", available on the Cobham Gaisler website [28]. It contains source code for the target C library and the host compiler tools (binutils, GCC, GDB). Installing the source code is optional but recommended when debugging applications using the C standard library [21]. The target libraries have been built with debug information making it possible for GDB to find the sources files. It allows, for example, to step through the target C standard library code. The tar file should be located into `/opt/bcc-2.0.2-gcc/src`. The sources can be installed by extraction the source distribution archive "bcc-2.0.2-src.tar.bz2" into `/opt/bcc-2.0.2-gcc/src`. The required command is given below.

```
tar xf bcc-2.0.2-src.tar.bz2
```

For building this source code, a script named "ubuild.sh" which located into a currently extracted bcc-2.0.2 folder. To build and install the BCC compiler tools, GDB and the C library in `/tmp/bcc-2.0.2-local`, the following step shall be performed:

```
./ubuild.sh --destination /tmp/bcc-2.0.2-local --toolchain –gdb
```

The GCC front-end, "`sparc-gaisler-elf-gcc`" command has been modified to support the following options specific to BCC and LEON systems:

`-qbsp=bspname`	Use target libraries, startup files and linker scripts for a specific LEON system. The parameter bspname corresponds to a Board Support Package (BSP). A description of the BSPs distributed with BCC is given in Chapter 7. The BSP leon3 is used as default if the -qbsp= option is not given.
`-qnano`	Use a version of the newlib C library compiled for reduced foot print. Thenano version implementations of the fprintf(), fscanf()

family of functions are not fully C standard compliant. Code size can decrease with up to 30 KiB when printf() is used.

`-qsvt` Use the single-vector trap model described in SPARC-V8 Supplement, SPARC-V8 Embedded (V8E) Architecture Specification.

Useful (standard) options are:

`-g` Generate debugging information should be used when debugging with GDB.

`-msoft-float` Emulate floating-point - must be used if no FPU exists in the system.

`-O2 or -Os` Optimize for maximum performance or minimal code size.

`-Og` Optimize for maximum debugging experience.

`-mcpu=leon3` Generate Sparc V8 code. Includes support for the casa instruction.

General development flow of new application such as compilation and debugging of applications is typically done with the following steps,

1. Compile and link the program with GCC
2. Debug program using a simulator (GDB connected to TSIM)
3. Debug program on remote target (GDB connected to GRMON)
4. Create boot-prom for a standalone application with mkprom2

In this study, second and fourth steps are not included. After BCC is installed, simple C code can be compiled and generated **elf** file. For simplicity, “Hello World” example is used. Examples of BCC is located under the following directory:

```
/opt/bcc-2.0.2-gcc/src/examples/
```

In this folder, several C/C++ and mkprom2 examples can be found. Example “Hello World” C code are given below.

```
#include <stdlib.h>
```

```
#include <stdio.h>

int main(void)
{
        printf("hello, world\n");
        return EXIT_SUCCESS;
}
```

This program basically print "*hello, world*" string to the terminal. To compile this program following command is used.

```
sparc-gaisler-elf-gcc-7.2.0 -mcpu=leon3 -qbsp=leon3 -msoft-float -qnano -qsvt hello.c -o hello.exe
```

First part of the command defined complier option and GCC complier is chosen. With the first option `-mcpu= leon3` complier generate SparcV8 code and it includes support for the casa instruction. Second option `-qbsp=leon3` defines board supporting package. Third option `-msoft-float` is used because defined on board system does not contain FPU. When fourth option `-qsvt` is used, complier used single-vector trap model described in SparcV8. Enable or disable decision of the single vector trapping can be set into the configuration. After these options for input C file declared. Output elf file name defined and it is generated with `-o` option.

After a simple program is compiled and it generates **elf** file, next step is the loading and debugging this application with GRMON debug tool.

2.7.3 GRMON installiation

GRMON is a general debug monitor for the Leon processor, and for SoC designs based on the GRLIB IP library [20]. GRMON can be downloaded from Cobham Gaisler website [29]. It includes the following functions:

• Read/write access to all system registers and memory
• Built-in disassembler and trace buffer management
• Downloading and execution of LEON applications
• Breakpoint and watchpoint management
• Remote connection to GNU debugger (GDB)

• Support for USB, JTAG, RS232, PCI, Ethernet and SpaceWire debug links
• Tcl interface (scripts, procedures, variables, loops etc.)

GRMON is serviced in two options; Evaluation/Academic or Professional version [20]. GRMON Evaluation/Academic version is used in this study. To install GRMON, extract the archive anywhere on the host computer. The archive contains a directory for each operating system (OS) that GRMON supports. Each OS folder contains additional directories as described in the list below.

```
grmon-eval-2.0.87/<OS>/bin
grmon-eval-2.0.87/<OS>/lib
grmon-eval-2.0.87/<OS>/share
```

The bin directory contains the executable. For convenience, it is recommended to add the bin directory of the host operating system to the environment variable PATH. This can be done with "`export`" command or change the `/etc/environment` document as described before.

GRMON can be found in the directory named "share". GRMON will try to automatically detect the location of the folder. A warning will be printed when starting GRMON if it fails to find the shared folder. If it fails to automatically detect the folder, then the environment variable GRMON_SHARE can be set to point the `share/grmon` directory. This environment variable can be set to `/opt/grmon-eval-2.0.87/linux64/share/grmon/` via "`export`" command or change `/etc/environment` document described before.

The lib directory contains some additional libraries that GRMON requires. GRMON will fail to start because of some missing libraries that are located in this directory, then add this path to the environment variable LD_LIBRARY_PATH. `/opt/grmon-eval-2.0.87/linux64/lib` directory can be added to environment variables via "`export`" command or change the `/etc/environment` document described before. An interactive GRMON debug session typically consists of the following steps. Steps 2 through 6 are performed using the GRMON terminal interface.

1. Starting GRMON and attaching to the target system
2. Examining the hardware configuration
3. Uploading application program
4. Setup debugging, for example, insert breakpoints and watchpoints
5. Executing the application
6. Debugging the application and examining the CPU and hardware state

The target device can be connected to host computer via several connections such as serial debug link, Ethernet debug link, JTAG debug link etc. Serial debug link is used for this study. GRMON is starting by giving the "`grmon`" command in a terminal window. For simplicity, current directory should be a directory of the file that is loaded to the target device. Else, full directory of the **elf** file should be indicated when the application is loaded to the target device.

The general options are mostly targeted independent options configuring the behavior of GRMON. Some of them affect how the target system is accessed both during connection and during the whole debugging session. All general options are described below.

`-abaud baudrate`

Set baud-rate for all UARTs in the system, (except the debug-link UART). By default, 38400 bauds is used.

`-ambamb [maxbuses]`

Enable auto-detection of AHBCTRL_MB system and (optionally) specifies the maximum number of buses in the system if an argument is given. The optional argument to -ambamb is decoded as below:

0, 1: No Multi-bus (MB) (max one bus)

2..3: Limit MB support to 2 or 3 AMBA PnP buses

4 or no argument: Selects Full MB support

`-c filename`

Run the commands in the batch file at start-up.

`-cfg filename`

Load fixed PnP configuration from a xml-file.

`-echo`

Echo all the commands in the batch file at start-up. Has no effect unless -c is also set.

`-edac`

Enable EDAC operation in memory controllers that support it.

`-freq sysclk`

Overrides the detected system frequency. The frequency is specified in MHz.

`-gdb [port]`

Listen for GDB connection directly at start-up. Optionally specify the port number for GDB communications. Default port number is 2222.

`-ioarea address`

Specify the location of the I/O area. (Default is 0xfff00000).

`-log filename`

Log session to the specified file. If the file already exists, the new session is appended. This should be used when requesting support.

`-ni`

Read plug&play and detect all system device, but don't do any target initialization.

`-nopnp`

Disable the plug&play scanning. GRMON won't detect any hardware and any hardware dependent functionality won't work.

`-nothreads`

Disable thread support.

`-u [device]`

Put UART 1 in FIFO debug mode if hardware supports it, else put it in loop-back mode. Debug mode will enable both reading and writing to the UART from the monitor console. Loop-back mode will only enable reading. The optional device parameter is used to select a specific UART to be put in debug mode. The device parameter is an index starting with 0 for the first UART and then increasing with one in the order they are found in the bus scan. If the device parameter is not used the first UART is selected.

`-udm [device]`

Put UART 1 in FIFO debug mode if hardware supports it. Debug mode will enable both reading and writing to the UART from the monitor console. The optional device parameter is used to select a specific UART to be put in debug mode. The device parameter is an index starting with 0 for the first UART and then increasing with one in the order they are found in the bus scan. If the device parameter is not used the first UART is selected.

`-ulb [device]`

Put UART 1 in loop-back mode. Loop-back mode will only enable reading from the UART o the monitor console. The optional device parameter is used to select a specific UART to be put in debug mode. The device parameter is an index starting with 0 for the first UART and then increasing with one in the order they are found in the bus scan. If the device parameter is not used the first UART is selected.

`-ucmd filename`

Load script specified by filename into all shells, including the system shell.

GRMON is starting with the "`grmon -u -digilent`" command in this study. First option `-u` indicates that debug mode enables both reading and writing to the UART from the monitor console. Second option `-digilent` used to connection to the target device and it describes connection type. Output of first connection into the GRMON console are shown in Figure 2.14.

```
cihan@cihan-K56CB:/opt/bcc-2.0.2-gcc/src/examples/hello$ grmon -u -digilent
  GRMON2 LEON debug monitor v2.0.87 64-bit eval version

  Copyright (C) 2017 Cobham Gaisler - All rights reserved.
  For latest updates, go to http://www.gaisler.com/
  Comments or bug-reports to support@gaisler.com

  This eval version will expire on 02/05/2018

JTAG chain (1): xc6slx45
  GRLIB build version: 4208
  Detected frequency:  50 MHz

  Component                            Vendor
  LEON3 SPARC V8 Processor             Cobham Gaisler
  LEON3 SPARC V8 Processor             Cobham Gaisler
  JTAG Debug Link                      Cobham Gaisler
  GR Ethernet MAC                      Cobham Gaisler
  LEON2 Memory Controller              European Space Agency
  AHB/APB Bridge                       Cobham Gaisler
  LEON3 Debug Support Unit             Cobham Gaisler
  SPI Memory Controller                Cobham Gaisler
  Single-port DDR2 controller          Cobham Gaisler
  Generic AHB ROM                      Cobham Gaisler
  Single-port AHB SRAM module          Cobham Gaisler
  Generic UART                         Cobham Gaisler
  Multi-processor Interrupt Ctrl.      Cobham Gaisler
  Modular Timer Unit                   Cobham Gaisler
  PS2 interface                        Cobham Gaisler
  PS2 interface                        Cobham Gaisler
  VGA controller                       Cobham Gaisler
  General Purpose I/O port             Cobham Gaisler
  AHB Status Register                  Cobham Gaisler

  Use command 'info sys' to print a detailed report of attached cores

grmon2>
```

Figure 2.14 : GRMON console view of first connection

When connecting for the first time it is essential to verify that GRMON has auto-detected all devices and their configuration correctly. At start-up, GRMON will print the cores and the frequency detected. "`info sys`" command shows all system elements with their start addresses, AHB numbers, Interrupt request number etc. Example output of the "`info sys`" command of two CPU Leon3 processor are shown below;

```
grmon2> info sys
  cpu0      Cobham Gaisler  LEON3 Sparc V8 Processor
            AHB Master 0
  cpu1      Cobham Gaisler  LEON3 Sparc V8 Processor
            AHB Master 1
  ahbjtag0  Cobham Gaisler  JTAG Debug Link
            AHB Master 2
  greth0    Cobham Gaisler  GR Ethernet MAC
            AHB Master 3
            APB: 80000E00 - 80000F00
            IRQ: 12
            edcl ip 192.168.0.51, buffer 2 kbyte
  mctrl0    European Space Agency  LEON2 Memory Controller
            AHB: 00000000 - 20000000
            APB: 80000000 - 80000100
            8-bit prom @ 0x00000000
  apbmst0   Cobham Gaisler  AHB/APB Bridge
            AHB: 80000000 - 80100000
  dsu0      Cobham Gaisler  LEON3 Debug Support Unit
            AHB: 90000000 - A0000000
            AHB trace: 256 lines, 32-bit bus
            CPU0:  win 8, hwbp 2, itrace 256, V8 mul/div, srmmu, lddel 1
                   stack pointer 0x40003ff0
                   icache 2 * 4 kB, 16 B/line
                   dcache 2 * 4 kB, 16 B/line
            CPU1:  win 8, hwbp 2, itrace 256, V8 mul/div, srmmu, lddel 1
                   stack pointer 0x40003ff0
                   icache 2 * 4 kB, 16 B/line
                   dcache 2 * 4 kB, 16 B/line
  spim0     Cobham Gaisler  SPI Memory Controller
            AHB: FFF00200 - FFF00300
            AHB: E0000000 - E1000000
```

```
            IRQ: 11
            SPI memory device read command: 0x03
  ddr2spa0  Cobham Gaisler  Single-port DDR2 controller
            AHB: 40000000 - 48000000
            AHB: FFF00100 - FFF00200
            No SDRAM found
  adev9     Cobham Gaisler  Generic AHB ROM
            AHB: 00000000 - 00100000
  ahbram0   Cobham Gaisler  Single-port AHB SRAM module
            AHB: 40000000 - 40100000
            32-bit static ram: 16 kB @ 0x40000000
  uart0     Cobham Gaisler  Generic UART
            APB: 80000100 - 80000200
            IRQ: 2
            Baudrate 38343, FIFO debug mode
  irqmp0    Cobham Gaisler  Multi-processor Interrupt Ctrl.
            APB: 80000200 - 80000300
  gptimer0  Cobham Gaisler  Modular Timer Unit
            APB: 80000300 - 80000400
            IRQ: 8
            8-bit scalar, 2 * 32-bit timers, divisor 50
  ps2ifc0   Cobham Gaisler  PS2 interface
            APB: 80000400 - 80000500
            IRQ: 4
  ps2ifc1   Cobham Gaisler  PS2 interface
            APB: 80000500 - 80000600
            IRQ: 5
  adev16    Cobham Gaisler  VGA controller
            APB: 80000600 - 80000700
  gpio0     Cobham Gaisler  General Purpose I/O port
            APB: 80000A00 - 80000B00
  ahbstat0  Cobham Gaisler  AHB Status Register
            APB: 80000F00 - 80001000
            IRQ: 7
```

Before the application is loaded to the target device via GRMON, the DSU should be activated from the configuration of Leon3 and it should also be activated on the hardware side. According to “README” document in Atlys design folder, DSU-Enable signal is mapped to switch SW7 and it should be activated. Otherwise, any

application is not loaded to the target device. Status of the DSU can be checked with the "`info sys`" command. Figure 2.15 shows the example output of the "`load`" command that is used to upload a Leon3 software application to the target system memory.

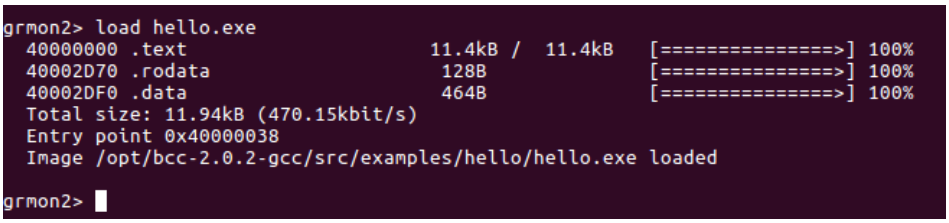


Figure 2.15 : GRMON console view of a `load` command

The "`verify`" command can be used to make sure that the file has been loaded correctly to memory as in Figure 2.16. Any discrepancies will be reported in the GRMON console.

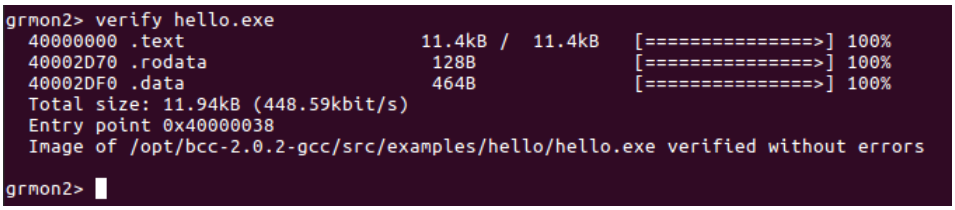


Figure 2.16 : GRMON console view of a `verify` command

After the application has been uploaded to the target with "`load`" command, the "`run`" command can be used to start execution. The entry-point taken from the **elf**-file during loading will serve as the starting address, the first instruction executed. The "`cont`" command resumes execution after a temporary stop, e.g. a breakpoint hit. The "`go`" command also affects the CPU execution, the difference compared to "`run`" command is that the target device hardware is not initialized before starting execution.

The output from the application normally appears on the Leon3 UARTs, not on the GRMON console. However, if GRMON is started with the "`-u`" option as in this study, the UART is put into debug mode and the output is tunneled over the debug-link and finally printed on the console by GRMON. Since the application changes (at least) the `.data` segment during run-time the application must be reloaded before it can be executed again. If the application uses the MMU or installs data exception handlers, GRMON should be started with "`-nb`" option to avoid going into break mode on a

page-fault or data exception. To prevent GRMON from interpreting it as its own breakpoints and stop the CPU one must use the "`-nswb`" option.

Breakpoints are inserted with the "`bp`" command. The subcommand (`soft, hard, watch, bus, data, delete`) given to "`bp`" determine which type of breakpoint is inserted, if no subcommand is given `bp` defaults to a software breakpoint. "`bp`" command is used to observe register changes when program run. The current value of the registers into breakpoints can be displayed with the "`reg`" command and name of the register such as g1, g2, o1, f1. If "`reg`" command is used without special register name, it displays current register window of a Leon3 processor. Example output of "`reg`" command is shown in Figure 2.17.

```
grmon2> reg
        INS       LOCALS    OUTS      GLOBALS
    0:  00000000  F34000C6  FFFFFFFF  00000000
    1:  40001E38  40002A90  00000000  00000001
    2:  FFFFFFFF  40002A94  0000000D  40002D24
    3:  00000000  400002A8  00000004  00000000
    4:  00000000  00000001  00000003  40003000
    5:  00000000  00000000  40002F5C  00000000
    6:  40003F30  00000080  40003ED0  00000000
    7:  400014B0  00000007  40001BC4  00000000

  psr: F34000C6  wim: 00000002  tbr: 40000800  y: 00000000

  pc:  400002A8  ta  0x0
  npc: 400002AC  call  0x40000628


grmon2>
```

Figure 2.17 : GRMON console view of `reg` command

2.7.4 Led test application

End of the installation of GRLIB, BCC2 and GRMON, the hardware of the implemented Leon3 processor can be tested with a simple application on and off LEDs on the Atlys board. First, the General Purpose Input/Output (GPIO) port declaration should be examined.

32-bit GPIO port is divided into parts such that, LEDs LD0 to LD5 are mapped to GPIO bits 0 to 5, switches SW0 to SW5 are mapped to GPIO bits 8 to 13, buttons BTNU, BTNL, BTND, BTNR, BTNC are mapped to GPIO bits 16 to 20 the PMODA port is mapped to GPIO bits 24 to 31. In order to set LEDS as output, GPIO registers should be set appropriately. Registers of GPIO port is defined in the GRLIB IP core User's Manual that can be found in `grlib-gpl-2017.3-b4208/doc/` folder.

According to manual required GPIO port registers for this application defined in Table 2.5.

Table 2.5 : Required part of the GPIO registers [30]

APB address offset	Register
0x00	I/O port data register
0x04	I/O port output register
0x08	I/O port direction register

Other registers such as interrupt mask, interrupt available, bypass etc. are also given in this manual. However, this application is required only these three registers for implementation. The base register of the GPIO is data register and it is set to 0x80000A00. This 32-bit hexadecimal value is set before and it can be found in the output of the "`info sys`" command on GRMON console. However, in this example, any data is not used as an input. The GPIO output register is set to 0x80000A04 because, the output register is declared to 0x04 according to GRIP manual. This value added to the base address of the GPIO port which is 0x80000A00. The GPIO direction register is set as 0x80000A08 because 0x08 is also added to the base address of the GPIO port. GPIO direction register is loaded with 0x0000003f to set all LEDs as an output. The data register is not used because any input does not necessary for this application. The output register can be set to different values depending on users' preference. For this example, the output register is set to 0x00000015. This configuration sets LEDs as one on and one off repeatedly. C code of LED test application is shown below.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
	// setting up registers of GPIO
		unsigned int *gpio_base = (int *) 0x80000A00;
	unsigned int *gpio_out = (int *) 0x80000A04;
	unsigned int *gpio_dir = (int *) 0x80000A08;

	//All LEDs are setted as output
	*gpio_dir = 0x0000003F;
```

```
        //010101
        *gpio_out = 0x00000015;

          return EXIT_SUCCESS;
}
```

This C code is compiled with BCC2 with the following command and it generates **elf** file is named “blink_led.exe”.

```
sparc-gaisler-elf-gcc-7.2.0 -mcpu=leon3 -qbsp=leon3 -msoft-float -qnano -qsvt blink_led.c -o blink_led.exe
```

After compilation is completed, **elf** file is loaded to Atlys board via GRMON. The loaded application is run with “`run`” command and its output is seen with LEDs. Figure 2.18 shows the example output of Led Test application. Two LEDs, LD7 and LD6 are dedicated to debug tool options. Thus, these LEDs cannot be used as GPIO port and their values are determined by the processor itself.

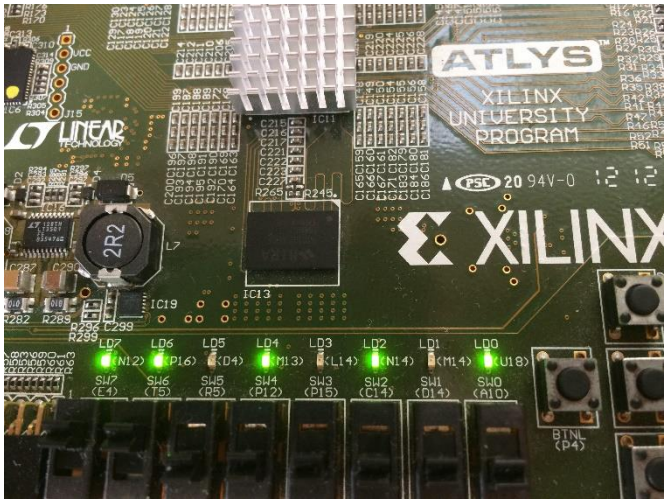


Figure 2.18 : Output of the Led Test application

2.7.5 Test of basic operations

For the Instruction Set Extension of Leon3 processor, current instructions of the Leon3 processor which rely on SparcV8 ISA is analyzed. First, main ALU operations are considered. Some of these operations are ADD, ADDC (add with carry), SUB, AND, OR, NOT, XOR etc. Some of the ALU operations set zero or carry as an output. Because of the implementation of AES block cipher, Carry or Zero flags are not

required for any a part of this block cipher hardware. In Leon3, ALU operations, flags, fetch-decode-execute cycle operations are done in integer unit that is a VHDL file named “iu3.vhd”. When this file is analyzed, it is recognized that logical operations is the simplest ALU operation that not required any other flag or internal signal can affect another part of integer unit. Thus, the addition of the new instructions can be done similar to available logic operations. Because of these reasons, basic logical operations like AND, OR, NOT, XOR is tested. First, C codes of these operations are written. Example C code of AND operation are shown below.

```
#include <stdio.h>
int main(void)
{
      int x, y, z;
      x = 6;
      y = 4;
      z = x & y;
      return 0;
}
```

Then, these codes are compiled by BCC2 and generate **elf** file. Then, this file is disassembled with the command that disassembles the **elf** file generated by BCC2. Example disassemble command is shown below.

```
sparc-gaisler-elf-objdump -D and.exe > and.asm
```

“`-D`” option is used for disassemble all parts of the **elf** file. “`-d`” option can also be used to disassemble only .data and .text parts of elf file. “`> and.asm`” is used to write disassembled elf file to other text file named “`and.asm`”. Without this part, disassembled elf file is shown on the terminal. Thanks to the disassembled elf file, assembly code of the written C code can be tracked. Disassembled main section of “`and.asm`” are shown below.

```
40000280 <main>:
40000280:   9d e3 bf 90       save  %sp, -112, %sp
40000284:   82 10 20 06       mov  6, %g1
40000288:   c2 27 bf fc       st  %g1, [ %fp + -4 ]
```

```
4000028c:   82 10 20 04       mov  4, %g1
40000290:   c2 27 bf f8       st  %g1, [ %fp + -8 ]
40000294:   c4 07 bf fc       ld  [ %fp + -4 ], %g2
40000298:   c2 07 bf f8       ld  [ %fp + -8 ], %g1
4000029c:   82 08 80 01       and  %g2, %g1, %g1
400002a0:   c2 27 bf f4       st  %g1, [ %fp + -12 ]
400002a4:   82 10 20 00       clr  %g1
400002a8:   b0 10 00 01       mov  %g1, %i0
400002ac:   81 e8 00 00       restore
400002b0:   81 c3 e0 08       retl
400002b4:   01 00 00 00       nop
```

First value in this example is the memory address of each line. Second 32-bit section is the instructions. Last part is the assembly code of the C code that includes AND operation with two variables and output is stored another variable.

Compilation of C codes and disassembly of **elf** files of logical operations completed, each **elf** file is loaded to the board and register values are tracked with breakpoints on GRMON console. Figure 2.19and Figure 2.20 show example debugging screens after **elf** file is loaded.

```
grmon2> load and_with_value.exe
  40000000 .text                    6.1kB /   6.1kB   [===============>] 100%
  40001890 .rodata                   16B              [===============>] 100%
  400018A0 .data                    464B              [===============>] 100%
  Total size: 6.61kB (462.77kbit/s)
  Entry point 0x40000038
  Image /opt/bcc-2.0.2-gcc/src/examples/basic_operations/and_with_value.exe loaded

grmon2> bp 0x40000288 cpu0
  Software breakpoint 1 at 0x40000288

grmon2> bp 0x40000290 cpu0
  Software breakpoint 2 at 0x40000290

grmon2> bp 0x4000029c cpu0
  Software breakpoint 3 at 0x4000029C

grmon2> bp 0x400002a0 cpu0
  Software breakpoint 4 at 0x400002A0

grmon2> go

  CPU 0:  breakpoint 1 hit
          0x40000288: c227bffc  st  %g1, [%fp - 0x4]  <main+8>
  CPU 1:  Power down mode

grmon2> reg g1
       g1 = 6 (0x00000006)

grmon2> cont

  CPU 0:  breakpoint 2 hit
          0x40000290: c227bff8  st  %g1, [%fp - 0x8]  <main+16>
  CPU 1:  Power down mode

grmon2> reg g1
       g1 = 4 (0x00000004)
```

Figure 2.19 : GRMON console view of the debugging (continued)

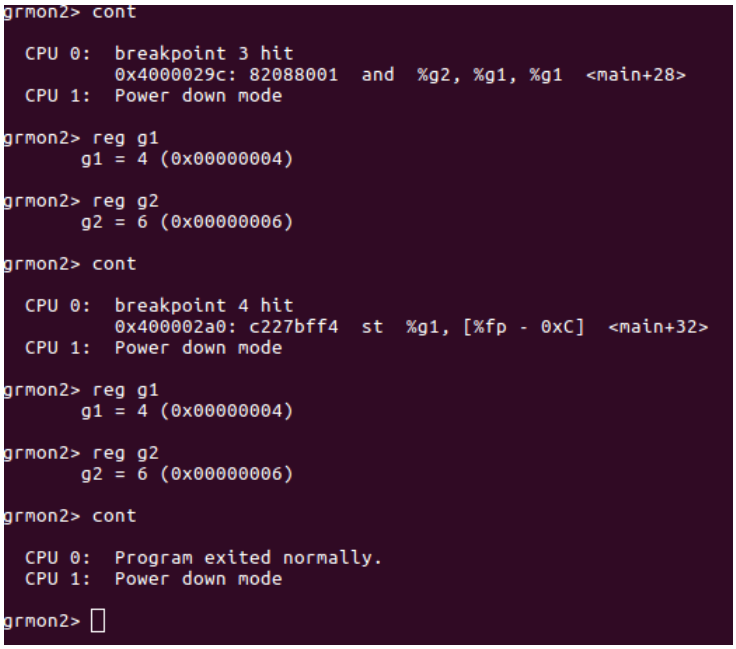


Figure 2.20 :GRMON console view of the debugging

2.7.6 Addition of new instructions

Integer Unit of the Leon3 and the sparcV8 instruction definitions analyzed for the addition of new instructions. In Integer Unit, names are used to define logical operations and these named are assigned to the 3-bit values. These 3-bit values show that these operations are ALU operations. Firstly, definitions of these names are found in Integer Unit. Blank values that are not assigned to any operation are determined. Then, the definition of the logical operation instruction names is detected in SparcV8 instruction declaration file named “sparc.vhd”. These instruction declarations are connected to the definition of instructions in Sparc V8 manual document that downloaded from Cobham Gaisler website [26].

Instruction formats are defined in Figure 2.21. Logical operations have the value 10 for *op* field. The *op3* field defines which operation is executed.

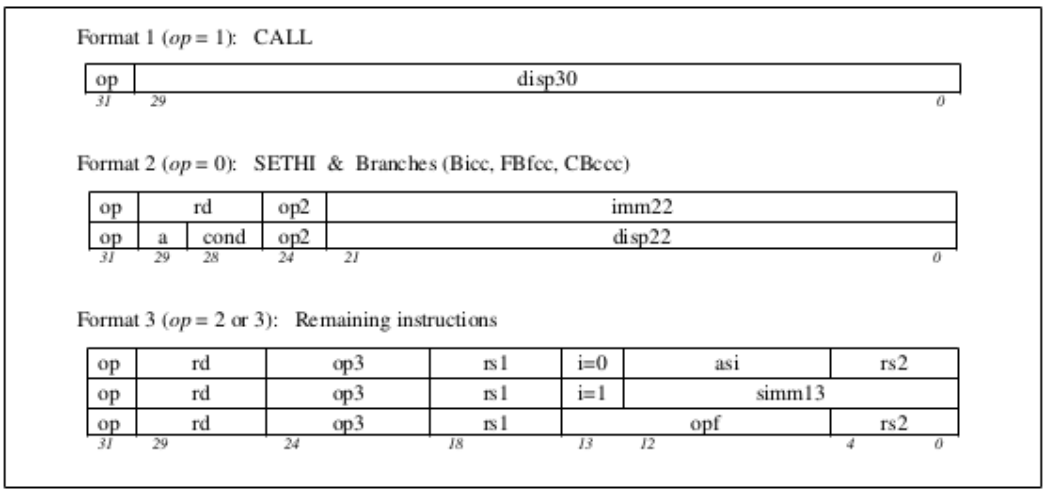


Figure 2.21 : SparcV8 ISA Instruction Formats [30]

Definition of instruction fields is given below:

op and *op2* : These 2- and 3-bit fields encode the 3 major formats and the format 2 instructions according toTable 2.6 and Table 2.7.

Table 2.6 : Different instruction formats according to the value of *op* [30]

Format	*op*	Instructions
1	1	CALL
2	0	Bicc, FBfcc, CBccc, SETHI
3	3	memory instructions
3	2	arithmetic, logical, shift, and remaining

Table 2.7 : Different instruction according to the value of *op2* [30]

op2	Instructions
0	UNIMP
1	unimplemented
2	Bicc
3	unimplemented
4	SETHI
5	unimplemented
6	FBfcc
7	CBccc

rd : This 5-bit field is the address of the destination (or source) r or f or coprocessor register(s) for a load/arithmetic (or store) instruction. For an instruction that read/writes a double (or quad), the least significant one (or two) bits are unused and should be supplied as zero by software.

a : The *a* bit in a branch instruction annuls the execution of the following instruction if the branch is conditional and untaken or if it is unconditional and taken.

cond : This 4-bit field selects the condition code(s) to test for a branch instruction.

imm22 : This 22-bit field is a constant that SETHI places in the upper end of a destination register.

disp22 and *disp30 :* These 30-bit and 22-bit fields are word-aligned, sign-extended, PC-relative displacements for a call or branch, respectively.

op3 : This 6-bit field (together with 1 bit from op) encodes the format 3 instructions.

i : The i bit selects the second ALU operand for (integer) arithmetic and load/store instructions. If
i = 0, the operand is r[rs2]. If i = 1, the operand is simm13, sign-extended from 13 to 32 bits.

asi : This 8-bit field is the address space identifier supplied by a load/store alternate instruction.

rs1 : This 5-bit field is the address of the first r or f or co-processor register(s) source operand. For an instruction that reads a double (or quad), the least significant bit (or 2 bits) are unused and should be supplied as zero by software.

rs2 : This 5-bit field is the address of the second r or f or co-processor register(s) source operand when i = 0. For an instruction that reads a double-length (or quad-length) register sequence, the least significant bit (or 2 bits) are unused and should be supplied as zero by software.

simm13 : This 13-bit field is a sign-extended 13-bit immediate value used as the second ALU operand for an (integer) arithmetic or load/store instruction when i = 1.

opf : This 9-bit field encodes a floating-point operate (FPop) instruction or a co-processor operate (CPop) instruction.

Definition of logical operations and their *op3* values and assembly language syntax are given in Table 2.8 and Table 2.9. Instruction format of logical operations is also given in Figure 2.22.

Table 2.8 : Definition and *op3* values of logical operations [30]

opcode	*op3*	*operation*
AND	000001	And
ANDcc	010001	And and modify icc
ANDN	000101	And Not
ANDNcc	010101	And Not and modify icc
OR	000010	Inclusive Or
ORcc	010010	Inclusive Or and modify icc
ORN	000110	Inclusive Or Not
ORNcc	010110	Inclusive Or Not and modify icc
XOR	000011	Exclusive Or
XORcc	010011	Exclusive Or and modify icc
XNOR	000111	Exclusive Nor
XNORcc	010111	Exclusive Nor and modify icc

Table 2.9 : Assembly language syntax of logical operation [30]

`and`	reg_{rs1} , *reg_or_imm* , reg_{rd}
`andcc`	reg_{rs1} , *reg_or_imm* , reg_{rd}
`andn`	reg_{rs1} , *reg_or_imm* , reg_{rd}
`andncc`	reg_{rs1} , *reg_or_imm* , reg_{rd}
`or`	reg_{rs1} , *reg_or_imm* , reg_{rd}
`orcc`	reg_{rs1} , *reg_or_imm* , reg_{rd}
`orn`	reg_{rs1} , *reg_or_imm* , reg_{rd}
`orncc`	reg_{rs1} , *reg_or_imm* , reg_{rd}
`xor`	reg_{rs1} , *reg_or_imm* , reg_{rd}
`xorcc`	reg_{rs1} , *reg_or_imm* , reg_{rd}
`xnor`	reg_{rs1} , *reg_or_imm* , reg_{rd}
`xnorcc`	reg_{rs1} , *reg_or_imm* , reg_{rd}

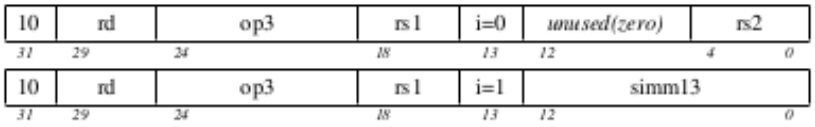


Figure 2.22 : Instruction format of logical operations [30]

New operations should have different *op3* values and different instruction names that are used in integer unit. Because using of the same value to different instructions may create chaos and overwrite. Blanck spaces in 3-bit values for logical ALU operation are determined. Only one value is not assigned any logical ALU operation but AES block cipher requires 3 new instructions to perform encryption. Thus, 3-bit value

definitions for the logical ALU operations are extended to 4-bit. This change provides additional 8 spaces for new operations.

After new operations are defined and hardware implemented with simple calculations like "*not aluin1 and not aluin2*", Instruction definitions should also be added to SparcV8 instruction definition file. o*p3* value has 6-bit and it is used for all operations except *call, jump,*and *branch* instructions. Thus, free 6-bit values are searched. It can be seen that 8 free space are available for the usage of new instructions. Three of them is used for new instructions. As a result, *op3* values of new instructions are determined as follows;

op3 of *new_ins_1* = 001101, *op3* of *new_ins_2* = 011001, *op3* of *new_ins_3* = 011101.

Based on the AND operation and its instruction format, new instructions are defined as follows;

new_ins_1 = 82 68 08 01, *new_ins_2* = 82 C8 8o 01, *new_ins_3* = 82 E8 80 01.

The next step is the generation of **elf** file that contains new instructions. For this aim, previously tested AND operation and its C code is used. In this code AND operation is used instead of the new operations. Then, this code is compiled with BCC2 same as before. After compilation, currently generated **elf** file is opened with one of the hexadecimal editors.

"Hexedit" is used to edit **elf** file in this study. After **elf** file opened with "hexedit", the main part of the application is found with the help of the disassembled "asm" file on currently generated **elf** file. The instruction of AND operation changed with the instruction of the new instruction and saved. Changed **elf** file is loaded to the processor with the help of the GRMON. A new application that contains new operation can be debugged with the breakpoints as described before.

2.7.7 Implementation of AES

AES block cipher is implemented with the new instructions; SubByte, ShiftRow ,and MixColumn. It also requires basic operations that are implemented in ALU. These

basic operations are XOR, left shift, addition, and AND. Firstly, new operations are constructed as a individual hardware with VHDL in Xilinx ISE. Each of the new operations is defined as a submodule and they implement their functions within these submodules. For example, SubByte operation uses defined 16x16 table that includes values changed with an input value of the plain matrix cell. SubByte operation takes one input as a row of the plain matrix. ShiftRow operation takes two input. One of them is the row of the plain matrix, and the other is the number of the shift operation. Shift operation basically implemented with concatenation in VHDL. MixColumn operation also requires one input that is a column of the plain matrix. Thus, before the MixColumn operation, columns of the plain matrix should be generated. MixColumn operation requires calculations and generates a result. After MixColumn operations, regeneration of the rows from columns of the plain matrix should be implemented.

All of these operations are implemented and required row-to-column and column-to-row conversations are made. Hardware implementation of the first round of AES block cipher is tested and desired results observed. The next step is the addition of this hardware of operations to the relevant location into Integer Unit of the Leon3.

VHDL implementation of the SubByte, ShiftRow, and MixColumn operations are placed to previously defined new instructions which are implemented as simple operations. SubByte operation changed with *new_ins_1*, ShiftRow is changed with *new_ins_2*, and MixColumn is changed with *new_ins_3*. Only ShiftRow requires the second operand, thus the dummy values are assigned to second operands of the SubByte and MixColumn operation in C code of AES block cipher.

After these operations are added to Integer Unit of Leon3, processor is re-synthesized and Atlys board reconfigured. C application that implements first round of the AES block cipher is written. In this code, input data is an array with length 4 that includes four rows of the data should be encrypted. Input key also defined as an array with length 4 that includes four rows of the key. Plain matrix is constructed from input data and the key matrix is constructed from input key. Defined operations and row-to-column, column-to-row operations are implemented on both plain matrix and key matrix. As a result of these operations, result data of first round is obtained.

BCC2 cross-compiler does not know new instructions and its input number. Thus, one default operation is used instead of the new operations. After BCC2 generates **elf** file, these default operations are changed with the new instructions via "hexedit" tool defined before. AND operation is chosen as a default operation because it has two operands and its instruction is easily noticeable. If AND operation is used instead of SubByte or MixColumn operations, second operand set as dummy value "9" for SubByte and "8" for MixColumn. These values are assigned to different variables before they are used.

After **elf** file is generated and required changes are done on **elf** file with "hexedit", **elf** file is loaded to the board and it is verified. Application is debugged with the breakpoints. It is seen that expected values are calculated according to current register values. Control of output data is done with if statements and LEDs on board. If all of four row of output data is equal to rows of desired output, 0x0000000f value is passed to the output register of GPIO port. Then first four LEDs are on. If one of the rows is truly calculated, 0x00000001 is passed to the output register of GPIO port. Equality of two or three rows are visualized by the assign 0x00000003 and 0x00000007 to the output register of GPIO port respectively.

Whole AES block cipher includes 10 round. At this stage, AES algorithm is not implemented as 10 round, since default on-chip RAM size is not enough to fit application that implements 10 round of AES algorithm. Function calls in C application may also cause problems with the pointer declarations because 128-bit data should be passed to function via array implementation. These problems are expected to solve after the interim report.

2.8 Project Work Plan and Current Position

Table 2.10 : Timetable of this study (continued)

#	Task	Responsible Group Member(s)	WEEKS													
			1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	**Hardware implementation**	**ALL**	X	X	X	X										
2	Executing C/C++ codes	ALL					X	X	X	X						

3	Changing the Arithmetic Logic Unit	ALL									X	X	X	X	X	X
4	Using the Instruction in C/C++ code	ALL														
5	Comparison with soft and hard implementation of cryptography algorithms	ALL														
6	Comparison of all given Results	ALL														

Table 2.11 : Timetable of this study

#	Task	Responsible Group Member(s)	WEEKS													
			15	16	17	18	19	20	21	22	23	24	25	26	27	28
1	**Hardware implementation**	**ALL**														
2	Executing C/C++ codes	ALL														
3	Changing the Arithmetic Logic Unit	ALL	X	X												
4	Using the Instruction in C/C++ code	ALL			X	X	X	X	X	X	X	X				
5	Comparison with soft and hard implementation of cryptography algorithms	ALL											X	X		
6	Comparison of all given results	ALL													X	X

According to Table 2.10 and Table 2.11, hardware implementation of the Leon3 is completed. This process continued more time as expected. Because different Linux distributions cause installation and usage problems. Moreover, first board chooses Spartan3E has not enough basic logical blocks. Thus, board change to Atlys which has Spartan6.

Executing C/C++ codes on Leon3 processor is completed. BCC2 is installed and some basic test codes run on implemented Leon3 processor.

Changing the ALU is implemented and completed. This process requires analysis of the SparcV8 ISA especially instruction format of it and blank spaces for new

operations. This phase also requires implementation of AES block cipher at hardware side and at Integer Unit of the Leon3.

Using new instructions in C/C++ code is completed. New instructions that are used for encryption are determined and C/C++ code that implements encryption with AES block cipher is written. New instructions are implemented as dummy AND operation that is used instead of new instructions in C code. After compilation, dummy and operations are changed to new operations on **elf** file. Then, AES algorithm works properly and encrypted data is obtained as expected.

Last two steps of the timetable are shown in Table 11 and Table 12, comparison with soft and hard implementation of cryptography algorithms and comparison of all given results has not been completed yet.

ORPSoC-v2

Introduction

3.

3.1 ORPSoC-v2 is a system on chip (SoC) that found based on OR1200 from OpenRISC 1000 processor family, and it is supported by various FPGA development board. ORPSoC-v2 is written in VerilogHDL, it includes the codes of CPU and peripheral units besides the some auxiliary softwares. Users can download these files and change the CPU and its peripherals by increasing or decreasing the number of units on it depending on the user's desires and needs. These codes, then, can be synthesized by softwares provided by Xilinx or Altera companies and then be configured for target FPGA development board.

PRESENT

3.2

Despite the establishment of AES decreased the need for new block ciphers, PRESENT algorithm had been developed, because AES is not suitable for implementations on limited source and power consumption such as IOT applications, In 2007[31].

3.2.1 Algorithm

PRESENT algorithm includes 64 bit block data lenght and two different key length options(80-bit, 128-bit). Key is recalculated with defined operations. These recalculated keys are used to generate encrypted data. PRESENT is also part of the symmetric key algorithms as AES.

In this study, 80-bit key version of PRESENT algorithm is used in order to decrease implementiation area. The process to encyrpt one 64-bit data block completed in 31 rounds. Each round is identical and composed of three successive different layers: addRoundKey layer, sBoxLayer and pLayer. The difference in each round comes from the key, which is updated at last of each round.

Block diagram of the PRESENT algorithm is shown in Figure 3.1[32].

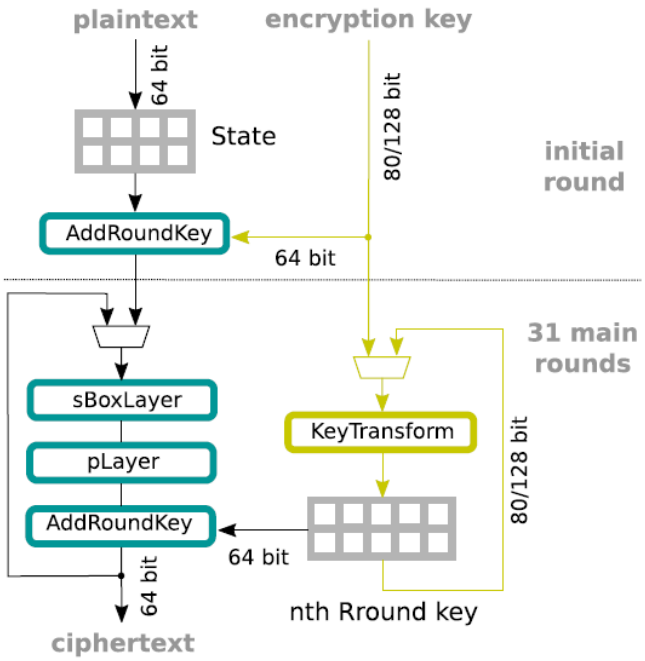


Figure 3.1 : Encryption procedure of PRESENT[32]

3.2.2 Operations of PRESENT Algorithm

PRESENT algorithm basically consist of 4 operation: AddRoundKey, sBoxLayer, pLayer and KeyTransform.

3.2.2.1 AddRoundKey

Adds the STATE to a 64-bit word from the roundkey using finite field arithmetic.

3.2.2.2 sBoxLayer

This layer is consist of 16 copies of a 4-bit to 4-bit S-Box, S0-S15. The current state is divided into sixteen 4-bit words fed into S-Boxes[33]. The content of the used S-Box in PRESENT is shown in Table 3.1.

Table 3.1 : sBox for PRESENT block cipher[31]

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

3.2.2.3 pLayer

pLayer is performing permutations on the bits of STATE. This layer changes the place of the bits in the STATE. The bit permutation used in PRESENT is given by the Table 3.2. Bit i of state is moved to bit position P(i).

Table 3.2 : pLayer pattern for PRESENT block cipher[31]

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(i)$	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(i)$	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P(i)$	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P(i)$	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

3.2.2.4 KeyTransform

Round keys are different from each other. For the key generation, previous round key is used. After the leftmost 64-bit of 80-bit key is extracted for AddRoundKey operation, the key register is rotated to left by 61 bit positions then sBox layer is applied to the left-most four-bit and then $k_{19}k_{18}k_{17}k_{16}k_{15}$ bits are put in exclusive-or with five-bit round counter value.

1. $[k_{79}k_{78}\ldots k_1k_0] = [k_{18}k_{17}\ldots k_{20}k_{19}]$
2. $[k_{79}k_{78}k_{77}k_{76}] = S[k_{79}k_{78}k_{77}k_{76}]$
3. $[k_{19}k_{18}k_{17}k_{16}k_{15}] = [k_{19}k_{18}k_{17}k_{16}k_{15}] \oplus \texttt{round\_counter}$

Figure 3.2 : General description for the key generation of PRESENT[31]

3.2.3 Hardware Implementation of PRESENT Algorithm

One can find the requested test-vectors in the original paper. The paper includes test vectors for all four cases one request (one or both of key and plaintext are 00...0000...00 or FF...FFFF...FF). Table 3.3 shows the test vectors.

Table 3.3 : Test Vectors for PRESENT block cipher[31]

Plaintext	Key	Ciphertext
00000000 00000000	00000000 00000000 0000	5579C138 7B228445
00000000 00000000	FFFFFFFF FFFFFFFF FFFF	E72C46C0 F5945049
FFFFFFFF FFFFFFFF	00000000 00000000 0000	A112FFC7 2F68417B
FFFFFFFF FFFFFFFF	FFFFFFFF FFFFFFFF FFFF	3333DCD3 213210D2

Pure hardware implementation of PRESENT is performed and tested. The simulation result is shown in Figure 3.3.

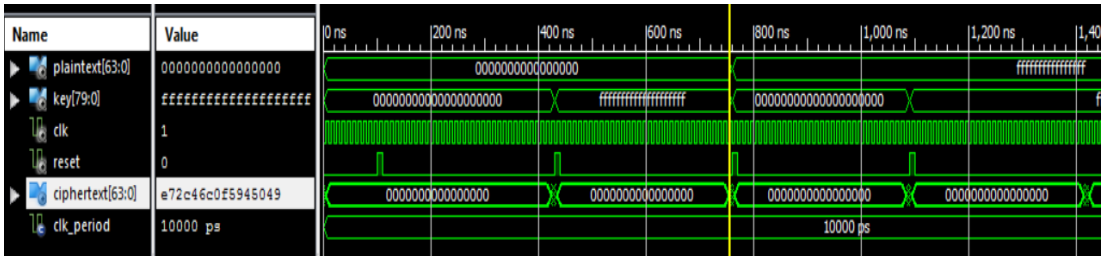


Figure 3.3 : Isim simulation result for pure hardware PRESENT algorithm

As it is seen on Figure 3.3,the ciphertext value is matched with the second row of the table x, given plaintext and key.

3.3 OpenRISC 1000 Instruction Set Architecture

OpenRISC is the project that conducted with the participation of voluntary people in order to develop open source ISA based on RISC (Reduced Instruction Set Computing) architecture.

In this context, the written codes are completely open and everyone who wishes have the right to use and modify designs. It is possible to show the freedoms that the offered license has basicially offers;

- Right to use codes unlimited
- Right to access and review source code
- Right to freely distribute source code
- Right to change source code

In this architecture, it is aimed to produce 32-bit and 64-bit instructions that can be used in a wide range such as embedded systems, automotive, portable computers etc. As it is known, processors are designed by creating their instruction sets. The 32 and

64 bit supported instruction sets included in the OpenRISC 1000 instruction set is shown in Figure 3.4.

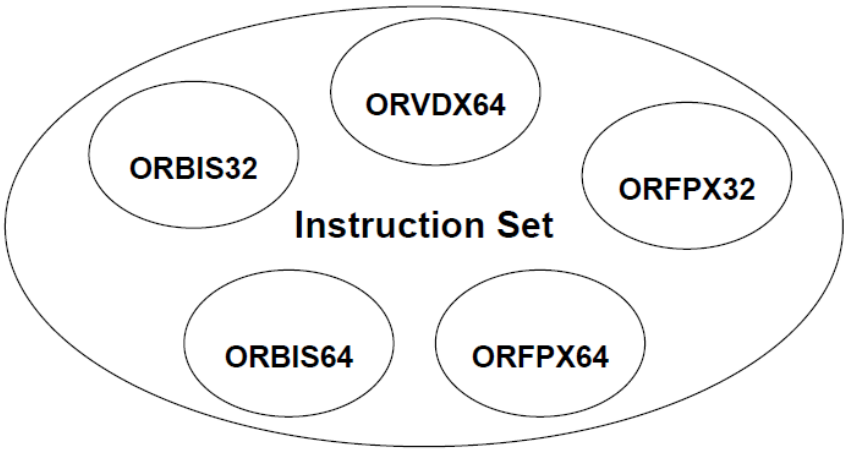


Figure 3.4 : Instruction Set of OpenRISC 1000 Family[3]

As shown in Figure 3.4 there are multiple sets of instructions. The OpenRISC Basic Instruction Set (ORBIS32) is the compulsory for all kinds of implementations and it consist of basic instructions such as addition, subtraction, multiplication, shift, jump etc.The 64-bit version of this set is ORBIS64. Besides, the OpenRISC Vector and Signal processing instruction set (ORVDX64 - OpenRISC Vector / DSP Extension) use 32-bit length instructions and process 64-bit long data..Likewise, the Floating Point Numbers Instruction Set (ORFPX32 - OpenRISC Floating Point Extension) is designed to handle floating-point numbers. In addition to all these, the custom instructions are also designed, so that user can add new instructions that can be useful for spesific designs.

l.add **Add Signed** **l.add**

31 26	25 21	20 16	15 11	10	9 8	7 . . . 4	3 . . . 0
opcode 0x38	D	A	B	reserved	opcode 0x0	reserved	opcode 0x0
6 bits	5 bits	5 bits	5 bits	1 bits	2 bits	4 bits	4bits

Figure 3.5 : Sample instruction model for OpenRISC 1000[3]

Format:

l.add rD,rA,rB

Description:

The contents of general-purpose register rA are added to the contents of general-purpose register rB to form the result. The result is placed into general-purpose register rD.

32-bit Implementation:

rD[31:0] < - rA[31:0] + rB[31:0]
SR[CY] < - carry
SR[OV] < - overflow

64-bit Implementation:

rD[63:0] < - rA[63:0] + rB[63:0]
SR[CY] < - carry
SR[OV] < - overflow

Exceptions:

Range Exception
Figure 3.5 shows a basic addition instruction struct on OpenRISC ISA, likewise one of the eight reserved custom instruction's struct is shown in Figure X.

l.cust5 **Reserved for ORBIS32/64 Custom Instructions** **l.cust5**

Figure 3.6 : Reserved custom instruction model for OpenRISC 1000 [3]

Format:

l.cust5

Description:

This fake instruction only allocates instruction set space for custom instructions. Custom instructions are those that are not defined by the architecture but rather by the implementation itself.

32-bit Implementation:

N/A

64-bit Implementation:

N/A

Exceptions:

N/A

3.3.1 OR1200

First implementation of Openrisc 1000 is performed by Damjen Lampret in 2000. This implementation is called OR1200 and it is composed of just processor core. OpenRISC

1000 is the name for ISA. Hence, it does not contain hardware codes for implementation. It is possible to sort the general properties of the OR1200 processor;

- All features of the processor can be changed by the user.
- Ability to perform high performance operations
- High speed data memory and memory management
- Wishbone interface compatibility
- processor parameters can be easily changed by the user

Following these general features, when viewed the processor a bit closer, the functional properties can be sorted in the following order;

- CPU - DSP (Digital Signal Processing Unit)
- Instruction and data cache
- Power (energy) management unit and interface
- Timer
- Interrupt control unit and interface

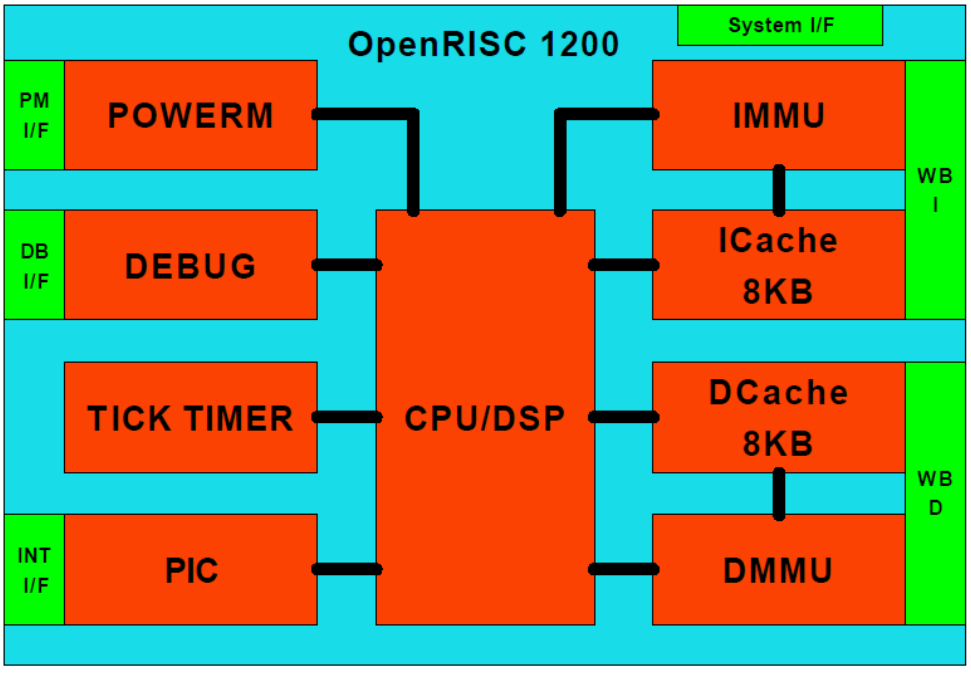


Figure 3.7 : OR1200 Core's architecture[3]

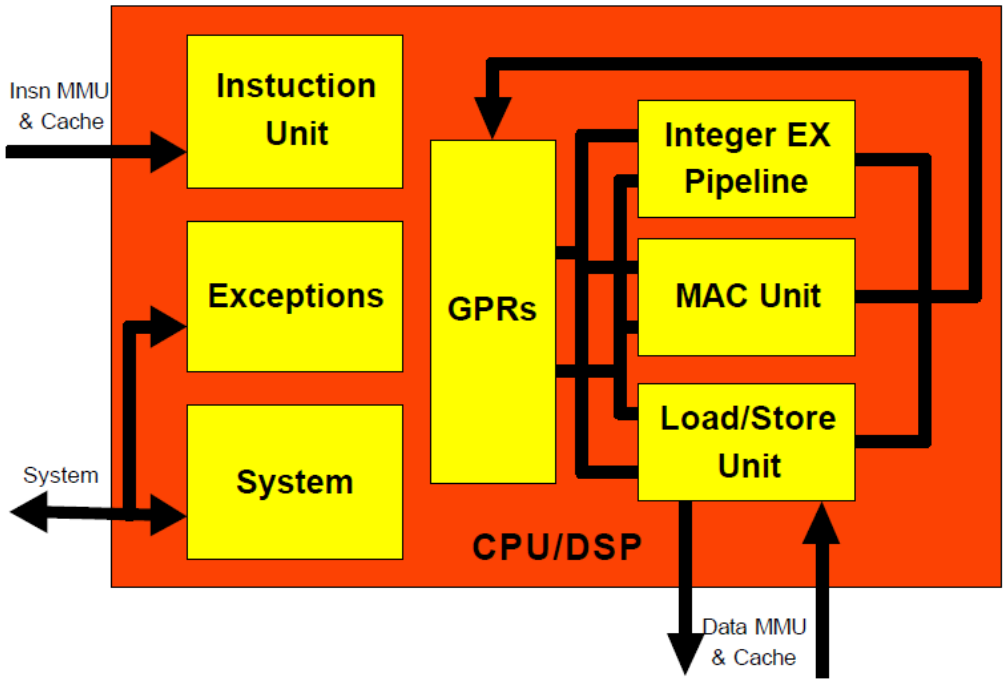


Figure 3.8 : OR1200 CPU-DSP Block Diagram[3]

In this study it is aimed to add each of PRESENT block cipher operation to custom instructions of OR1200 processor. Thus, required hardware addition of new instructions is reduced. It also reduces usage of the source of the FPGA on target board.

3.4 Tools for ORPSoC-v2

3.4.1 ISE Webpack Design Suite Installation

In this study, ATLYS board, whose features are given before, is used for implementation of OR1200 processor. ATLYS board is compatible with ISE Design Suite. Hence We used Xilinx ISE Design Suite. Webpack edition is free to use and can be downloaded from the following link: https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools/v2012_4---14_7.html.

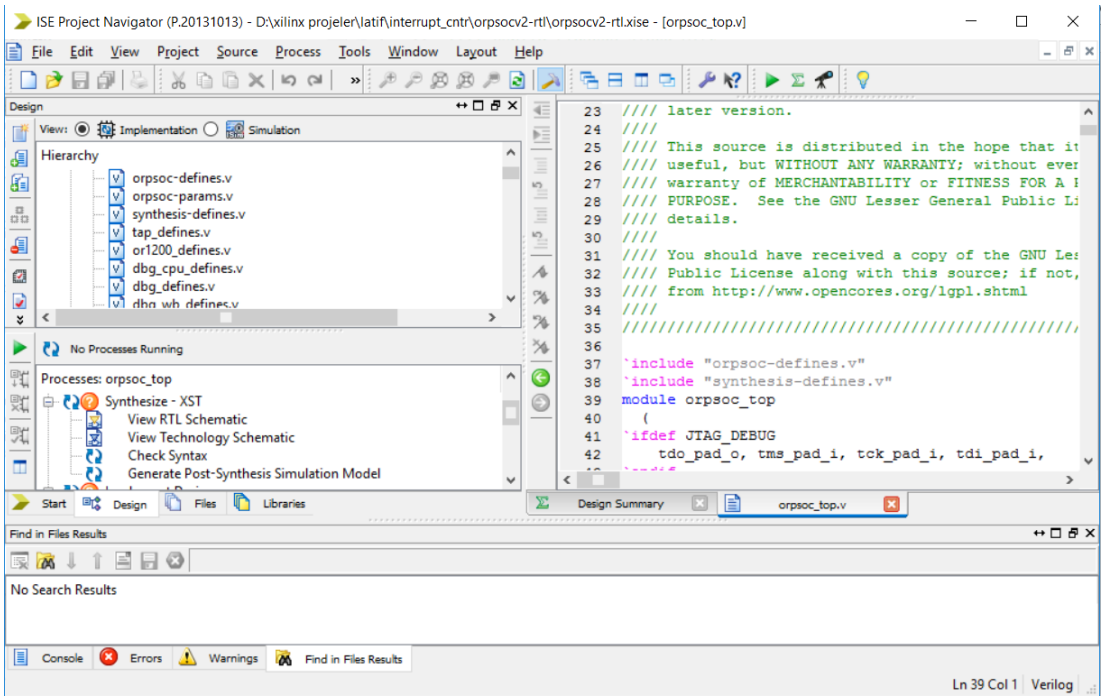


Figure 3.9 : ISE running on Windows10

ToolBox for OR1200 runs on Linux OS, hence virtual machine is needed for cross compiling codes on linux, while hardware implementations are being performed on ISE on Windows. Here is the link for free Oracle VM VirtualBox that is used in this study: https://www.virtualbox.org/wiki/Downloads

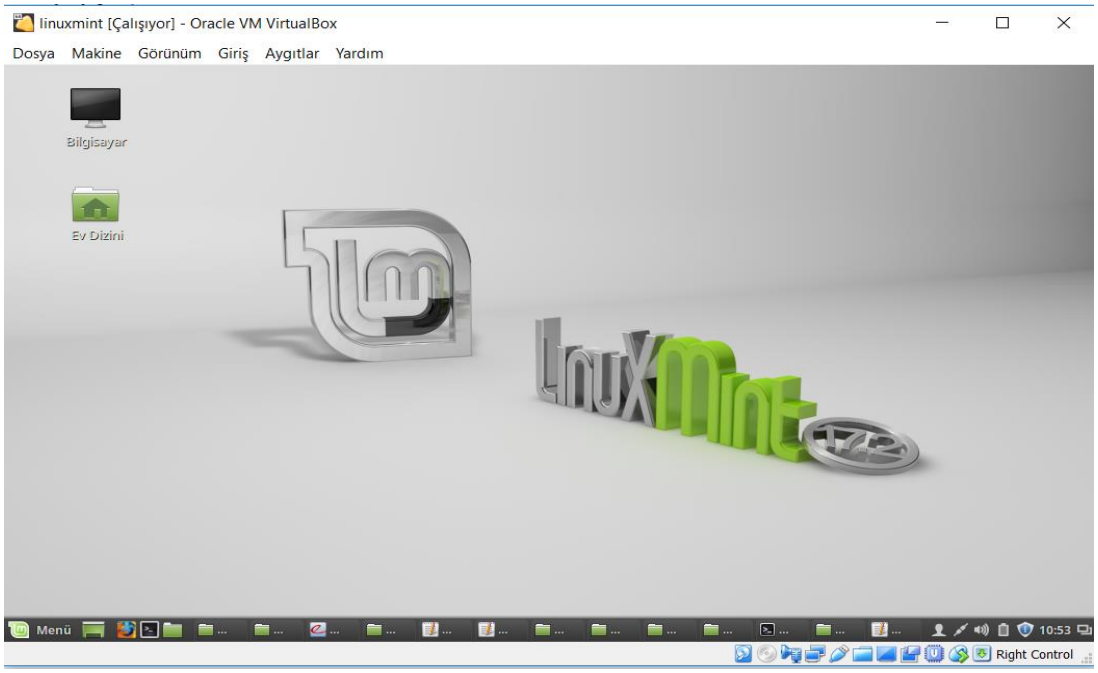


Figure 3.10 : LinuxMint 17.2 running on Oracle VM VirtualBox

3.4.2 ToolBox installation and usage for OR1200 processor

3.4.2.1 Cross compiler installation

Cross compiler is named different according to its compiled library;

- or1k-elf- : For applications that doesn't need operating system (OS) (bare metal)
- or1k-linux-uclibc : For applications that runs on Linux OS (uClibc library)
- or1k-linux-musl : For applications that runs on Linux OS (musl library)

In this study or1k-elf is used as Linux application is not needed for encryption app.

- Downloading and opening the code package of Binutils;

```
wget http://ftp.gnu.org/gnu/binutils/binutils -2.25.tar.bz2
tar xjvf binutils -2.25.tar.bz2
```

- Downloading Gcc code package;

```
git clone https://github.com/openrisc/or1k-gcc
```

- Downloading Newlib code package;

```
get ftp://sourceware.org/pub/newlib/
newlib -2.2.0.20150225.tar.gz
tar xzvf newlib -2.2.0.20150225.tar.gz
```

Before the installation, environmental variables should be defined and saved to the home directory. This definition can be done as below;

```
export PREFIX=/opt/or1k-elf
export PATH=\$PATH:\$PREFIX/bin
```

As it is seen, the variable named PATH shows the or1k-elf folder which is in root/opt directory.All compiler tools will be installed in this folder.

The read/write permissions must be given to the folder before the installation;

```
sudo mkdir \$PREFIX
sudo chown <user>:<user>\$PREFIX
```

After needed preperations for installation is done as described. The followings must be done for Binutils installation;

```
Mkdir build-binutils ; cd build-binutils
../binutils-2.25/configure --target=or1k-elf
--prefix=\$PREFIX –enable-shared \
--disable -itcl --disable –tk –disable-tcl
--disable -winsup --disable -gdbtk \
--disable –libgui --disable -rda --disable -sid
--disable -sim –with-sysroot
make
make install
cd ..
```

For Newlib installation followings are to be done;

```
mkdir build-newlib; cd build-newlib
../newlib -2.2.0.20150225/configure --target=or1k-elf
--prefix=\$PREFIX
make
make install
cd ..
```

For gcc stage 2 installation followings are to be done;

```
mkdir build-gcc-stage2; cd build-gcc-stage2
../or1k-gcc/configure --target=or1k-elf --prefix=\$PREFIX
--enable-languages=c , c++ --disable-shared
--disable-libssp –with-newlib
make
```

make install

cd ..

When these steps of the setup are completed, the code compiler for OpenRISC processors and also one of the most important code compilers, GNU GCC is ready. It is now quite easy to compile C, C ++ and assembly code for OpenRISC.

3.5 Implementation of OR1200 Processor

ORPSoC-v2 project codes can be downloaded from the following link;

svn co http://opencores.org/ocsvn/openrisc/openrisc/trunk/orpsocv2

After downloading codes, project file can be opened on ISE 14.7.

3.5.1 Led Test Application

End of the installation of needed tools and softwares, the hardware of the implemented OR1200 processor can be tested with a simple application on and off LEDs on the Atlys board. First, the General Purpose Input / Output (GPIO) port declaration should be examined.

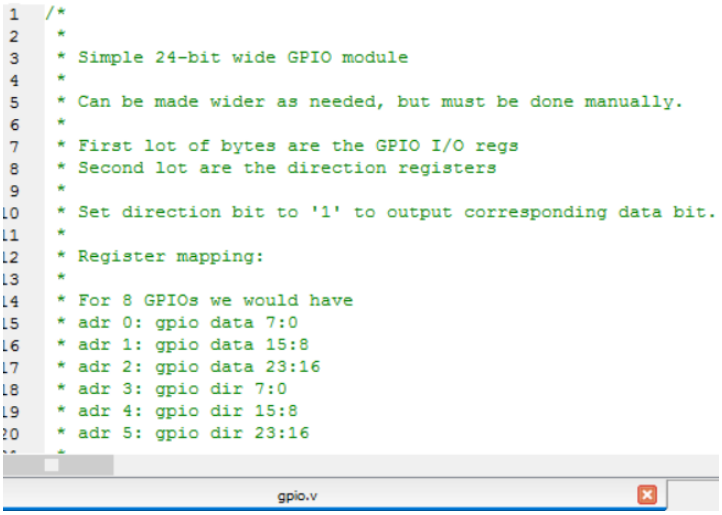


Figure 3.11 : GPIO module on orpsoc-v2

24-bit GPIO port is divided into parts such that, LEDs LD0 to LD7 are mapped to GPIO bits 0 to 7, switches SW0 to SW7 are mapped to GPIO bits 8 to 15, buttons BTNL, BTND, BTNR, BTNC BTNC are mapped to GPIO bits 16 to 20. In order to set LEDS as output, GPIO registers should be set appropriately.

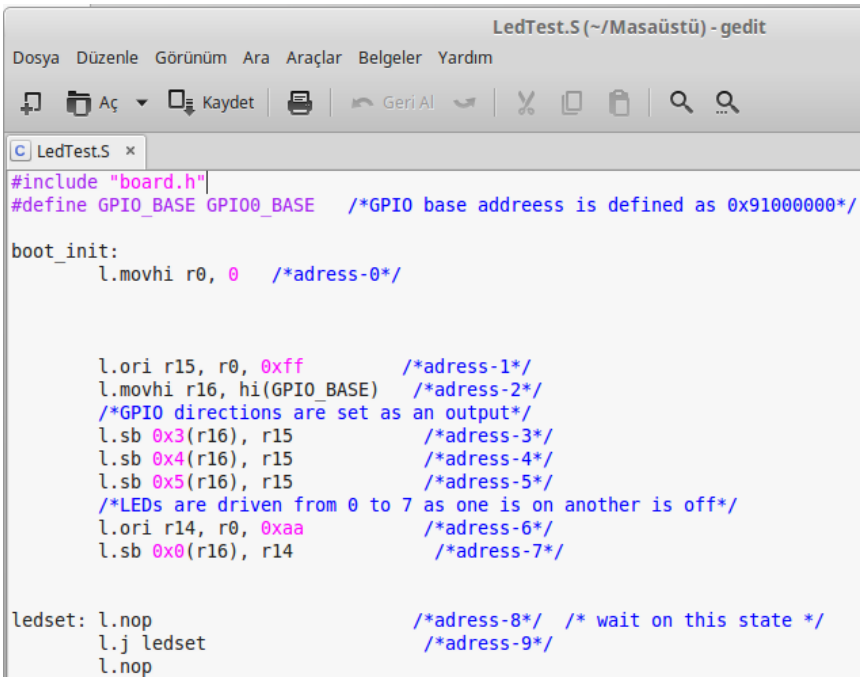


Figure 3.12 : Assembly code of Led application

The codes written in Assembly language in Figure 3.12 sets the direction as an output and drive the leds, then step into infinite loop. To get executable file, the following commands must be written on terminal:

or1k-elf-cpp -P LedTest.S LedTest.asm #C preprocessor include,define process and comment stuffs

or1k-elf-as -o LedTest.elf LedTest.asm # create elf formatted executable code from pure asm file

or1k-elf-objcopy -O binary LedTest.elf LedTest.bin # transform elf formatted file to flat binary

To store the application that will run on OR1200, in this study, we prefer to write machine codes on bootrom.v file in the orpsoc-v2 project. Hence the time that would be lost on spi-flash programming is saved. The following command produce machine codes from the executable file and write it on .v file

bin2vlogarray < LedTest.bin > bootrom.v

After the obtained bootrom.v file is added to the project. Design is implemented and leds are driven as in the Figure 3.13.

Figure 3.13 : Output of the LedTest application on OR1200 processor

LedTest application comprise of 10 machine codes, but the PRESENT encryption algorithm takes higher number of machine codes. OR1200's bootrom is limited by 64 machine codes by default. To be able to store encryption algorithm on bootrom the rom address width parameter on orpsoc-params.v is changed from 6 to 7. Orpsoc-params.v is the file that parameters of the SoC is defined in.

```
86  // ROM
87  parameter wbs_i_rom0_data_width = 32;
88  parameter wbs_i_rom0_addr_width = 7; //It was 6 before
89  parameter rom0_wb_adr = 4'hf;
```

Figure 3.14 : ROM Address width parameter of OR1200

Then, OR1200 boot address parameter is changed also from 32'hf0000100 to 32'hf0000200 in order to provide space for larger rom memory. Previously the rom0 module's local address was kept in between 7. and 2. bit of wbs_i_rom0_adr_i[31:0] signal. Hence there was no problem. After the manipulation of rom address width to 7. rom0 module's local address is kept in between 8. and 2. bit of the wbs_i_rom0_adr_i[31:0] signal. Previous value of boot address parameter result in starting from the 64. value of bootrom.v because the 8.bit's value was always '1'. By changing the boot address value to 32'hf0000200 that value on 8.bit is shifted to 9.bit and the problem is solved. As it is seen on wb_adr_i[8:2] signal on Figure X, up to 128 machine code is possible with the modification that is done.

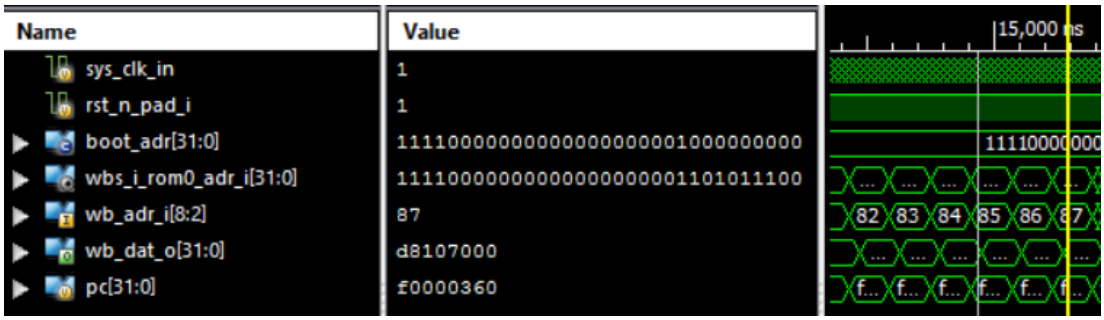


Figure 3.15 : ISIM simulation result of bootrom modification

```
1805  ////////////////////////////////////////////////////////////////////////////////
1806  // Boot Address Selection                                                     //
1807  //                                                                            //
1808  // Allows a definable boot address, potentially different to the usual reset //
1809  // vector to allow for power-on code to be run, if desired.                   //
1810  //                                                                            //
1811  // OR1200_BOOT_ADR should be the 32-bit address of the boot location          //
1812  // OR1200_BOOT_PCREG_DEFAULT should be ((OR1200_BOOT_ADR-4)>>2)               //
1813  //                                                                            //
1814  // For default reset behavior uncomment the settings under the "Boot 0x100"  //
1815  // comment below.                                                             //
1816  //                                                                            //
1817  ////////////////////////////////////////////////////////////////////////////////
1818  // Boot from 0xf0000100
1819  `define OR1200_BOOT_PCREG_DEFAULT 30'h3c00007f
1820  `define OR1200_BOOT_ADR 32'hf0000200
```

Figure 3.16 : Boot Address modification on OR1200

3.5.1 Custom Instruction Test Application

Fifth custom instruction of OR1200 performs move byte, set bit and clear bit operations for test applications. The following C code is written to test if l.cust5 instruction work.

/* Test three operations of custom instruction 5 */

#include "../support/support.h"

void buserr_except(){}

void dpf_except(){}

void ipf_except(){}

void lpint_except(){}

void align_except(){}

void illegal_except(){}

void hpint_except(){}

void dtlbmiss_except(){}

void itlbmiss_except(){}

```
void range_except(){}
void syscall_except(){}
void res1_except(){}
void trap_except(){}
void res2_except(){}

/* Custom instruction l.cust5: move byte

  Move byte custom instruction moves a least significant byte from source register rB and
  combines it with other bytes from source register rA and places combined result into rD. Location
  of the placed byte in rD depends on the immediate.
*/
#define MOVBYTE(dr,sr,sb,i) asm volatile ("l.cust5\t%0,%1,%2,%3,1" : "=r" (dr) : "r" (sr), "r" (sb), "i" (i)); report(dr)

/* Custom instruction l.cust5: set bit

  Take source register rA, set a specified bit to 1 and place result to destination register rD.
  Bit to be set is specified with an immediate.
*/
#define SETBIT(dr,sr,i) asm volatile ("l.cust5\t%0,%1,r0,%2,2" : "=r" (dr) : "r" (sr), "i" (i)); report(dr)

/* Custom instruction l.cust5: clear bit

  Take source register rA, clear a specified bit to 0 and place result to destination register rD.
  Bit to be cleared is specified with an immediate.
*/
#define CLRBIT(dr,sr,i) asm volatile ("l.cust5\t%0,%1,r0,%2,3" : "=r" (dr) : "r" (sr), "i" (i)); report(dr)
```

```
/* Test case for "move byte" custom instruction

   Move least significant byte from variable s into different byte positions of variable d.
   Every time a byte move is done compute checksum of variable d. Final checksum is used to verify
   correct operation.

*/
unsigned long test_movbyte()
{
        unsigned long s, d, r;

        s = 0x12345678;
        r = d = 0xaabbccdd;

        MOVBYTE    (d, d, s, 0);
        r += d;
        MOVBYTE    (d, d, s, 1);
        r += d;
        MOVBYTE    (d, d, s, 2);
        r += d;
        MOVBYTE    (d, d, s, 3);
        r += d;

        return (r);
}

/* Test case for "set bit" custom instruction

   Set a couple of bits of variable d to 1.
   Every time a bit is set compute checksum of variable d. Final checksum is used to verify
```

```
   correct operation.

*/
unsigned long test_setbit()
{
        unsigned long d, r;

        r = d = 0x00000000;

        SETBIT      (d, d, 10);
        r += d;
        SETBIT      (d, d, 15);
        r += d;
        SETBIT      (d, d, 19);
        r += d;
        SETBIT      (d, d, 25);
        r += d;

        return (r);
}

/* Test case for "clear bit" custom instruction

   Clear a couple of bits of variable d to 0.
   Every time a bit is cleared compute checksum of variable d. Final checksum is used
to verify
   correct operation.

*/
unsigned long test_clrbit()
{
        unsigned long d, r;

        r = d = 0xffffffff;
```

```
        CLRBIT      (d, d, 10);
        r += d;
        CLRBIT      (d, d, 15);
        r += d;
        CLRBIT      (d, d, 19);
        r += d;
        CLRBIT      (d, d, 25);
        r += d;

        return (r);
}

int main()
{
        unsigned long result = 0;

        result += test_movbyte();
        result += test_setbit();
        result += test_clrbit();

        printf("RESULT: %.8lx\n", result);
        report(result);
        exit(result);
}
```

3.6 l.cust5 custom instruction works properly on c code.

Project Work Plan and Current Position

According to Table 2.10 and Table 2.11, hardware implementation of the OR1200 is completed. Installation of the toolchain for OR1200 took time more than expected. Because the some packages toolchain use has changed by the time and some files needed to be updated.

Executing C/C++ codes on OR1200 processor is completed.

Arithmetic logic unit is not changed especially for PRESENT algorithm but custom instruction is tested using C/C++ code.

Last two steps of the timetable are shown in Table 2.10 and Table 2.11, comparison with soft and hard implementation of cryptography algorithms and comparison of all given results has not been completed yet.

ARM Cortex M0 DesignStart-r2p0

Introduction

4. ARM is one of the leading microprocessor designing company in the world. Eventhough the company doesn't manufacture silicon, company designs the most of
4.1 the microprocessors' architecture.

ARM microprocessors are divided into 3 types:

1-A Family: Applications processors for feature rich OS and 3rd party applications.

2-R Family: Embedded processors for real-time signal processing, control applications

3-M Family: Microcontroller-oriented processors for MCU, ASSP, and SoC applications

After the rise of the softcore microprocessors like OpenRisc, Leon, Risc and such, ARM decided to publish some of the M class microprocessors' hardware description codes in order to compete with the softcore microprocessors. Yet, to hold its codes confidential, company didn't share all of the codes openly; with only sharing the crypted versions of the hardware codes and obfuscated features, such as, not being able to debug and interrupt on some versions of the softcore microprocessor. Cortex M0 DesignStart is the product of this softcored version of the Cortex M0 microprocessor
4.2 [38].

v6-M Architecture

4.2.1 ARM architecture profiles

ARMv7 is documented as a set of architecture profiles, defined as follows:

ARMv7-A the application profile for systems supporting the ARM and Thumb instruction sets, and requiring virtual address support in the memory management model.

ARMv7-R the realtime profile for systems supporting the ARM and Thumb instruction sets, and requiring physical address only support in the memory management model.

ARMv7-M the microcontroller profile for systems supporting only the Thumb instruction set, and where overall size and deterministic operation for an implementation are more important than absolute performance.

While profiles were formally introduced with the ARMv7 development, the A-profile and R-profile have existed implicitly in earlier versions, associated with the Virtual Memory System Architecture (VMSA) and Protected Memory System Architecture (PMSA) respectively.
ARMv6-M is a subset of ARMv7-M, that provides:
• a lightweight version of the ARMv7-M programming model.
• the Debug Extension that includes architecture extensions for debug support.
• ARMv6 Thumb 16-bit instruction set compatibility at the application level.
• an optional Unprivileged/Privileged Extension.
• an optional PMSA Extension [34].

4.2.2 Instruction set architecture (ISA)

ARMv6-M implements the ARMv6-M Thumb instruction set, including a number of 32-bit instructions that use Thumb-2 technology. The ARMv6-M instruction set comprises:

• All of the 16-bit Thumb instructions from ARMv7-M, except CBZ, CBNZ, and IT.
4.3 • The 32-bit Thumb instructions, BL, DMB, DSB, ISB, MRS, and MSR [34].

Differences Between Cortex M0 and DesignStart

As mentioned before, there are slight differences between Cortex M0 processor and Cortex M0 processor of DesignStart. Mentioned differences and more can be seen on the Table 4.1.

Table 4.1 : Differences between DesignStart and Full processor versions [35]

Feature	Full Cortex-M0 processor	Cortex-M0 processor from DesignStart
Verilog code	Commented plain-text RTL	Flattened and obfuscated RTL
AMBA*3 AHB-Lite interface	Master and optional slave ports	Master port only
Armv6-M instruction set	Armv6-M instruction set support	Armv6-M instruction set support
Multiplier options	Fast single-cycle or small 32-cycle	Fast single cycle multiplier
Nested vectored interrupt controller (NVIC)	1-32 interrupt inputs	32 interrupt inputs only
Wake-up Interrupt Controller (WIC)	Optional	None
Architectural clock gating	Optional	None
24-bit system timer, SysTick	Optional reference clock	Reference clock supported
Hardware debugger interface	Optional Serial-Wire or JTAG	Serial-Wire only
Hardware debug support	Optional single step with up to four breakpoints, up to two watchpoints and PC sampling	Single step with four breakpoints, two watchpoints and PC sampling
Low-power signaling and domains	Optional state-retention power domains and power control signaling	SLEEPING, TXEV and RXEV signaling only

4.4 Nexys 4 Artix-7 FPGA Board

4.4.1 Introduction

The Nexys4 DDR board is a complete, ready-to-use digital circuit development platform based on the latest Artix-7™ Field Programmable Gate Array (FPGA) from Xilinx. With its large, high-capacity FPGA (Xilinx part number XC7A100T-1CSG324C), generous external memories, and collection of USB, Ethernet, and other ports, the Nexys4 DDR can host designs ranging from introductory combinational circuits to powerful embedded processors. Several built-in peripherals, including an accelerometer, temperature sensor, MEMs digital microphone, a speaker amplifier, and several I/O devices allow the Nexys4 DDR to be used for a wide range of designs without needing any other components.

Figure 4.1 : Nexys 4 Artix-7 FPGA Board [37]

4.4.2 Features

The Artix-7 FPGA is optimized for high performance logic, and offers more capacity, higher performance, and more resources than earlier designs. Artix-7 100T features include:

• 15,850 logic slices, each with four 6-input LUTs and 8 flip-flops
• 4,860 Kbits of fast block RAM
• Six clock management tiles, each with phase-locked loop (PLL)
• 240 DSP slices
• Internal clock speeds exceeding 450 MHz
• On-chip analog-to-digital converter (XADC)

The Nexys4 DDR also offers an improved collection of ports and peripherals, including:

• 16 user switches
• 16 user LEDs
• Two 4-digit 7-segment displays
• USB-UART Bridge

• 12-bit VGA output
• 3-axis accelerometer
• Two tri-color LEDs
• Micro SD card connector
• PWM audio output
• PDM microphone
• Temperature sensor
• 10/100 Ethernet PHY
• Serial Flash
• 128MiB DDR2
• Four Pmod ports
• Pmod for XADC signals
• Digilent USB-JTAG port for FPGA programming and communication
• USB HID Host for mice, keyboards and memory sticks [37]

4.5 Implementation

In DesignStart package, from debugging module to internal memories, every internal and external modules are included. In order to implement the microprocessor, only thing to do is, basicly including the verilog files into a Intel Quartus project and following the basic implementation steps. Unfortunately, DesignStart is not %100 Xilinx Vivado/ISE IDE compatible. So, this approach doesn't work for Xilinx FPGA boards. Yet, it clearly can be seen from DesignStart's manual, which modules should be included to the project, in order to implement the microprocessor. After including the files, we had to "debug" all of the errors by one by -which are occured because the system was not compatible for the Xilinx Vivado- before actually implementing the microprocessor [36].

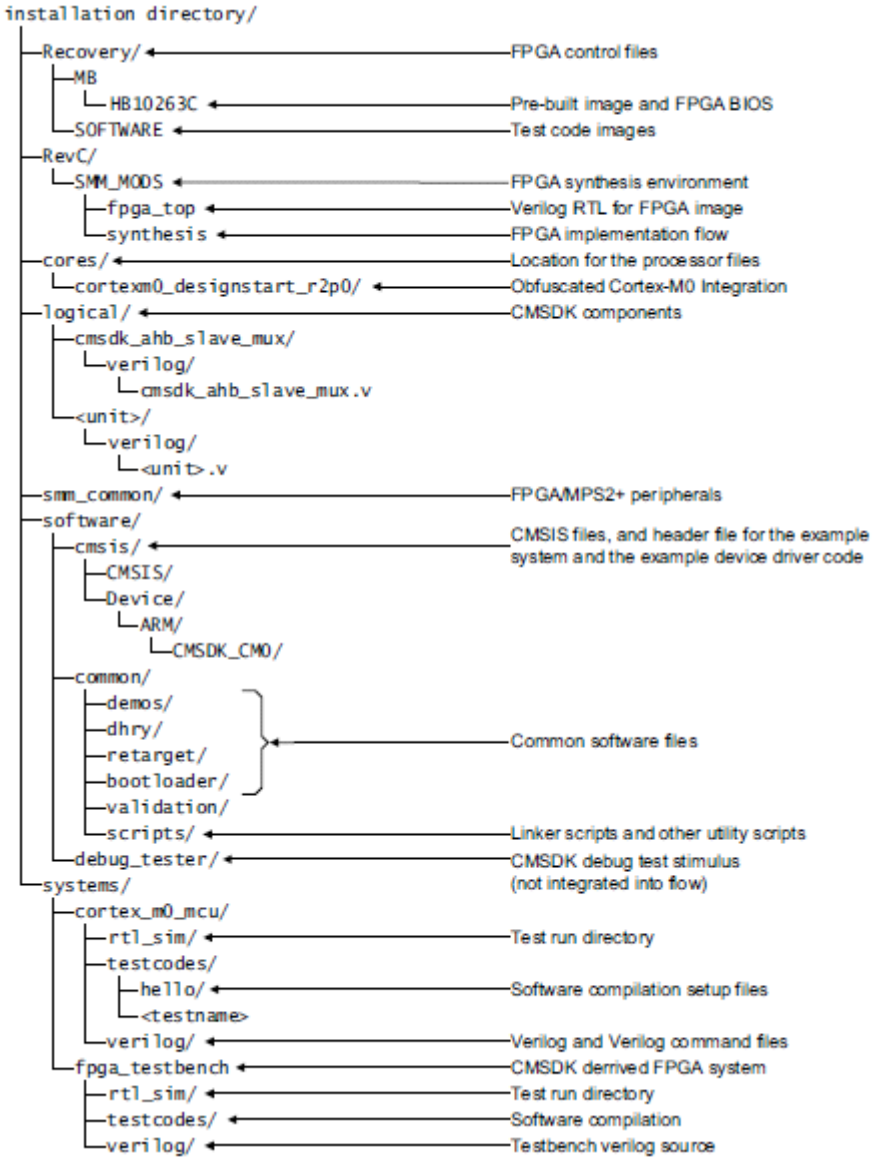


Figure 4.2 : Files included into DesignStart [35]

4.5.1 Simulation

DesignStart has a pre-made testbench module included, which enables the users to simulate the microprocessor working process way before the implemantation. This way, one can understand the faulty signals before actually implementing and embedding the microprocessor into the FPGA board.

This testbench module is positioned as the top module of the design, yet it has to be deleted before implementing and embedding the microprocessor onto the board, which makes cmsdk_mcu the top module. It can be seen the module diagram of the DesignStart on Figure 4.3.

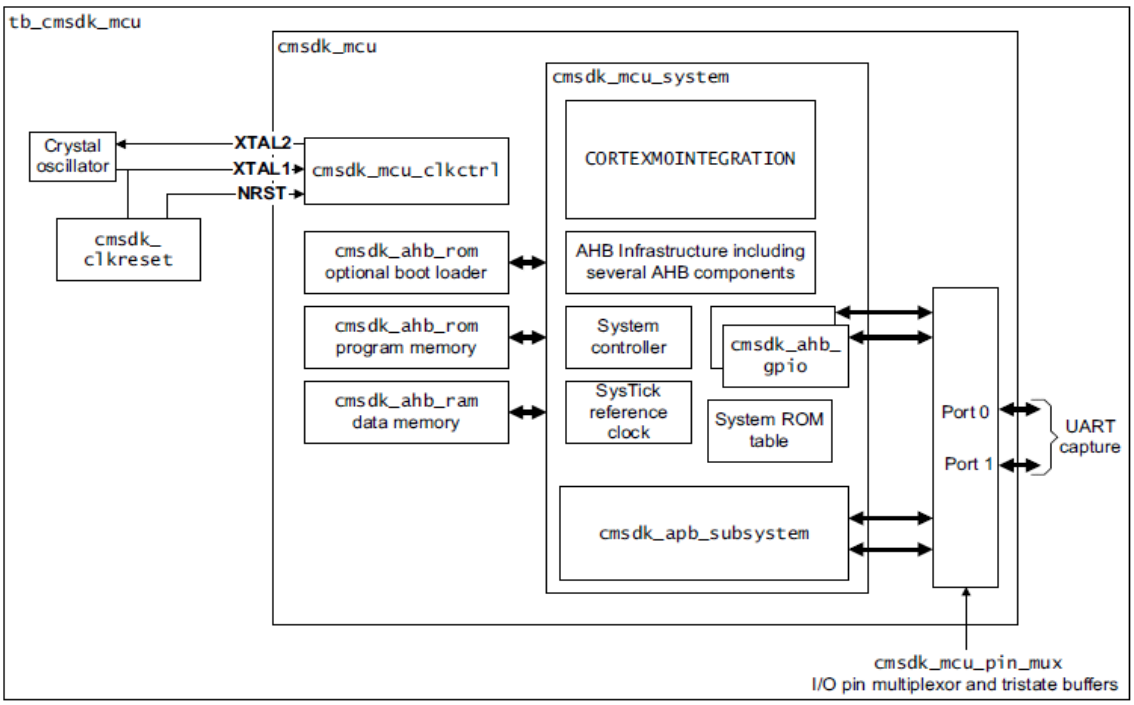
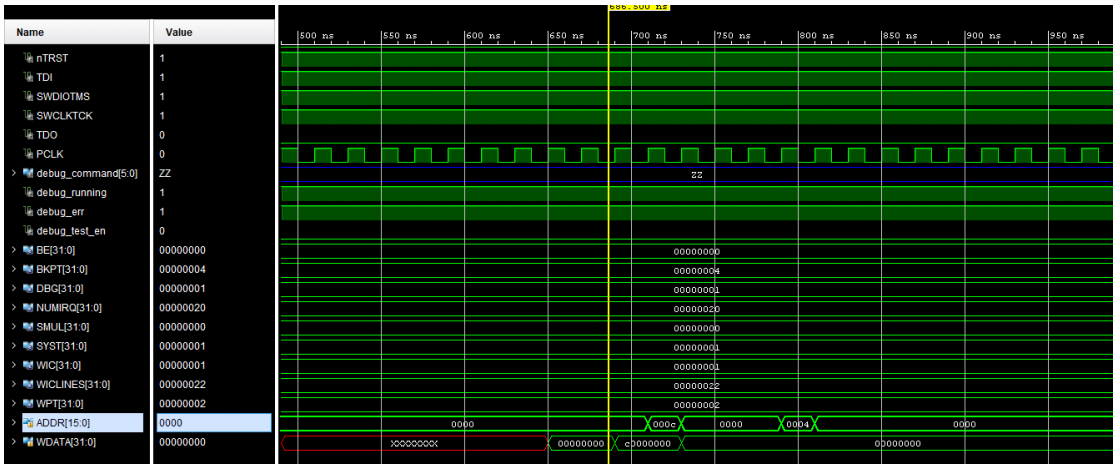


Figure 4.3 : Testbench and cmsdk_mcu modules [35]

On Figure 4.4, it can be seen that synthesized microprocessor design is working and responding to our inputs.



4.6

Figure 23 : Example simulation

Work Plan and Current Position

It's clearly can be seen that, we are behind our expectations on work plan. This happened due to the diffuculties to adjust DesignStart codes one by one in order to make it Xilinx FPGA compatible. At this point, we are almost done adjusting the module files and trying to understand if our processor is working correctly after our changes.

RI5CY

Introduction

5. RISC-V is a fresh, open-source, ISA designed for high performance and power efficiency. It is founded by UC Berkeley. In this project implementation of this ISA is
5.1 by ETH Zurich is used. ETH’s implementation is named Pulpino RI5CY core. It's backed from many powerful IC brands such as nvidia, Google, Microsemi[39].

Pulpino is a System on Chip(SoC) platform which consists multiple io peripherals such as Universal Asynchronous Receiver-Transmitter (UART), Serial Peripheral Interface. It also consist hardware and instruction level optimizations such as Hardware loops, post-increment adds, vector multiplication. The microprocessor in the system is RI5CY core which is promised to be highly efficient for ultra low energy. The algorithm that will be added is Advanced Encryption Standart (AES) which is explained in the section 2.6.

5.2 Pulpino Architecture

Pulpino has many usefull peripherals to communicate with outside world. It also has Advanced Extensible Interface (AXI) bus and Advanced Peripheral Bus for high speed and low speed peripherals respectively. Rest of the peripherals can be seen in the Figure 5.1.

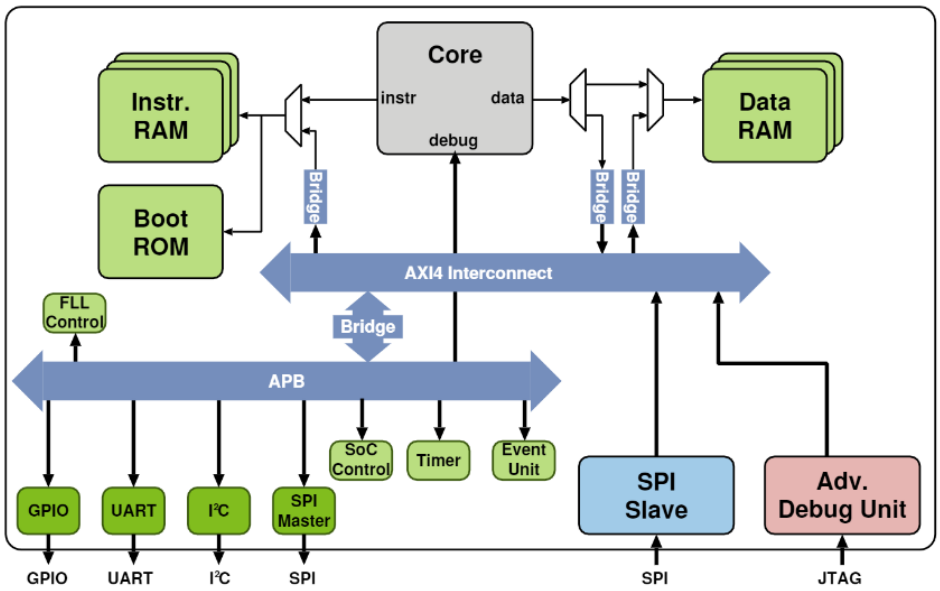


Figure 5.1 : Pulpino SoC[40]

The core has hardware loop optimizations and post-increment load and stores to further increase the performance[41]. It also has floating point support if wanted. A hardware loop is an extension in the core which controls the loops in hardware automatically which is more faster and energy efficient because it gets rid of redundant instruction's fetch cycle. In order to set up hardware loop, loop's end adress and loop count must be specified. RI5CY support 2 nested hardware loops. The loop instructions can be seen in the figure 5.2.

Mnemonic		Description
Long Hardware Loop Setup instructions		
lp.starti	**L, uimmL**	lpstart[L] = PC + (uimmL << 1)
lp.endi	**L, uimmL**	lpend[L] = PC + (uimmL << 1)
lp.count	**L, rs1**	lpcount[L] = rs1
lp.counti	**L, uimmL**	lpcount[L] = uimmL
Short Hardware Loop Setup Instructions		
lp.setup	**L, rs1, uimmL**	lpstart[L] = pc + 4 lpend[L] = pc + (uimmL << 1) lpcount[L] = rs1
lp.setupi	**L, uimmS, uimmL**	lpstart[L] = pc + 4 lpend[L] = pc + (uimmS << 1) lpcount[L] = uimmL

Figure 5.2 : Hardware loop setup instructions[41]

The core has fully independent pipeline, meaning that whenever possible data will propagate through the pipeline and therefor does not suffer from any unneeded stalls[41]. The pipeline is designed to be out-of-order compatible. Meaning that instructions that does not need Write-Back stages can be completed without going into Write-Back stage and create unwanted stalls. The data path of the pipeline can be seen in the figure 5.3.

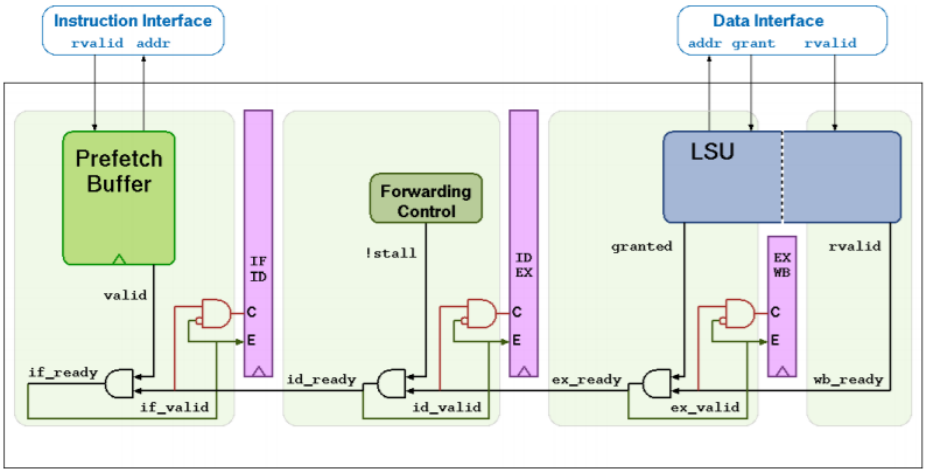


Figure 5.3 : RI5CY pipeline[41]

Cross Compiler

5.3 Cross compiler creates machine code other than the platform that it runs. For example, the code for ARM CPU can be created by using a x86 CPU. In this part the code for RISC-V ISA(Instruction Set Architecture) is created by using x86 machine.

5.3.1 General Flow

By using cross compiler, any C code can be run on the RI5CY core. GCC compiler creates an .elf file.

$ riscv32-unknown-elf-gcc –o program.elf program.c

This file(program.elf) is basically an assembly code of the C code and it is not the machine code yet. The assembly code can be examined by using objdump command.

$ riscv32-unknown-elf-objdump –d program.elf

By using objcopy function .bin file(the machine code file) can be created.

$ riscv32-unknown-elf-objcopy -O binary program.elf program.bin

The bin file can be examined by the xxd command.

$ xxd program.bin

5.3.2 Setup

Prequisities are given below:

Xilinx Vivado 2015.1
Cmake version >= 2.6
Gcc >= 5.2
Python >= 2.7

5.3.2.1 RI5CY GNU Toolchain

First, default gcc and g++ compilers must be installed in the system.

To install these to the system[43]:

$ sudo apt update

$ sudo apt upgrade

$ sudo apt install build-essential

To check the version of gcc

$ gcc --version

To check the location of the gcc compiler:

$ which gcc

Then the command below must be ran for RISC-V gcc compiler[44]:

$ sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev

After these steps, the ri5cy_gnu_toolchain can be downloaded from the pulp_platform github.

After downloading the files, "make" command must be typed in the terminal. It is recommended that the installation should be made in the /opt folder. Or the installation files can be copied into the /opt folder.

After the installation the path of the cross compilers must be included to the PATH permanently. To do this, "bashrc." in the home folder must be changed. The code in "bashrc." is always run when a new terminal window opens. “bashrc.” is located in the home folder. To see “bashrc.” user must enable “Show hidden files” by right-clicking in the home folder. Then “bashrc.” can be simply opened by double clicking on it.

To the last line of the "bashrc." the command below must be entered:
PATH=$PATH:/INSTALL_LOCATION/ri5cy_gnu_toolchain-master/install/bin

```
# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if ! shopt -oq posix; then
  if [ -f /usr/share/bash-completion/bash_completion ]; then
    . /usr/share/bash-completion/bash_completion
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi

if [ -x /usr/bin/mint-fortune ]; then
     /usr/bin/mint-fortune
fi
PATH=$PATH:/opt/ri5cy_gnu_toolchain-master/install/bin
PATH=$PATH:/opt/Xilinx/Vivado/2015.1/bin:/home/tolga/Xilinx/14.7/ISE_DS/ISE/bin/lin64
```

Figure 5.4 : Inside of “bashrc.”.

5.3.2.2 Cmake and Python2

Cmake and the python must be installed to proceed further. To install cmake any linux version greater than 2.6 can be downloaded from https://cmake.org/download/. After downloading the files, the commands below must be entered inside the download folder to the terminal:

$ ./bootstrap

$ make

$ make install

To install python commands below must be entered:

$ sudo apt-get update

$ sudo apt-get install build-essential checkinstall

$ sudo apt-get install libreadline-gplv2-dev libncursesw5-dev libssl-dev libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev

$ cd /usr/src

$ sudo wget https://www.python.org/ftp/python/2.7.14/Python-2.7.14.tgz

$ sudo tar xzf Python-2.7.14.tgz

$ cd Python-2.7.14

$ sudo ./configure

$ sudo make altinstall

Then the version can be checked by the command:

$ python2.7 -V

5.3.2.3 Pulpino Master file

After adding the cross compiler to the PATH, the project files can be downloaded from the pulp_platform/pulpino github. Then, a new folder named "build" must be created in the sw folder. Then, the script "cmake_configure.riscv.gcc.sh" must be run in the terminal.

The tool installation is now complete. At this moment, apps in the installation folder can be compiled using the make command in the build folder. The result will be in the build/apps[45].

Example: make boot_code.install will create a boot_code.sv in the "build/apps/boot_code/boot".

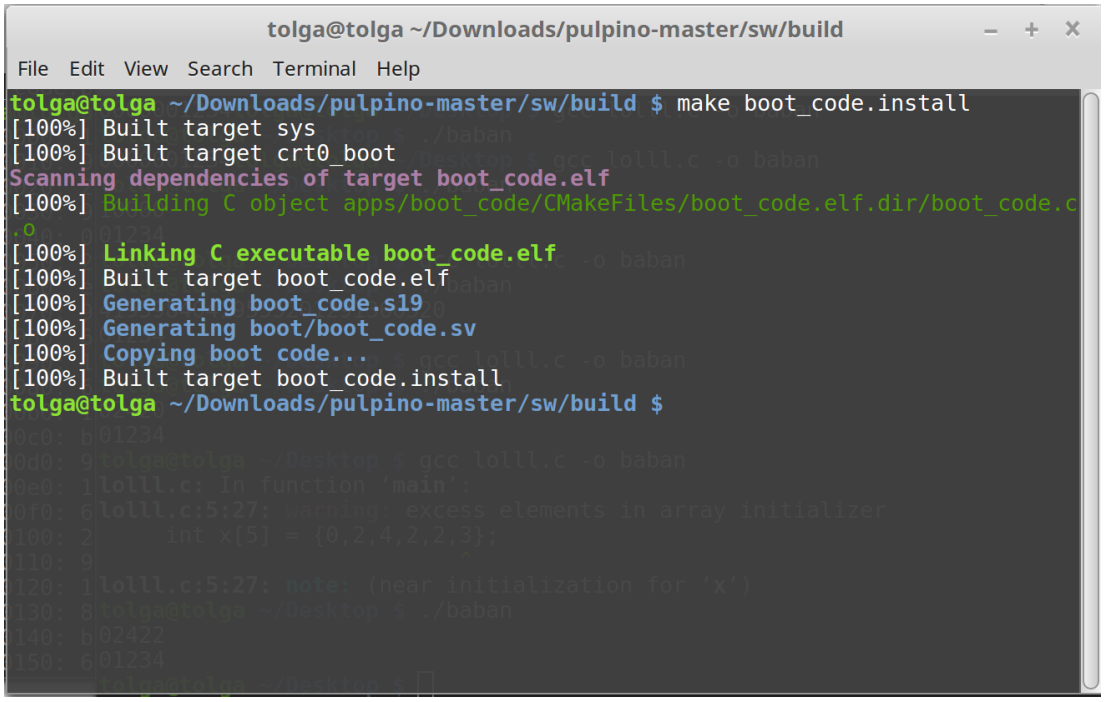


Figure 5.5 : Successful boot_code.sv compilation. This will create the machine code that will be ran when the cpu starts up. Contents of boot_code.sv must be copied into the design. Notice that this automatically creates "boot_code.elf".

```
Disassembly of section .text.startup.main:

00008150 <main>:
    8150:   1101                    addi    sp,sp,-32
    8152:   cc22                    sw  s0,24(sp)
    8154:   45d1                    li  a1,20
    8156:   4501                    li  a0,0
    8158:   1a101437            lui s0,0x1a101
    815c:   ca26                    sw  s1,20(sp)
    815e:   c84a                    sw  s2,16(sp)
    8160:   c64e                    sw  s3,12(sp)
    8162:   c452                    sw  s4,8(sp)
    8164:   c256                    sw  s5,4(sp)
    8166:   ce06                    sw  ra,28(sp)
    8168:   00008ab7            lui s5,0x8
    816c:   054000ef            jal 81c0 <uart_set_cfg>
    8170:   00008a37            lui s4,0x8
    8174:   0421                    addi    s0,s0,8
    8176:   0aa00993            li  s3,170 # aa <__DYNAMIC+0xaa>
    817a:   000f44b7            lui s1,0xf4
    817e:   05500913            li  s2,85 # 55 <__DYNAMIC+0x55>
    8182:   459d                    li  a1,7
    8184:   280a8513            addi    a0,s5,640 # 8280 <uart_wait_tx_done+0x18>
    8188:   01342023            sw  s3,0(s0) # 1a101000 <_stack_start+0x19ff9000>
    818c:   090000ef            jal 821c <uart_send>
    8190:   0d8000ef            jal 8268 <uart_wait_tx_done>
    8194:   24048793            addi    a5,s1,576 # f4240 <_edata+0xebf34>
    8198:   0037c0fb            lp.setup    x1,a5,819e <main+0x4e>
    819c:   0001                    nop
    819e:   0001                    nop
    81a0:   459d                    li  a1,7
    81a2:   288a0513            addi    a0,s4,648 # 8288 <uart_wait_tx_done+0x20>
    81a6:   01242023            sw  s2,0(s0)
    81aa:   072000ef            jal 821c <uart_send>
    81ae:   0ba000ef            jal 8268 <uart_wait_tx_done>
    81b2:   24048793            addi    a5,s1,576
    81b6:   0037c0fb            lp.setup    x1,a5,81bc <main+0x6c>
    81ba:   0001                    nop
    81bc:   0001                    nop
    81be:   b7d1                    j   8182 <main+0x32>
```

Figure 5.6 : make boot_code.read will create the assembly code for the given c code as boot_code.read located in the “/pulpino-master/sw/build/apps/boot_code”.Notice that this automatically creates “boot_code.elf”.

Applications

In this part, various Pulpino RI5CY labs will be examined. In this labs the core is used as a simple microprocessor. The labs are verified by implementation and simulation with xsim simulator.

5.4

5.4.1 LED Blink

```
#include <gpio.h>
#include <utils.h>
#include <pulpino.h>

void writeGpio(int input);
void delay(int input);

int main()
{
    while(1) // Forever
    {
        writeGpio(0xaa);    // Write 1010 1010
        delay(1000000);     // Wait
        writeGpio(0x55);    // Write 0101 0101
        delay(1000000);     // Wait
    }
}
void delay(int input)
{
    int x = 0;
    while(x++ < input) asm volatile("nop");         // "nop" is "No operation" assembly instruction.
                                                    // "volatile" command will protect the code after from
                                                    // Compiler's optimizations.
}

void writeGpio(int input)
{
    delay(0);                                       // Somehow this enable writing to gpio multiple times.
    *(volatile int*) (GPIO_REG_PADOUT) = input;     // GPIO pins has a dedicated address to them.
                                                    // GPIO_REG_PADOUT is the dedicated adress.
                                                    // Treating the address as a pointer we can
                                                    // write whatever we like.
}
```

Figure 5.7 : Main of led blink**.**

This code will simply write the first byte of the gpio_out pins 0xAA and 0x55 forever. To connect the gpio pins to the leds, necessary XDC changes need to be made. To see the proper locations of the board master XDC file must be downloaded via internet.

To upload the code to the FPGA, the created “boot_code.sv” contents must be copied in to Vivado project. After doing so, standart implementation flow can be followed. For code to work “fetch_enable” input also needs to be high. In the default state, “fetch_enable” is connected to SW0.

Bear in mind that xsim simulator may sometimes fail to simulate the lab. For unknown cause xsim simulator will fail to simulate long loops because it can not simulate the resulting assembly code. So, simulating long loops must be avoided.

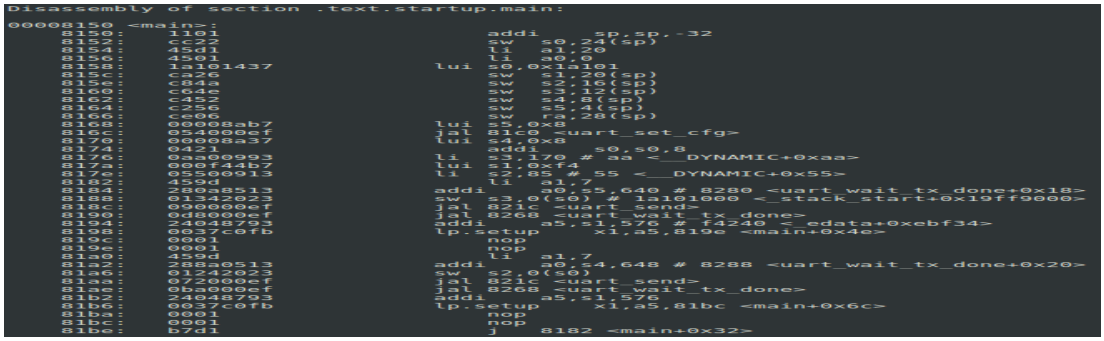


Figure 5.8 : In the addresses 0x8198 and 0x81b6 a hardware loop(lp.setup) is defined for long loops. This is text is result from boot_code.read command

5.4.2 UART

```
#include <gpio.h>
#include <uart.h>
#include <utils.h>
#include <pulpino.h>

int main()
{
    uart_set_cfg(0, 20);                        // Necessary clock division for
                                                // 100 MHz clock input to the system.
    uart_send("Hello World!\n", 13);            // Send string with 13 characters.
    uart_wait_tx_done();                        // Wait for transaction to finish.
    uart_send("I've come for peace!\n", 21);    // Send string with 21 characters.
    uart_wait_tx_done();                        // Wait for transaction to finish.

    return 0;
}
```

Figure 5.9 : Main of UART code

This code will send 2 string with uart_tx pin. Right now it's configuration with "Nexyx4DDR" is 115200 Baud rate with 8n1 setup. To see the result "minicom" terminal program need to be used. By using the command **sudo apt-get minicom** program can be downloaded form the terminal. After downloading the program the proper setup can be made by the command **sudo minicom -s** this will bring setup for UART type and the Port selection. To determine which usb port is the uart port, the terminal command **ls /dev/tty*** can be used. Simply plug-in and out the usb and look for the missing ttyUSBx.

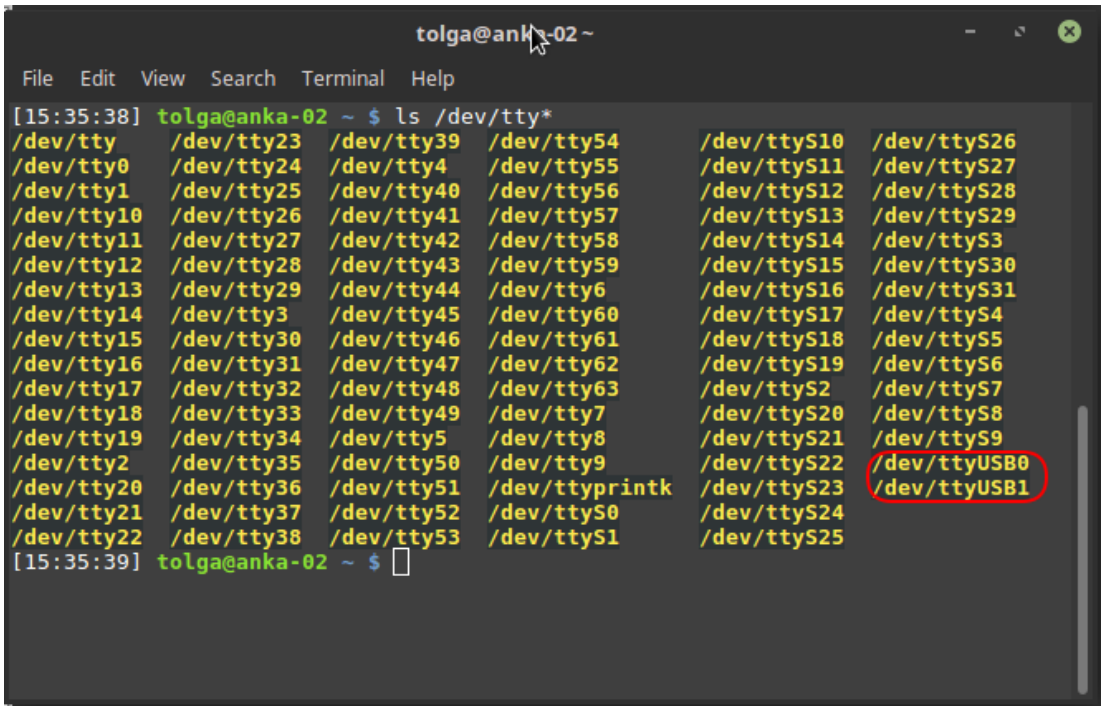


Figure 5.10 : The **ls /devtty*** command. In the red circle USB ports can be seen

Minicom program need to be opened with sudo privileges. To have nice format, after opening the minicom press CTRL-A then U this will enable cartridge return for '\n' character. CTRL-A then C will clear the console.

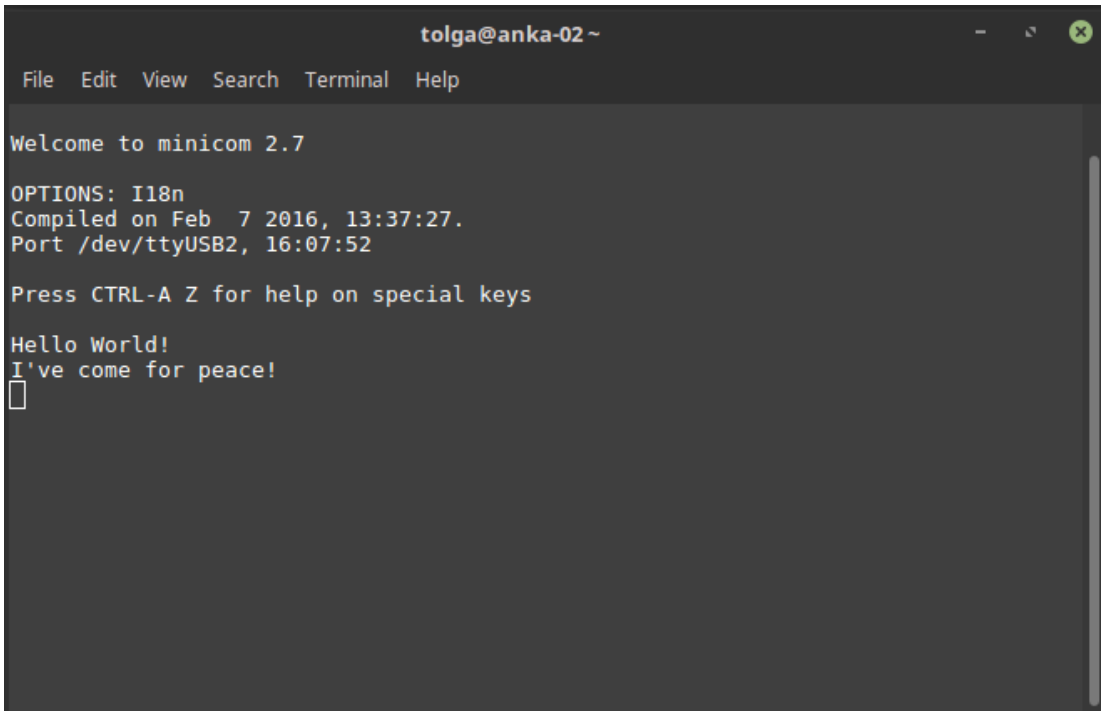


Figure 5.11 : Program's output seen with minicom 2.7.

Because of the Baud Rate depends highly on system clock, for custom implementations UART synchronization problems will occur and rubbish data will be displayed to the minicom console. To solve this problem easily, uart_set_cfg function can be sweeped and a string can be send each iteration.

```
for(x = 0;x<1000;x++)
{
        uart_set_cfg(0,x);
        uart_send("\n",1);
}
```

5.4.3 SPI

```
spi_setup_master(1);                            // Sets direction for SPI master pins with only one CS.

uart_set_cfg(0, 20);                            // UART 115200 8N1.

check_spi_flash();                              // Checks SPI flash's ID.
                                                // For some reason this need to be send twice for proper start.

*(volatile int*) (SPI_REG_CLKDIV) = 0x4;        // Divides system clock by 4 for SPI communication.

if (check_spi_flash())                          // If flash is not properly set up. Trap the software.
{
    uart_send("ERROR!\n", 7);
    while (1);
}


spi_setup_cmd_addr(0x06, 8, 0, 0);              // 0x06 is Write enable command. 8 is the command's lenght in bits.
                                                // 0 is the command's adress it can be as long as 4 bytes.
                                                // The last input 0 is adress' lenght in bits.
spi_set_datalen(0);                             // This sets the data lenght for the following transmision.
spi_start_transaction(SPI_CMD_WR, SPI_CSN0);    // This starts transaction. SPI_CMD_WR means the transaction is writing.
                                                // SPI_CSN0 means 0th slave will be communicated.
while ((spi_get_status() & 0xFFFF) != 1);       // This polls the spi transaction in progress flag.


int fifo_in [1];                                // Input fifo, this will hold the read value.
int fifo_out[1] = { 0x12345678 };               // Output fifo, holds the data to be writen.
int addr = 0x250;                               // The adress that is going to be used for loopback.

spi_setup_cmd_addr(0x21, 8, addr, 32);          // 0x12 is Sector Erase command. It uses single data pin.
                                                // 8 is command's lenght in bits.
                                                // addr is the command's adress.
                                                // 32 is the lenght of the adress.
spi_set_datalen( 32);                           // Data length is 32 bits since 1 integer is going to be writen.
spi_write_fifo(fifo_out, 32);                   // This will load the data to tx fifo.
spi_start_transaction(SPI_CMD_WR, SPI_CSN0);    // This starts transaction. SPI_CMD_WR means the transaction is writing.
                                                // SPI_CSN0 means 0th slave will be communicated.
while ((spi_get_status() & 0xFFFF) != 1);       // This polls the spi transaction in progress flag.

uart_send("Sector Erase!\n", 11);

while((read_spi_status() & 0x01) == 1);         // This polls for write in progress flag in the flash.
                                                // if WIP is 1 that means the writing operation is not done.
spi_setup_dummy(0, 0);                          // Dummy cycle count for the transaction.

spi_setup_cmd_addr(0x12, 8, addr, 32);          // 0x12 is Write command. It uses single data pin.
                                                // 8 is command's lenght in bits.
                                                // addr is the command's adress.
                                                // 32 is the lenght of the adress.
spi_set_datalen( 32);                           // Data length is 32 bits since 1 integer is going to be writen.
spi_write_fifo(fifo_out, 32);                   // This will load the data to tx fifo.
spi_start_transaction(SPI_CMD_WR, SPI_CSN0);    // This starts transaction. SPI_CMD_WR means the transaction is writing.
                                                // SPI_CSN0 means 0th slave will be communicated.
while ((spi_get_status() & 0xFFFF) != 1);       // This polls the spi transaction in progress flag.

uart_send("Data Sent!\n", 11);

while((read_spi_status() & 0x01) == 1);         // This polls for write in progress flag in the flash.
                                                // if WIP is 1 that means the writing operation is not done.
spi_setup_cmd_addr(0x13, 8, addr, 32);          // 0x13 is Read command. It uses single data pin.
                                                // 8 is command's lenght in bits.
                                                // addr is the command's adress.
                                                // 32 is the lenght of the adress.
spi_set_datalen( 32);                           // Data length is 32 bits since 1 integer is going to be writen.
spi_start_transaction(SPI_CMD_RD, SPI_CSN0);    // This starts transaction. SPI_CMD_RD means the transaction is reading.
spi_read_fifo(fifo_in, 32);                     // This will read the rx fifo.
while ((spi_get_status() & 0xFFFF) != 1);       // This polls the spi transaction in progress flag.

uart_send("Data Read!\n", 11);

uart_send_int(fifo_in[0]);

return 0;
```

Figure 5.12 : Main of spi loopback code

This code will make a loopback at desired address in the flash memory. The SPI protocol for the “s25fl128s” always need a command to be send from MOSI pin no matter if the transmission is quad mode. It will use only the MOSI pin. First of all, write enable pin must be sent to the flash before doing anything.

In this flash the transitions from 0 to 1 and 1 to 0 needs different instructions. To write correct data the flash must be erased(0 to 1). After that programming instruction can be run(1 to 0). Sending the correct instruction does not write the data instantly. Write in Progress flag in status register 1 must be polled in order to make sure that the write operation is successful.

The most troubling part about the SPI was figuring out which instructions to send in order for correct operation. Because of this, the datasheet must be read carefully.

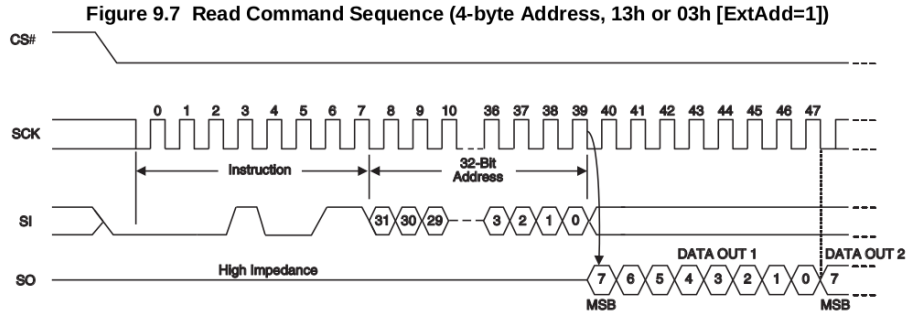


Figure 5.13 : Example instruction sequence with s25fl128s NOR flash. First the instruction(READ) is sent with SI(slave in), then the address of the data is sent. Then, the flash module will return the data as long as the SCK is kept running. Beware that the CS(chip select or slave select) signal is kept low for entire duration of the communication. The data from the flash will fill the input fifo of the RI5CY core, and the contents can be accessed by the function “read_fifo”[42].

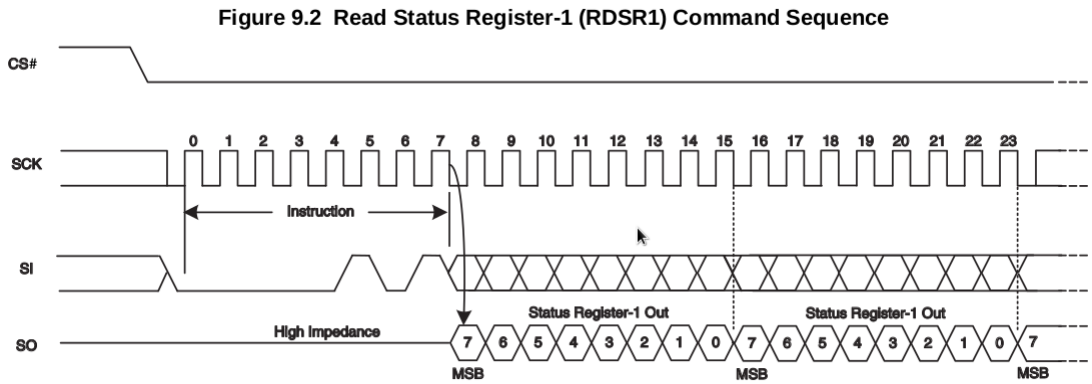


Figure 5.14 : The Read Status Register command. This can be sent after any program or erase commands to check whether the operation is complete. SCK can be continuously supplied to continuously read the status register-1[42].

Boot

5.5

```
//-------------------------------------------------------------
// Read header
//-------------------------------------------------------------

int header_ptr[8];
int addr = 0;

spi_setup_dummy(0, 0);

// cmd 0xEB fast read, needs 8 dummy cycles

int *instr = INSTR_RAM_BASE_ADDR; // Start writing from this adress.

//-------------------------------------------------------------
// Read Instruction RAM
//-------------------------------------------------------------

uart_send("CpyIns\n", 7);

addr = 0x0;
spi_setup_dummy(0, 0);
for (int i = 0; i < 1; i++)
{
//reads 16 4KB blocks
// cmd 0xEB fast read, needs 8 dummy cycles
spi_setup_cmd_addr(0x13, 8, ((addr << 8) & 0xFFFFFF00), 32);
spi_set_datalen(32768);
spi_start_transaction(SPI_CMD_RD, SPI_CSN0);
spi_read_fifo(instr, 32768);

instr += 0x400;  // new address = old address + 1024 words
addr  += 0x1000; // new address = old address + 4KB

uart_send_block_done(i);
}

while ((spi_get_status() & 0xFFFF) != 1);

//-------------------------------------------------------------
// Read Data RAM
//-------------------------------------------------------------

uart_send("Cpy D\n", 6);

uart_wait_tx_done();

uart_send("Done.\n", 6);

uart_wait_tx_done();
for(int x = 0; x < 100; x++)
{
    uart_send_int(*((volatile unsigned int *)INSTR_RAM_START_ADDR + x));
    uart_wait_tx_done();
}
//-------------------------------------------------------------
// Set new boot address -> exceptions/interrupts/events rely
// on that information
//-------------------------------------------------------------

BOOTREG = 0x00;

//-------------------------------------------------------------
// Done jump to main program
//-------------------------------------------------------------

//jump to program start address (instruction base address)
jump_and_start((volatile int *)(INSTR_RAM_START_ADDR));
```

Figure 5.15 : Main of boot.c

Booting from flash is simply reading the instructions from the flash to the instruction memory in the RI5CY core. In this lab a single 4KB block is read from the flash to the instruction memory. After reading is complete the programs jumps to “instruction base address + 0x80” because reset handler program is stored here. After reset handler, the program’s main is ran.

To create the boot image, first the programs elf file is needed. After creating the elf file it needs to be converted to bin file. This can be done by using objcopy. Lastly the endian order of the bin file must be changed. Vivado changes endian when writing to the flash.

Elf file can be created by using the command:

make boot_code.elf

Elf to bin conversion can be made with command:

riscv32-unknown-elf-objcopy -O binary input.elf output.bin

Endian change can be made with the command:

riscv32-unknown-elf-objcopy -I binary -O binary --reverse-bytes=4 input.elf output.bin

After creating the bin file properly. The bin file can now be transferred to the flash memory. To do this hardware manager needs to be opened. After auto connecting and selecting the fpga from the hardware list in the hardware tab. Configure Memory Device option needs to be selected.

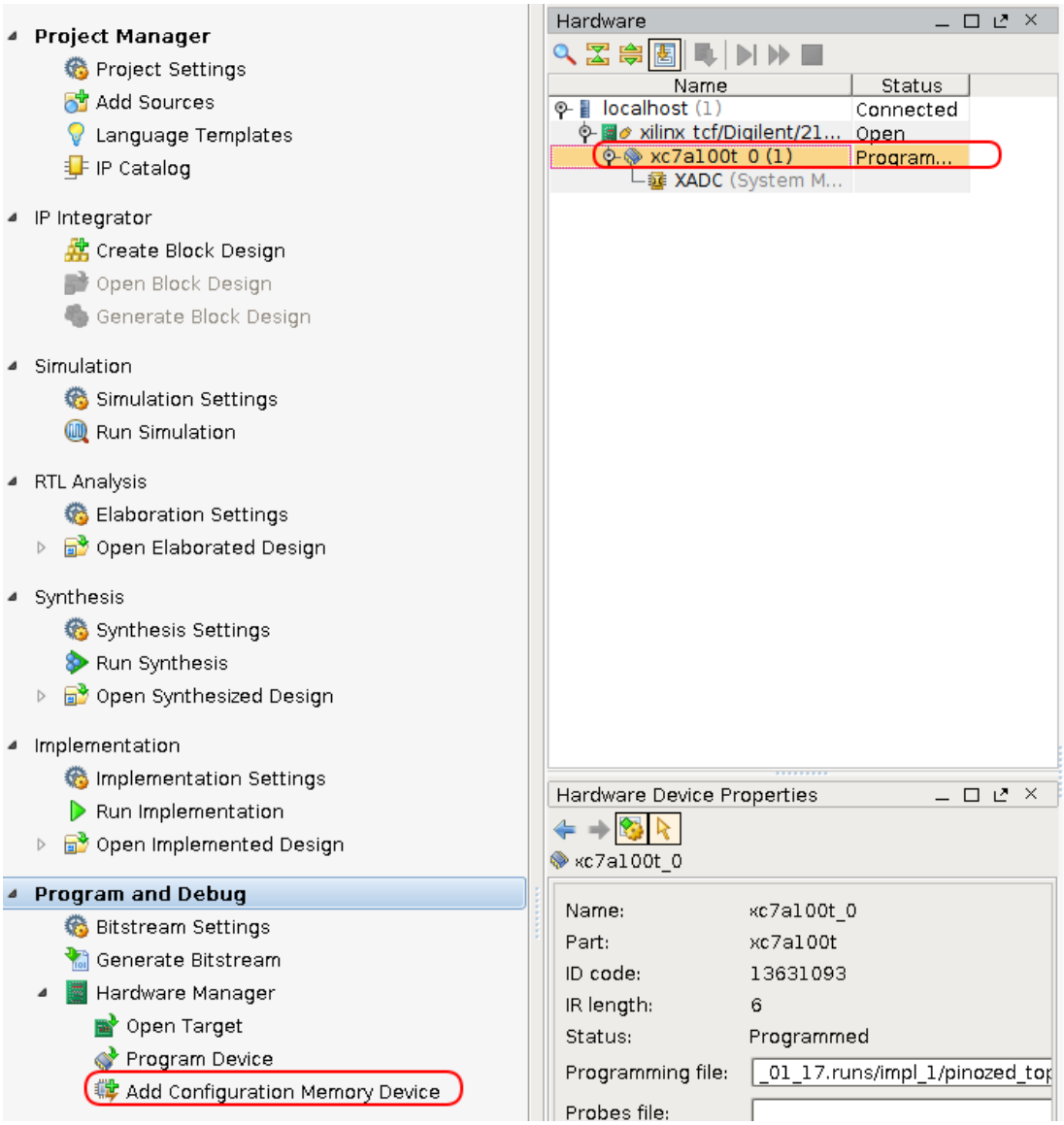


Figure 5.16 : Screen Shot for hardware manager

First select the fpga “xc7a100t” then select Add Configuration Memory device. After that select the bin file that is going to be loaded to the flash. After doing that the core should boot from the flash.

5.6 Project Plan and Current Status

Right now, I'm able run any c code whether in boot rom or flash. I have an understanding of the core and, I know the hiearcy. I was distracted because of GPIO, SPI and UART problems which made me behind schedule I am currently at 14th week on Table 11. However the problems they caused gave me insight on the core much more. Which means that I understand the instruction cycle and the hiearcy better. From now on I need to add my extension hardware into ALU then I can call it easily by changing the bin file.

REFERENCES

[1] P. Grabher, J. Großschädl, and D. Page, "Light-weight instruction set extensions for bit-sliced cryptography," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 5154 LNCS, pp. 331–345, 2008.

[2] Z. Hou, D. Sanan, A. Tiu, Y. Liu, and K. C. Hoa, "An executable formalisation of the SPARCv8 instruction set architecture: A case study for the LEON3 processor," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 9995 LNCS, no. 1, pp. 388–405, 2016.

[3] D. Lampret, C. Chen, M. Mlinar, and J. Rydberg, "OpenRISC 1000 architecture manual," *Descr. Assem. ...*, no. C, pp. 1–331, 2003.

[4] A. J. Salim, N. R. Samsudin, S. I. M. Salim, and Y. Soo, "Multiply-accumulate instruction set extension in a soft-core RISC Processor," *2012 10th IEEE Int. Conf. Semicond. Electron. ICSE 2012 - Proc.*, pp. 512–516, 2012.

[5] F. Dresden and L. Fanucci, "Instruction Set Extensions for Secure Applications," *Proc. 2016 Conf. Des. Autom. Test Eur.*, pp. 1529–1534, 2016.

[6] B. Bilgin, E. B. Kavun, and T. Yalcin, "Towards an ultra lightweight crypto processor," *Proc. - 2011 Work. Light. Secur. Priv. Devices, Protoc. Appl. Light. 2011*, pp. 76–83, 2011.

[7] H. Groß and T. Plos, "On using instruction-set extensions for minimizing the hardware- implementation costs of symmetric-key algorithms on a low-resource microcontroller," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 7739 LNCS, pp. 149–164, 2013.

[8] S. Tillich, J. Großschädl, and A. Szekely, "An instruction set extension for fast and memory-efficient AES implementation," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 3677 LNCS, pp. 11–21, 2005.

[9] D. Selent, "Advanced encryption standard," *Fed. Inf. Process. Stand. FIPS-197*, vol. 197, no. Fips 197, pp. 1–10, 2001.

[10] S. J. Shepherd, "The Tiny Encryption Algorithm," *Cryptologia*, vol. 31, no. 3, pp. 233–245, 2007.

[11] A. Kchaou, W. El Hadj Youssef, R. Velazco, and R. Tourki, "An exhaustive analysis of seu effects in the sram memory of soft processor," *J. Eng. Sci. Technol.*, vol. 13, no. 1, pp. 58–68, 2018.

[12] C. Damman, G. Edison, F. Guet, E. Noulard, L. Santinelli, and J. Hugues, "Architectural performance analysis of FPGA synthesized LEON processors," *Proc. 27th Int. Symp. Rapid Syst. Prototyp. Shortening Path from Specif. to Prototype - RSP '16*, pp. 33–40, 2016.

[13] P. Ranganathan, S. Adve, and N. P. Jouppi, "Performance of image and video processing with general-purpose processors and media ISA extensions," *Proc.*

26th Int. Symp. Comput. Archit. (Cat. No.99CB36367), vol. 0, no. 3604, pp. 124–135, 1999.

[14] P. G. De Massas, P. Amblard, and F. Pétrot, "On SPARC LEON-2 ISA extensions experiments for MPEG encoding acceleration," *VLSI Des.*, vol. 2007, 2007.

[15] A. Kchaou, W. El Hadj Youssef, and R. Tourki, "Software implementation of AES algorithm on leon3 processor," *STA 2014 - 15th Int. Conf. Sci. Tech. Autom. Control Comput. Eng.*, pp. 237–242, 2014.

[16] A. S. Eissa, M. A. Elmohr, M. A. Saleh, K. E. Ahmed, and M. M. Farag, "SHA-3 Instruction Set Extension for A 32-bit RISC processor architecture," *Proc. Int. Conf. Appl. Syst. Archit. Process.*, vol. 2016–November, pp. 233–234, 2016.

[17] N. Ben Hadjy Youssef, W. El Hadj Youssef, M. Machhout, R. Tourki, and K. Torki, "Instruction set extensions of AES algorithms for 32-bit processors," *Proc. - Int. Carnahan Conf. Secur. Technol.*, vol. 2014–October, no. October, 2014.

[18] C. C. Gaisler, "GRLIB IP Library User ' s Manual," no. April, 2015.

[19] L. G. Guide and L. G. Guide, "LEON/GRLIB Design and Configuration Guide," pp. 1–32, 2016.

[20] G. Ip, M. The, and U. M. November, "GRMON2 User â€™ s Manual," no. November, pp. 1–218, 2017.

[21] "BCC User â€™ s Manual," no. June, 2017.

[22] DIGILENT, "Atlys™ FPGA Board Reference Manual," p. 19, 2018.

[23] D. Coppersmith, "The Data Encryption Standard (DES) and its strength against attacks," *IBM J. Res. Dev.*, vol. 38, no. 3, pp. 243–250, 1994.

[24] N. Fips, "Announcing the ADVANCED ENCRYPTION STANDARD (AES)," *Byte*, vol. 2009, no. 12, pp. 8–12, 2001.

[25] "Xilinx webpage." [Online]. Available: https://www.xilinx.com/support/download.html. [Accessed: 20-Mar-2018].

[26] "Leon/GRLIB." [Online]. Available: https://www.gaisler.com/index.php/downloads/leongrlib. [Accessed: 20-Mar-2018].

[27] "BCC binaries." [Online]. Available: https://www.gaisler.com/anonftp/bcc2/bin/. [Accessed: 20-Mar-2018].

[28] "BCC source files." [Online]. Available: https://www.gaisler.com/anonftp/bcc2/src/. [Accessed: 20-Mar-2018].

[29] "GRMON website." [Online]. Available: https://www.gaisler.com/index.php/downloads/debug-tools. [Accessed: 20-Mar-2018].

[30] SPARC International Inc., "The SPARC Architecture Manual V8," p. 295, 1992.

[31] A. Bogdanov *et al.*, "PRESENT : An Ultra-Lightweight Block Cipher,"

Springer Berlin Heidelb., pp. 450–466, 2007.

[32] C. A. Lara-Nino, A. Diaz-Perez, and M. Morales-Sandoval, “Lightweight Hardware Architectures for the Present Cipher in FPGA,” *IEEE Trans. Circuits Syst. I Regul. Pap.*, vol. 64, no. 9, pp. 2544–2555, 2017.

[33] A. Khattab, Z. Jeddi, E. Amini, and M. Bayoumi, “RFID Security,” 2017.

[34] ARM Limited Inc., "ARM v6-M Architecture Reference Manual" p. 26, 2017.

[35] ARM Limited Inc., "ARM Cortex-M0 DesignStart,Eval Revision:r2p0,User Guide" pp. 1.5-2.2, 2017.

[36] Web.fi.uba.ar, 2018. [Online]. Available: http://web.fi.uba.ar/~pmartos/publicaciones/SASE2011_ImplementingTheCortexM0DesignStartProcessorInALowEndFPGA.pdf. [Accessed: 02- Apr- 2018].

[37] Xilinx.com, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/university/Vivado-Teaching/HDL-Design/2013x/Nexys4/Supporting%20Material/Nexys4_RM_VB1_Final_3.pdf. [Accessed: 02- Apr- 2018].

[38] Sase.com.ar, 2018. [Online]. Available: http://www.sase.com.ar/2012/files/2012/09/M0_v6M_Q312.pdf. [Accessed: 02- Apr- 2018].

[39] "RISC-V Foundation | Instruction Set Architecture (ISA)", *RISC-V Foundation*, 2018. [Online]. Available: https://riscv.org/. [Accessed: 02- Apr- 2018].

[40] "pulp-platform/pulpino", *GitHub*, 2018. [Online]. Available: https://github.com/pulp-platform/pulpino/blob/master/doc/datasheet/datasheet.pdf. [Accessed: 02- Apr- 2018].

[41] *Pulp-platform.org*, 2018. [Online]. Available: https://pulp-platform.org//wp-content/uploads/2017/11/ri5cy_user_manual.pdf. [Accessed: 02- Apr- 2018].

[42] *Cypress.com*, 2018. [Online]. Available: http://www.cypress.com/file/177966/download. [Accessed: 02- Apr- 2018].

[43] "Ubuntu Linux Install GNU GCC Compiler and Development Environment - nixCraft", *nixCraft*, 2018. [Online]. Available: https://www.cyberciti.biz/faq/howto-installing-gnu-c-compiler-development-environment-on-ubuntu/. [Accessed: 02- Apr- 2018].

[44] "riscv/riscv-tools", *GitHub*, 2018. [Online]. Available: https://github.com/riscv/riscv-tools/blob/master/README.md. [Accessed: 02- Apr- 2018].

[45] "pulp-platform/pulpino", *GitHub*, 2018. [Online]. Available: https://github.com/pulp-platform/pulpino/blob/master/README.md. [Accessed: 02- Apr- 2018].