ISTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

LOW-POWER GENERAL PURPOSE PROCESSOR DESIGN AND INSTRUCTION SET EXTENSION FOR AES

M.Sc. THESIS

Muhammed ŞAİROĞLU

Department of Electronics and Communication Engineering

Electronic Engineering

Thesis Advisor: Assoc. Prof. Dr. Sıddıka Berna Örs YALÇIN

MARCH 2020

ISTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

LOW-POWER GENERAL PURPOSE PROCESSOR DESIGN AND INSTRUCTION SET EXTENSION FOR AES

M.Sc. THESIS

Muhammed ŞAİROĞLU
(504171254)

Department of Electronics and Communication Engineering

Electronic Engineering

Thesis Advisor: Assoc. Prof. Dr. Sıddıka Berna Örs YALÇIN

MARCH 2020

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

DÜŞÜK GÜÇ TÜKETİMLİ GENEL AMAÇLI İŞLEMCİ TASARIMI VE AES İÇİN KOMUT KÜMESİ GENİŞLETİLMESİ

YÜKSEK LİSANS TEZİ

Muhammed ŞAİROĞLU
(504171254)

Elektronik ve Haberleşme Mühendisliği Anabilim Dalı

Elektronik Mühendisliği Programı

Tez Danışmanı: Assoc. Prof. Dr. Sıddıka Berna Örs YALÇIN

MART 2020

Muhammed ŞAİROĞLU, a M.Sc. student of ITU Institute of Science and Technology 504171254 successfully defended the thesis entitled **"LOW-POWER GENERAL PURPOSE PROCESSOR DESIGN AND INSTRUCTION SET EXTENSION FOR AES"**, which he/she prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Assoc. Prof. Dr. Sıddıka Berna Örs YALÇIN** Istanbul Technical University	....
Jury Members :	**Asst. Prof. Ayşe Yılmazer Metin** Istanbul Technical University	....
	Asst. Prof. Tuba Ayhan MEF University	....
		....

Date of Submission : 27 March 2020
Date of Defense : 6 April 2020

To my family

FOREWORD

I would like to thank my supervisor Assoc. Prof. Dr. Sıddıka Berna Örs YALÇIN for her guidance, kind advice, and help throughout my M.Sc studies.

I am also grateful to Panasonic Life Solutions Turkey company for giving me the opportunity to complete my studies along with my work.

March 2020 Muhammed ŞAİROĞLU
Embedded Software Engineer

TABLE OF CONTENTS

Page

ABBREVIATIONS

AES	: Advanced Encryption Standard
ALU	: Arithmetic Logic Unit
ASIC	: Application-Specific Integrated Circuit
ASIP	: Application-Specific Instruction Set Processor
CFG	: Control Flow Graph
DES	: Data Encryption Standard
FPGA	: Field Programmable Gate Arrays
GF	: Galois Field
GPP	: General-Purpose Processor
HEX	: Hexadecimal numeral system
IM	: Instruction Memory
IoT	: Internet of Things
IR	: Instruction Register
NOP	: No Operation Instruction
PC	: Program Counter
RAM	: Random Access Memory
VHDL	: Very High Speed Integrated Circuit Hardware Description Language

LIST OF TABLES

LIST OF FIGURES

LOW-POWER GENERAL PURPOSE PROCESSOR DESIGN AND INSTRUCTION SET EXTENSION FOR AES

SUMMARY

In the last years, there has been a big growth in the demand for portable electronic devices. Most of these devices need to operate on a thrifty energy budget and they must be designed to work under extreme energy constraints for a long time. Also, a lot of smart devices need to communicate with the outer world and with other devices, and all these communications must be secure. These requirements have increased the investments in developing low-power integrated circuits with encryption capabilities.

In this thesis, a low-power general purpose processor design is presented. Then the processor design is improved by extending the instruction set with instructions for the Advanced Encryption Standard (AES).

In chapter one, many embedded systems architectures for low-power applications are introduced, then the Advanced Encryption Standard is explained.

In chapter two, the designed processor's instruction set is given, and its architecture is explained in detail. Then the processor architecture is improved by adding many pipeline stages. Pipeline hazards are handled without complicating the processor architecture. Both processor designs (the non-pipelined and the pipelined) were tested with simple programs to compare its performances. The pipelined processor showed better results in terms of the required clock cycles to finish test programs, the throughput and the consumed energy. Both processor designs were also compared with the well-known Xilinx PicoBlaze processor. The pipelined processor beat PicoBlaze according to the maximum clock rate and dynamic on chip power.

In chapter three, The AES algorithm is implemented in Assembly language and is run on the pipelined processor. Then AES algorithm code is investigated using its control flow graphs. New instructions are added to the standard instruction set by combining related and sequential instructions from the algorithm code and creating new instructions that solves software problems faster. It is showed that the added instructions reduced the required time to finish AES encryption to 52% and the consumed power to 37% without having a significant increase in the architecture size.

DÜŞÜK GÜÇ TÜKETİMLİ GENEL AMAÇLI İŞLEMCİ TASARIMI VE AES İÇİN KOMUT KÜMESİ GENİŞLETİLMESİ

ÖZET

Son yıllarda, taşınabilir elektronik cihazlara olan talepte büyük bir artış olmuştur. Bu cihazların çoğunun enerji tasarrufu yapabilmesi ve bu şekilde uzun süre çalışabilecek şekilde tasarlanması gerekmektedir. Ayrıca, birçok akıllı cihazın dış dünyayla ve diğer cihazlarla iletişim kurması gerekmektedir ve tüm bu iletişim güvenli şekilde sağlanmalıdır. Bu gereksinimler, şifreleme özelliklerine sahip düşük güçlü entegre devrelerin geliştirilmesine yönelik yatırımları artırmıştır.

Bu tezde genel amaçlı kullanılabilecek düşük güçlü işlemci tasarımı sunulmuştur. Daha sonra, işlemcinin komut seti Gelişmiş Şifreleme Standardı (AES) için yeni talimatlarla genişletilerek geliştirilmiştir.

Birinci bölümde düşük güçlü uygulamalar için birçok gömülü sistem mimarisi tanıtılmış, ardından Gelişmiş Şifreleme Standardı (AES) anlatılmıştır.

İkinci bölümde, tasarlanmış işlemcinin komut seti verilmiş ve mimarisi ayrıntılı olarak anlatılmıştır. Daha sonra birçok boru hattı aşaması eklenerek işlemci mimarisi geliştirilmiştir. Boru hattı tehlikeleri, işlemci yapısını karmaşıklaştırmadan giderilmiştir. Her iki işlemcinin tasarımları (boru hattı olmayan ve boru hattı olan), performanslarını karşılaştırmak için basit programlarla test edilmiştir. Boru hattı işlemcisi test programlarını bitirmek için gereken saat döngüleri, verimleri ve tüketilen enerjileri açısından daha iyi sonuçlar vermiştir. Her iki işlemci de herkes tarafından bilinen Xilinx PicoBlaze işlemcisi ile karşılaştırılmıştır. Boru hattı işlemcisi, PicoBlaze işlemcisini dinamik gücü ve maksimum saat hızı açısından yenmiştir.

Üçüncü bölümde, AES algoritması Assembly dilinde yazılmıştır ve boru hattı işlemcisinde çalıştırılmıştır. Daha sonra kontrol akış grafiğini kullanarak AES algoritma kodu incelenmiştir. Algoritma kodunda birbirine bağlı ve art arda gelen komutlar birleştirilerek ve yazılım problemlerini daha hızlı çözebilen yeni komutlar oluşturarak standart komut setine yeni komutlar eklenmiştir. Eklenen komutlar, işlemcinin mimari boyutunda önemli bir artış yapmadan AES şifrelemesini bitirmek için gereken süreyi 52%'ye ve tüketilen enerjiyi 37%'ye düşürdüğü gösterilmiştir.

1. INTRODUCTION

1.1 The Era of Low-Power Devices

Nowadays, Portable electronic devices are widely used in everyone daily life, and they are getting involved in our lives more and more. These devices majorly depends on rechargeable batteries, and the low power consumption in these devices translates to longer run time on a full charged battery, and higher number of charge cycles until the end of useful battery life. These are very important end user care-about. Also, low power translates to less heat dissipation which means fewer cooling parts and smaller designs.

Low power consumption is becoming more important in portable electronic devices markets as many users started to choose devices with better battery life and smaller size over devices with higher performance and capabilities.

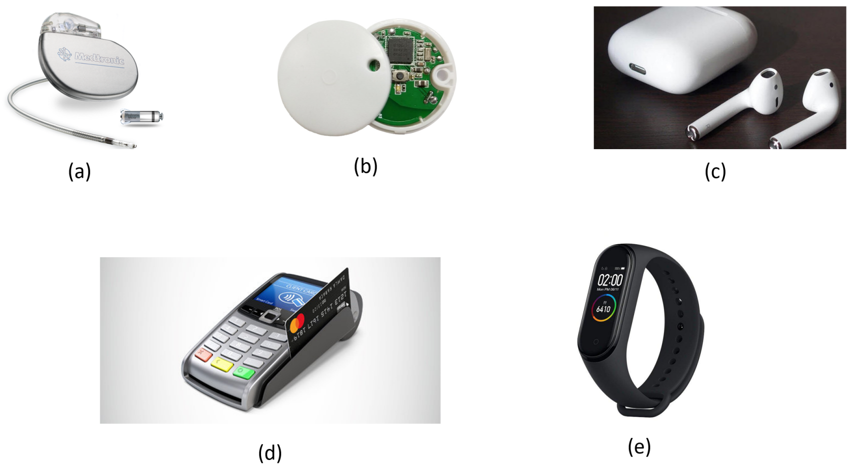


Figure 1.1: Figure illustrates examples of widely-used low-power electronic devices. (a) pacemaker, (b) IoT sensor, (c) wireless headphones, (d) POS terminal, (e) smartwatch.

1.2 Choosing the Right Architecture for Low-Power Applications

When electronic engineers are asked to design a low-power device, they are faced with a myriad of core technologies, all claiming to best save power for a given application. So how do they know which one will meet their energy consumption requirements? The next section examines the benefits of the different architectures and compare the design trade-offs between them.

1.2.1 Application-specific integrated circuit (ASIC)

As the name implies, ASIC is an integrated circuit chip manufactured for a particular use, not for general-purpose use. Some examples of ASIC chip include a battery charging circuit in a mobile phone, high-efficiency bitcoin miner, video decoder etc.

ASICs offer high application-specific performance because the designer can tune hardware gates specific to the target application. Furthermore, ASICs can achieve decent power efficiency when the design is specifically targeted for power efficiency.

However, ASIC solutions suffer from their lack of flexibility as they cannot be reprogrammed to implement new algorithms. This means that only a single application or specification can be supported, and a separate ASIC is needed for each new application and specification. In addition, the cost of building new ASIC chips using latest manufacturing technology is increasingly high, particularly for relatively small quantities.

1.2.2 General-purpose processor (GPP)

A general-purpose processor is capable of performing many different functions under the direction of instructions. The general-purpose processor can execute another task, if a different set of instructions are given.

General-purpose processor based solutions have the advantage of being off-the-shelf and less expensive. However, higher power and area consumption, and lower speed performance are potential disadvantages for GPP compared to more application-specific implementations, because they target a broad range of embedded applications.

1.2.3 Application-specific instruction set processor (ASIP)

ASIP is application dependent instruction processor. It is used for processing the various instruction set inside a combinational circuit of an embedded system.

The idea of ASIP is to get the best out of general-purpose processor programmability while at the same time trying to offer performance efficiency as high as ASIC's. The primary approach is to maximize the design domain of the microarchitecture by actively adding particular instructions.

This specialization of the core provides a trade-off between the flexibility of a GPP and the good performance and power consumption of an ASIC.

Table 1.1 gives a comparison between GPP, ASIP and ASIC according to the performance, flexibility, power consumption and reusing. Figure 1.2 depicts the tradeoff between energy-efficiency and flexibility for several architecture paradigms.

Table 1.1: A comparison between GPP, ASIP and ASIC

	ASIC	GPP	ASIP
Performance	Very high	Low	High
Flexibility	Poor	Excellent	Good
Power	Small	Large	Medium
Reuse	Poor	Excellent	Good

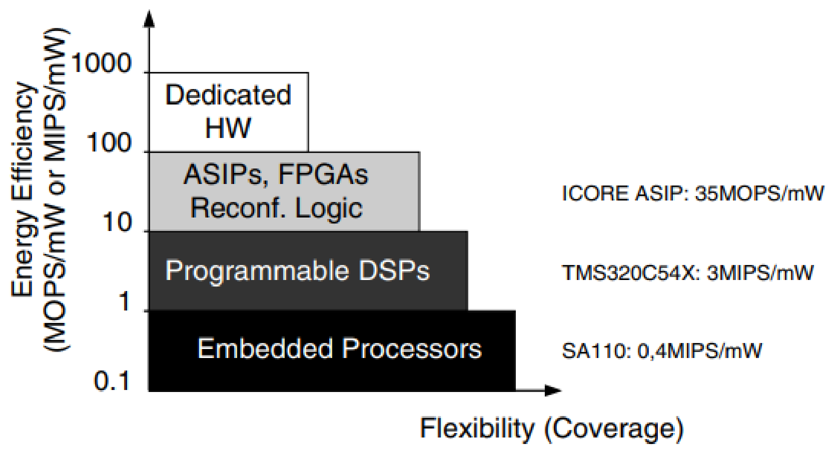


Figure 1.2: Energy flexibility trade-off for several embedded systems architectures [1]

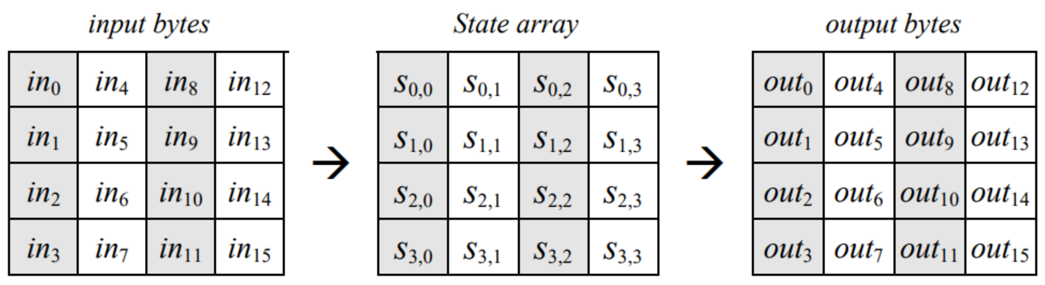
ASIP is a good architecture choice for low-power applications when the flexibility is a required feature or when the application is too complex to be done as a dedicated hardware.

1.3 The Advanced Encryption Standard (AES)

AES is an encryption standard accepted by the United States government. It is also known as Rijndael cipher. As DES (the Data Encryption Standard) [2] algorithm became weak and lost its reliability in the face of developing technology, NIST (the National Institute of Standards and Technology) organized a competition in order to set a new encryption standard. Two Belgian researchers Joan Daemen and Vincent Rijmen won the competition with their Rijndael algorithm. NIST published AES as U.S. (FIPS 197) standard [3] on November 26, 2001 after a long standardization and verification process. AES provides higher reliability, and it also has advantages in terms of being easy to implement compared to the DES.

Although the algorithm supports different key and block size, the standard includes 128-bit, 192-bit or 256-bit key lengths with a fixed 128-bit block size. In AES, 128-bit data blocks are considered as 4 words, each consisting of 32-bit. When starting the encryption process with AES, the 128-bit, 4-word data block is written into the state array and all the necessary operations during the algorithm are performed using this array. After the last operation of encryption, the final version of the state array is written to the output array.

For example; as illustrated in Figure 1.3, the input data block that consists of {in_0, in_1 ... in_15} bytes is written to the state array and all necessary operations are performed on this array. After the operations are completed, the encrypted data is copied to the output as {out_0, out_1, . . . , out_15} byte array.



input bytes
in_0 in_4 in_8 in_{12}
in_1 in_5 in_9 in_{13}
in_2 in_6 in_{10} in_{14}
in_3 in_7 in_{11} in_{15}
State array
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
output bytes
out_0 out_4 out_8 out_{12}
out_1 out_5 out_9 out_{13}
out_2 out_6 out_{10} out_{14}
out_3 out_7 out_{11} out_{15}

Figure 1.3: AES state array input and output

The AES algorithm generally consists of two processes, the first process is cipher process, and the second process is key expansion process.

The algorithm has a repetitive structure, in cipher process, the round transformations are repeated many times depending on the length of the key. The number of rounds according to the key length is given in Table 1.2.

Table 1.2: Key length round combinations in AES

AES Type	Key length	Number of Rounds
AES-128	128	10
AES-192	192	12
AES-256	256	14

1.3.1 Cipher process

At the start of this process, the input data block is copied to the state array, then four different byte-oriented transformations are applied on the state. They are:

- SubBytes: byte substitution using a substitution table (S-box),
- ShiftRows: shifting rows of the state array by different offsets,
- MixColumns: mixing the data within each column of the state array,
- AddRoundKey: adding a round key to the state.

These transformations are described in details in the following subsections.

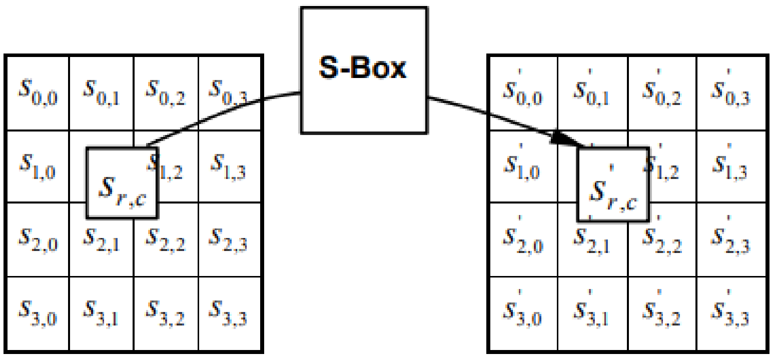
Cipher process for 128-bit key length is described in pseudocode in Code 1.1.

Code 1.1: Cipher process pseudocode

```
Cipher (byte in[16], byte out[16], byte expanded_key [176])
{
  byte state [16] = in;
  AddRoundKey(state, expanded_key,0);
  for (round = 1; round  <= 9; round++)
  {
    SubByte(state);
    ShiftRows(state);
    MixColumns(state);
    AddRoundKey(state, expanded_key, round);
  }
  SubByte(state);
  ShiftRows(state);
  AddRoundKey(state, expanded_key, 10);
  Out = state;
}
```

1.3.1.1 SubBytes transformation

SubBytes transformation is a non-linear operation that is performed on each byte of the state independently as shown in Figure 1.4. In this function each byte in the state array is replaced with a byte from an 8-bit substitution box (S-box). The output of this function is different for each different input.



S-Box
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,2}$ $s_{1,3}$
$s_{r,c}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s'_{0,0}$ $s'_{0,1}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{r,c}$
$s'_{2,0}$ $s'_{2,1}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,1}$ $s'_{3,2}$ $s'_{3,3}$

Figure 1.4: AES SubBytes transformation

S-box values can be obtained in two stages. The first stage is finding the multiplicative inverse of the input in the finite field $GF(2^8)$. The polynomial used to define this field is $p(x) = x^8 + x^4 + x^3 + x + 1$. 0 is mapped to itself because it doesn't have a multiplicative inverse. The second stage is applying an affine transformation which can be described as multiplying and adding the output of the previous stage (as a polynomial over $GF(2^8)$) with constant matrices. Figure 1.5 illustrates this operation.

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

Figure 1.5: AES affine transformation

S-box can be implemented in different ways. Some implementation methods are examined in detail in section 3.2.2

S-box output for each possible input is given in Figure 1.6 in hexadecimal representation.

		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 1.6: AES S-box table

1.3.1.2 ShiftRows transformation

In ShiftRows transformatıon, all the rows are shifted, except for the first row of the state matrix. Row 2 is shifted one byte, row 3 is shifted two bytes, and the last row is shifted three bytes. The block diagram of ShiftRows is given in Figure 1.7.

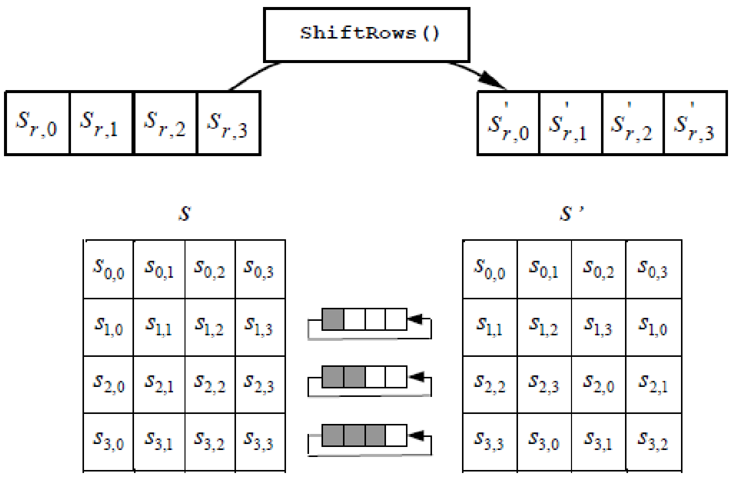

ShiftRows()
$S_{r,0}$ $S_{r,1}$ $S_{r,2}$ $S_{r,3}$
$S'_{r,0}$ $S'_{r,1}$ $S'_{r,2}$ $S'_{r,3}$
S
S'
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,1}$ $s_{1,2}$ $s_{1,3}$ $s_{1,0}$
$s_{2,2}$ $s_{2,3}$ $s_{2,0}$ $s_{2,1}$
$s_{3,3}$ $s_{3,0}$ $s_{3,1}$ $s_{3,2}$

Figure 1.7: AES ShiftRows transformation

1.3.1.3 MixColumns transformation

MixColumns transformation is performed independently on each column in the state matrix. While performing this operation, each column is considered as a polynomial in $GF(2^8)$ and multiplied modulo x^4+1 with a fixed polynomial $a(x)$, given by

$a(x) = 3 * x^3 + x^2 + x + 2$

MixColumns transformation can also be performed as a matrix multiplication. Let

$s'(x) = a(x) * s(x)$

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

Where s(x) is the state column before the transformation and s'(x) is the new state column after the transformation. Figure 1.8 illustrates the MixColumns transformation.

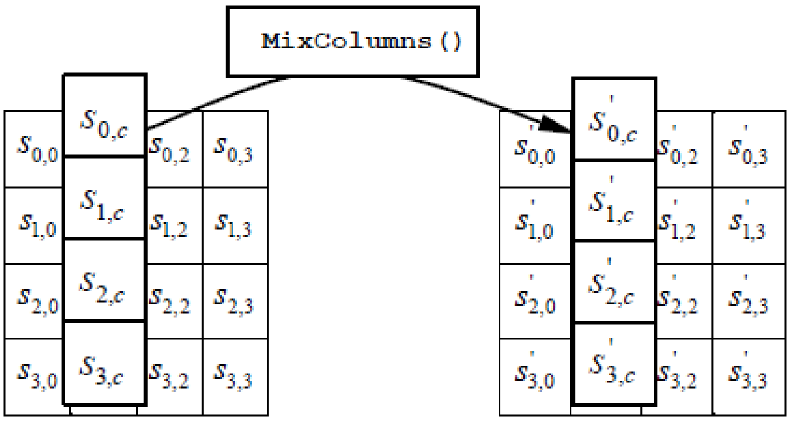


Figure 1.8: AES MixColumns transformation

1.3.1.4 AddRoundKey transformation

In AddRoundKey transformation, the state matrix is XORed with a 128-bit round key matrix that is generated in key expansion process before. Key expansion process will be examined in detail in the next section. The general diagram of the AddRoundKey transformation is given in Figure 1.9.

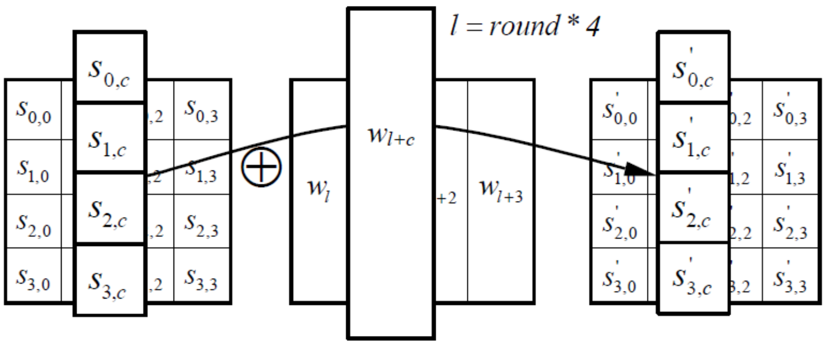


Figure 1.9: AES AddRoundKey transformation

1.3.2 Key expansion process

The AES algorithm takes the K key array and generates the necessary key blocks for each round. These key blocks are also known as the round keys. The AES algorithm can work with different key lengths, but since the length of the state is fixed at 128-bit, the generated round keys length is 128-bit too. The round key is used in AddRoundKey transformation which is the last transformation in round transformations. Key expansion process generates a total of $4 * (\text{number of rounds} + 1)$ words: the AES algorithm requires an initial set of 4 words, and each of the rounds requires 4 words of key block. The resulted expanded key is an array of 4-bytes words with length equals to $4 * (\text{number of rounds} + 1)$.

The Key Expansion process for 128-bit key length is described in pseudocode in Code 1.2.

Code 1.2: Key Expansion Process pseudocode

```
KeyExpansion(byte key[16], word expanded_key[44])
{
  for (j = 0; j < 4; j++)
  {
    expanded_key[j] = word(key[4*j+0], key[4*j+1], key[4*j+2], key[4*j+3]);
  }
  for (j=4; j < 44; j++)
  {
    word temp = expanded_key[j-1];
    if (j % 4 == 0)
    {
      temp = SubWord(RotWord(temp)) ^ Rcon[j/4];
    }
```

```
        expanded_key[j] = expanded_key[j-4] ^ temp;
    }
}
```

SubWord function applies SubBytes transformation on a four-byte input word and produce an output word. RotWord function applies a cyclic permutation on a four-byte input word [b0, b1, b2, b3] and returns the word [b1, b2, b3, b0]. Rcon is the round constant word array, it consists of values given by $[2^{(j-1)}$ in $GF(2^8)$, 0, 0, 0] where j starts at 1. Figure 1.10 illustrates AES key expansion process for a 128-bit key.

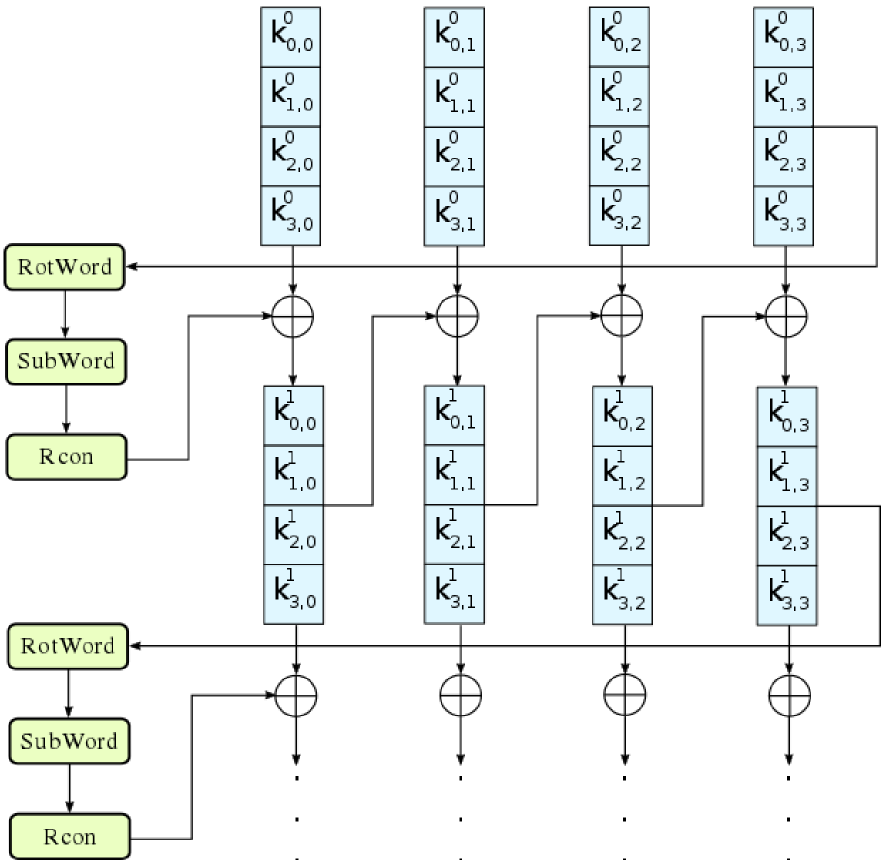


Figure 1.10: AES key expansion process

1.4 ASIP Designing Guideline

Before starting to design an ASIP for AES, we set a guideline to follow. It includes the following rules:

(a) **Build the design from scratch:** This makes us fully knowing the design details, and when power and area analyses are done, we can easily track the problematic components and fix them.

(b) **Build it as simple as possible:** We have to avoid any complexity in the design as possible. Complex designs cause more power consumption, larger chip area and slower clock rate.

(c) **Build a general-purpose processor then extend its instruction set:** Starting an ASIP project with making a general-purpose processor before adding the extended instructions gives you a chance to test and verify your design before the things mix up. Also, it helps you to know the effect of the extended instructions on the total design according to the power consumption and operating frequency.

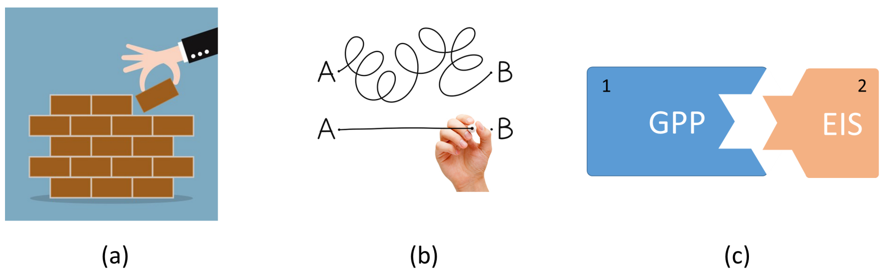


Figure 1.11: Figure illustrates the set ASIP designing guideline.

2. GENERAL-PURPOSE PROCESSOR DESIGN

The first step we took in designing a general-purpose processor was determining its instruction set, then according to the determined instruction set a data path and a control unit is designed.

After finishing our first design we found that we can improve it by pipelining it. To be sure that the added pipeline stages don’t cause losses in power consumption and operating frequency, many tests were done, and its results are reported. Next sections will explain in details the processor design, the made improvements, simulation steps and the applied tests.

2.1 The Instruction Set

The processor has 24 different instructions which have the same width 18 bit. The first 5 bits of each instruction are used for the operation code and the other bits usage differs. The instruction set is listed and explained in Table 2.1.

LOD operation is used to load a register with a specific value. FTC operation is used to fetch data from the data memory to a register, and STR operation is used to store data into the data memory from a register. MOV operation is used to load a register with a value from another register.

There are 7 different jump instructions. Each one of them is used to jump to an absolute address or a relative address. The relative address is coded as 11-bit signed value so it can be jumped +2047 instruction forward or -2048 instruction backward at a time.

There is also "No Operation" instruction that has no effect on the registers or the data memory and it can be used as a delay slot especially after and before jump instructions in the enhanced processor.

Table 2.1: The instruction set of the designed processor

Instruction	Function	Code
Registers and Data Memory Operations		
LOD Rx,Value	Rx = Value	00000 rrrr 0vvvvvvvv
FTC Rx,[N]	[N] = Rx	00001 rrrr 0nnnnnnnn
FTC Rx,[R15]	[R15] = Rx	00001 rrrr 100000000
STR [N],Rx	Rx = [N]	00010 rrrr 0nnnnnnnn
STR [R15],Rx	Rx = [R15]	00010 rrrr 100000000
MOV Rd,Rs	Rd = Rs	00011 dddd 00000ssss
Jump Operations		
JMP Add	Unconditional Jump to direct address	00100 0 aaaaaaaaaaaa
JMP Rel	Unconditional Jump to relative address	00100 1 eeeeeeeeeeee
JZ/JE Add	Jump if zero / equal	00101 0 aaaaaaaaaaaa
JZ/JE Rel	Jump if zero / equal	00101 1 eeeeeeeeeeee
JNZ/JNE Add	Jump if non zero / not equal	00110 0 aaaaaaaaaaaa
JNZ/JNE Rel	Jump if non zero / not equal	00110 1 eeeeeeeeeeee
JC/JB Add	Jump if carry/ below	00111 0 aaaaaaaaaaaa
JC/JB Rel	Jump if carry/ below	00111 1 eeeeeeeeeeee
JNC/JAE Add	Jump if not carry / above or equal	01000 0 aaaaaaaaaaaa
JNC/JAE Rel	Jump if not carry / above or equal	01000 1 eeeeeeeeeeee
JA Add	Jump if above	01001 0 aaaaaaaaaaaa
JA Rel	Jump if above	01001 1 eeeeeeeeeeee
JBE Add	Jump if below or equal	01010 0 aaaaaaaaaaaa
JBE Rel	Jump if below or equal	01010 1 eeeeeeeeeeee
ALU Operations		
AND Rd,Ry.Rz	Rd = Ry & Rz	01011 dddd 0zzzz yyyy
OR Rd,Ry,Rz	Rd = Ry | Rz	01100 dddd 0zzzz yyyy
XOR Rd,Ry,Rz	Rd = Ry ^Rz	01101 dddd 0zzzz yyyy
NOT Rd,Ry	Rd = ∼Ry	01110 dddd 00000 yyyy
SHL Rd,Ry	Rd = Ry<<1	01111 dddd 00000 yyyy
SHR Rd,Ry	Rd = Ry>>1	10000 dddd 00000 yyyy
ROL Rd,Ry	Rd = Ry rol 1	10001 dddd 00000 yyyy
ROR Rd,Ry	Rd = Ry ror 1	10010 dddd 00000 yyyy
ADD Rd,Ry,Rz	Rd = Ry + Rz	10011 dddd 0zzzz yyyy
INC Rd,Ry	Rd = Ry + 1	10100 dddd 00000 yyyy
SUB Rd,Ry,Rz	Rd = Ry - Rz	10101 dddd 0zzzz yyyy
DEC Rd,Ry	Rd = Ry - 1	10110 dddd 00000 yyyy
CMP Ry,Rz	Compare Rz with Ry	10111 0000 0zzzz yyyy
NOP	No operation	11111 1111 11111 1111

All other instructions are ALU operations. Some of them have 2 operands and some of them have 3 operands. The destination register of all ALU operations can be one of the source register/s or a different register.

2.2 The Data Path

The data path is a set of functional units that carry out data processing operations. Its diagram is shown in Figure 2.1. The data memory and the register file have a clock input to synchronize the writing operations. However, reading operations are not synchronized so the memory cell or the register value will be shown on the output data bus as soon as its address is on the address bus.

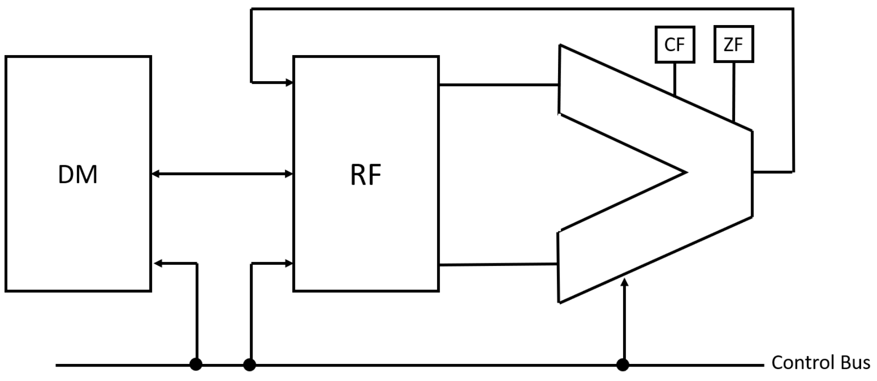


Figure 2.1: The data path diagram

2.2.1 The arithmetic logic unit

The one byte-wide arithmetic logic unit (ALU) performs all processor calculations, including:

- bitwise logic operations such as AND, OR, XOR and NOT
- shift and rotate operations
- basic arithmetic operations such as addition, subtraction, increment and decrement
- arithmetic compare

The ALU also gives the status of the executed ALU operation result. The status signals are carry-out and zero. The carry-out flag state changes with arithmetic operations only (ADD, INC, SUB, DEC, CMP) while the zero flag state changes with all ALU operations. The ALU diagram is shown in Figure 2.2.

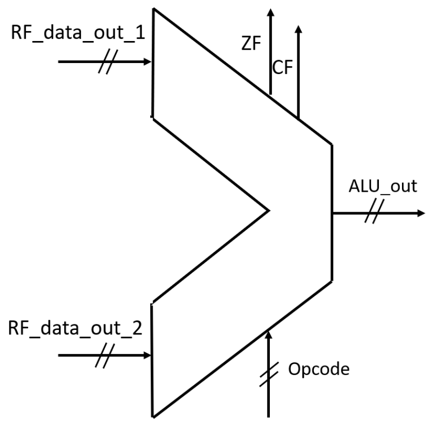


Figure 2.2: The ALU diagram

2.2.2 The register file

The register file of the designed processor has 16 8-bit wide general-purpose registers and they are designated as R0, R1, .., R15. Its diagram is shown in Figure 2.3. Register file input data can be from another register, data memory, ALU operation result or a LOD instruction value. The fifteenth register's value can be used as a pointer to a location in the data memory so there is a bus mapped directly to it.

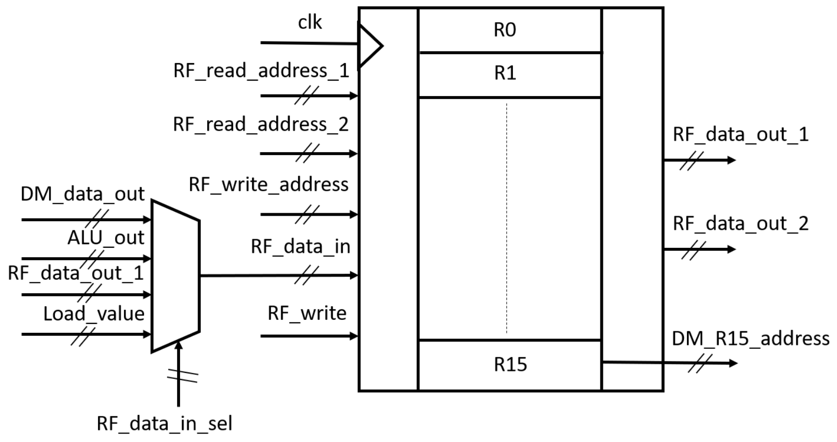


Figure 2.3: The register file diagram

2.2.3 The data memory

The data memory is a simple 256-byte RAM. Its diagram is shown in Figure 2.4. Thanks to the 2-way MUX on the address input, The data memory's 8-bit address can be specified by the control unit to be either a direct address from an instruction, or an indirect address from the content of the fifteenth register (R15) of the register file.

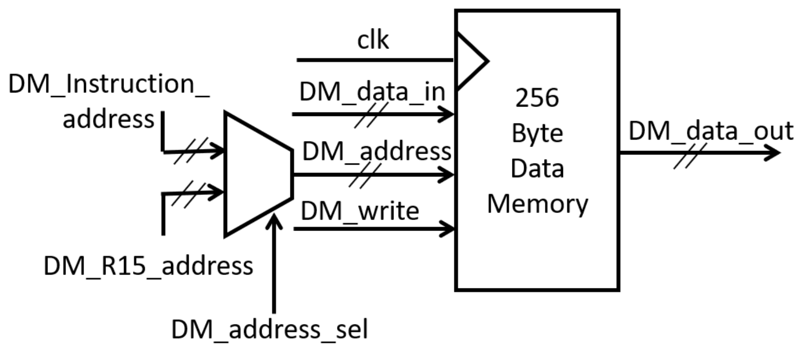


Figure 2.4: The data memory diagram

2.3 The Control Unit

The diagram of the control unit and its all sub units is shown in Figure 2.5.

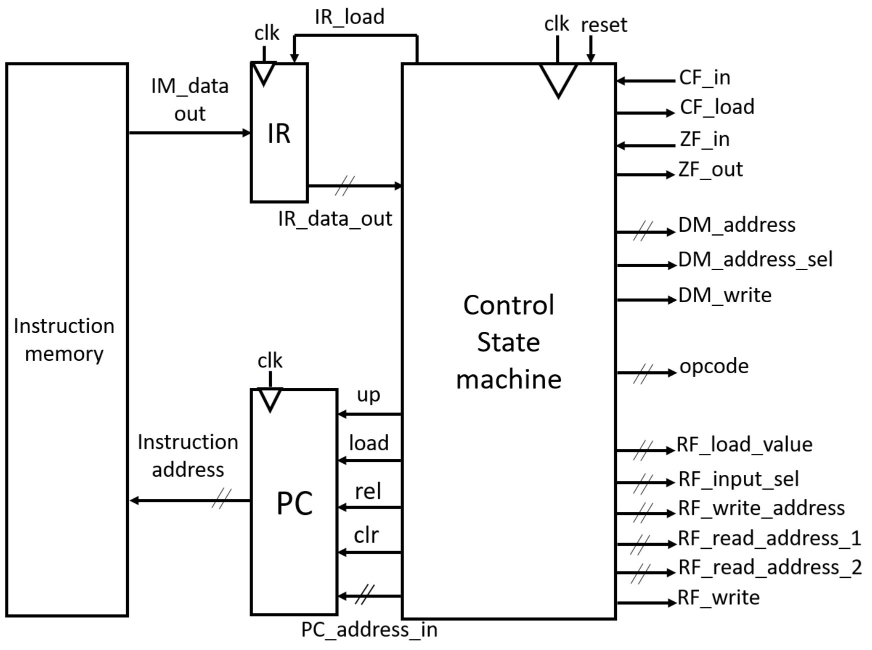


Figure 2.5: The control unit diagram in the non-pipelined processor

2.3.1 The instruction memory

The designed processor can execute up to 4096 instructions from a single block RAM. Each instruction is 18-bit wide. The output data of the instruction memory (IM) is connected to instruction register (IR) that is used to hold the instruction for the control state machine.

2.3.2 The program counter

The program counter (PC) points to the next instruction to be executed. According to the control signals that come from the control state machine, the next instruction address can be a specific absolute address, a relative address, the instruction just after the current instruction or the first instruction in the instruction memory. If the 12-bit PC reaches the top of the memory at 0xFFF, it rolls over to location 0x000.

2.3.3 The control state machine

The control state machine has 3 states: Initialize, Fetch and Execute. In Initialize state the program counter is cleared. In Fetch state the instruction register is loaded with an instruction from the instruction memory and the program counter increases its counter. In Execute state the instruction in the instruction register is executed after it is decoded, and all control signals and addresses are set. The transition between these states is shown in Figure 2.6.

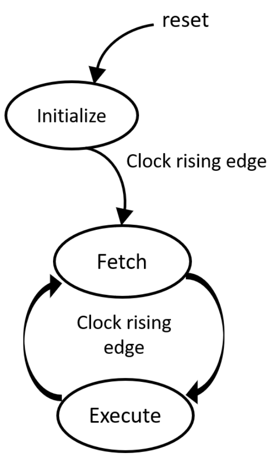


Figure 2.6: The state diagram of the control state machine in the non-pipelined processor

2.4 Improving the General-Purpose Processor Design

Our aim in this work is to reduce the average clock cycles number that is required to finish executing one instruction. In the previous design two clock cycles were required to execute an instruction and the processor cannot process more than one instruction at the same time.

In the enhanced design a three-stage pipeline was implemented. The first stage is represented by the instruction memory and the program counter. The second stage is represented by the control state machine. The last stage is represented by the data path. In the first stage an instruction is fetched from the instruction memory and loaded to the instruction register. In the second stage the fetched instruction is decoded and the control signals of the data path and the program counter are produced then loaded to state registers. In the third stage the instruction is executed and stored. The new control unit diagram is shown in Figure 2.7.

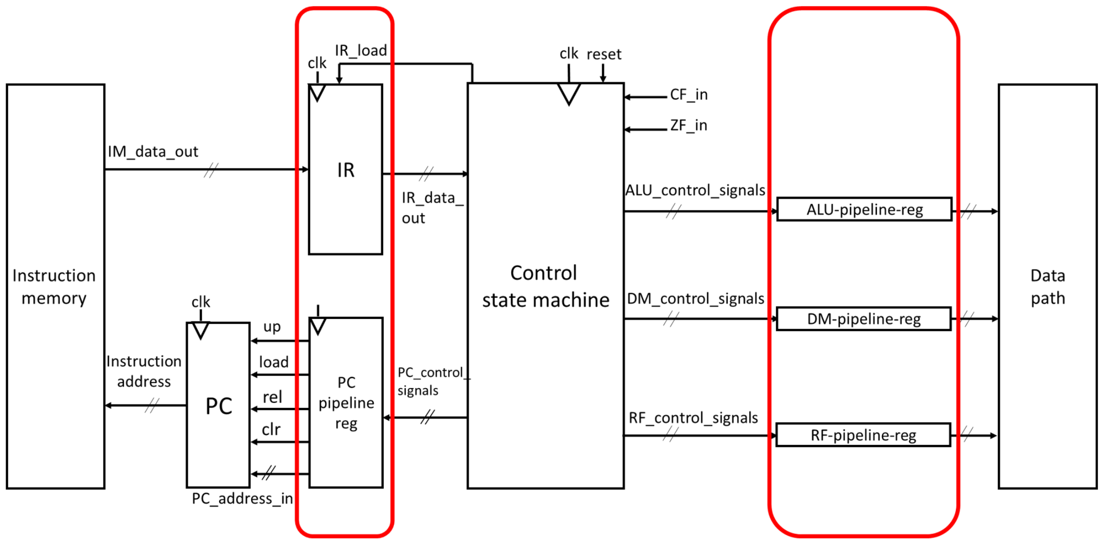


Figure 2.7: The control unit diagram in the pipelined processor

Fetch and Execute states of the former processor's control state machine are merged into one new state "Run" in the new pipelined processor. The new state diagram is shown in Figure 2.8. In this new state an instruction is decoded and executed while a new instruction is fetched from the instruction memory at the same cycle. Although an instruction requires three clock cycles to fully processed in this design, the average clock cycles number that is required to finish executing one instruction became one clock cycle, this is because the processor processes three instructions at the same time. This makes our processor relatively fast among 8-bit processors.

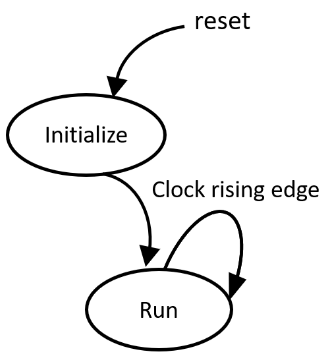


Figure 2.8: The state diagram of the control state machine in the pipelined design

2.4.1 Pipeline hazards handling

In every pipelined processor architecture there are three types of hazards can be occurred. They are structural hazards, data hazards and control hazards. In order to keep the processor design as simple as possible no new hardware units are wanted to be added, and all hazards are avoided without complicating the processor structure.

2.4.1.1 Structural hazards

Structural hazards arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution [4]. In our pipeline design two instructions cannot be executed in the ALU at the same time so this type of hazards cannot be occurred in the processor.

2.4.1.2 Data hazards

Data hazards arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline [4]. In our pipeline design this can happen when a jump instruction's condition depends on the result of the instruction that is being executed at the same time. We can avoid this problem by adding a delay represented by a NOP instruction before each conditional jump instruction.

2.4.1.3 Control hazards

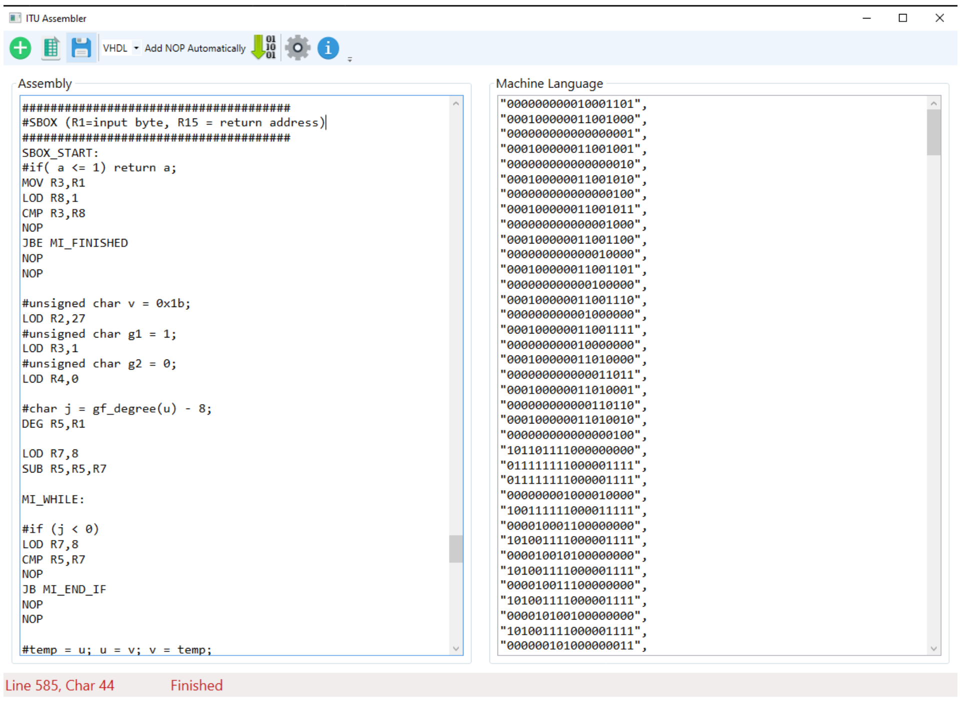
Control hazards arise from the pipelining of instructions that change the PC [4]. In our pipeline design this can happen when a jump instruction will be executed but the instruction register is already loaded with the next instruction and the PC is pointing to the instruction that comes after them. A simple way to solve this problem is adding two NOP instructions after each jump instruction.

2.5 The Assembler

A simple assembler program has been developed with C# to help us in writing test programs. The used assembler user interface is shown in Figure 2.9. This assembler is capable of:

- Giving meaningful and clear error messages if the entered assembly code has some syntax errors.
- Saving assembly codes to a file and opening a previously saved one.
- Showing the cursor position as line and character number.
- Translating the assembly code to HEX code or VHDL code (to be used in instruction memory).
- Adding delay instructions (NOP) automatically if the target is the pipelined processor.

Also, the assembler C# code is written in a way makes adding new instructions to decode is so easy.



```
########################################
#SBOX (R1=input byte, R15 = return address)
########################################
SBOX_START:
#if( a <= 1) return a;
MOV R3,R1
LOD R8,1
CMP R3,R8
NOP
JBE MI_FINISHED
NOP
NOP

#unsigned char v = 0x1b;
LOD R2,27
#unsigned char g1 = 1;
LOD R3,1
#unsigned char g2 = 0;
LOD R4,0

#char j = gf_degree(u) - 8;
DEG R5,R1

LOD R7,8
SUB R5,R5,R7

MI_WHILE:

#if (j < 0)
LOD R7,8
CMP R5,R7
NOP
JB MI_END_IF
NOP
NOP

#temp = u; u = v; v = temp;
```

```
"0000000000010001101",
"00010000011001000",
"00000000000000001",
"00010000011001001",
"00000000000000010",
"00010000011001010",
"00000000000000100",
"00010000011001011",
"00000000000001000",
"00010000011001100",
"00000000000010000",
"00010000011001101",
"00000000000100000",
"00010000011001110",
"00000000001000000",
"00010000011001111",
"00000000010000000",
"00010000011010000",
"00000000000011011",
"00010000011010001",
"00000000000110110",
"00010000011010010",
"00000000000000100",
"10110111100000000",
"01111111100001111",
"01111111100001111",
"00000000100001000",
"10011111100011111",
"00001000110000000",
"10100111100001111",
"00001001010000000",
"10100111100001111",
"00001001110000000",
"10100111100001111",
"00001010010000000",
"10100111100001111",
"00000010100000011",
```

Line 585, Char 44 Finished

Figure 2.9: The developed assembler user interface

2.6 Simulations

2.6.1 Simulation method

Both designs were described with Very High Speed Integrated Circuit Hardware Description Language (VHDL) [5] and then implemented and simulated in Xilinx Vivado Environment [6]. First of all, the maximum clock frequency of the designs was found. Then the Switching Activity Interchange Format (SAIF) [7] files were generated by running "Post-Implementation Timing Simulation". The generated SAIF files is used in Vivado power reports to give more accurate results. In simulation steps we verified the processors behavior.

2.6.2 Simulation results

Both designs were compared with Xilinx PicoBlaze processor [8] according to instruction memory size, RAM size, maximum clock rate and power consumption. All processors were tested on Xilinx Spartan 7 Series XC7S6 FPGA. The test results are shown in Table . 2.2.

Table 2.2: Comparison between the designed processors and Xilinix PicoBlaze processor

Processor	Non-pipelined	Pipelined	Xilinx PicoBlaze
Instruction mem.	4K	4K	4K
RAM	256 byte	256 byte	256 byte
Maximum clock rate	110 MHz	155 MHz	135 MHz
Dynamic on-chip power	0.006 W	0.007 W	0.008 W

Both designs consume less power than PicoBlaze processor. The non-pipelined processor has a lower maximum frequency than PicoBlaze processor's maximum frequency, but the pipelined processor's maximum frequency is higher. This is because of the pipeline stages which allow to cut the delay of the critical paths in the processor.

Two simple programs were used to test the non-pipelined and the pipelined processors. The first program calculates the multiplication of two numbers from the data memory then writes the result to it. The second program calculates the modular multiplication

((A * B) mod N, where A, B < N) of numbers from the data memory and returns the result to it too. Algorithm 1 and Algorithm 2 demonstrate the used algorithms in these programs.

Algorithm 1 Multiplication program

```
procedure MULTIPLY($C = A \times B$)
    $C \leftarrow 0$
    for $i \leftarrow B : 0$ do
        $C \leftarrow C + A$
    end for
end procedure
```

Algorithm 2 Modular multiplication program

```
procedure MODULAR MULTIPLICATION($C = A * B \bmod N$)
    $C \leftarrow 0$
    for $i \leftarrow k - 1 : 0$ do
        $C \leftarrow C \times 2$
        if $C > N$ then
            $C \leftarrow C - N$
        end if
        $C \leftarrow C + b_i \times A$
        if $C > N$ then
            $C \leftarrow C - N$
        end if
    end for
end procedure
```

The required number of cycles to finish these two programs, the throughput and the energy consumption at the maximum frequency for the non-pipelined and the pipelined designs are reported in Table 2.3 and Table 2.4.

Table 2.3: The performance results of the non-pipelined and the pipelined processors for the multiplication program

Multiplication program	Non-pipelined processor	Pipelined processor
Clock cycles	34	28
Throughput	24.6 Mbit/s	42.2 Mbit/s
Energy	1.8 nW/s	1.2 nW/s

The pipelined processor finished both programs with clock cycles less than the non-pipelined processor, and its throughput and energy results are better because it can operate at higher frequency.

Table 2.4: The performance results of the non-pipelined and the pipelined processors for the modular multiplication program

Modular multiplication program	Non-pipelined processor	Pipelined processor
Clock cycles	198	196
Throughput	4.2 Mbit/s	6 Mbit/s
Energy	10.2 nW/s	8.8 nW/s

2.7 Chapter Conclusion

The first step in our ASIP project was making a general-purpose processor. We designed a simple general-purpose processor by determining its standard instruction set then implementing its data path and control unit using VHDL in Vivado Environment. Then we improved our design by pipelining it. Pipeline hazards are avoided without complicating the processor structure. Finally, we compared the simulation results of both designs with Xilinx PicoBlaze processor. Both designs consumed less power than PicoBlaze processor, and the pipelined processor's maximum frequency was higher than PicoBlaze processor's maximum frequency. Also, the pipelined design finished testbench programs with clock cycles less than the non-pipelined design and consumed less energy.

3. EXTENDING THE INSTRUCTION SET FOR AES

3.1 Rules for Extending the Instruction Set

Before starting to extend the instruction set of the designed processor, we had to put some rules to ensure that the added instructions don't turn our processor away from the goals of this project. These rules are:

- The added instructions cannot be too complex, else it will lead to a long-time delay and reduce the efficiency of the entire system.
- New instructions shouldn't add new massive hardware, else the power consumption in the processor will increase significantly.
- The opcodes and the operands of an added instruction must fit with the original instructions codes structure and cannot be longer than them.

3.2 Design Flow of the Extended Instruction Set

Our extended instruction set design flow consists of 5 steps, they are:

3.2.1 Dividing the algorithm into several independent functions

We divided the AES algorithm to many independent functions. They are round transformations functions (SubBytes - ShiftRows - MixColumns - AddRoundKey), KeyExpanstion function, and S-box function.

3.2.2 Implementing the functions of the algorithm in C

We wrote the C code for each AES function and test it separately then we combined them and verified the whole algorithm. The inputs of the algorithm which are the initial state and the cypher key are fed to the algorithm from RAM. The expanded key and the output state of the algorithm are stored to RAM too.

There are several ways to implement S-box function. One of them is hardcoding the S-box table into RAM. We avoided this way because our processor RAM size is 256 bytes and sbox table size is 256 bytes too, this gives us no room to use RAM for another purpose.

On the other hand, S-box function output can be generated by getting the multiplicative inverse in Galois Field $GF(2^8) = GF(2)[x]/(x^8 + x^4 + x^3 + x + 1)$ of the input then applying the following affine transformation.

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Where [b7, b6, . . . , b1, b0] is the multiplicative inverse and [s7, s6, . . . , s1, s0] is the S-box output as a vector.

The transformation can be simplified to be easy to implement in C language as in the next formula.

```
s = b ^  (b <<< 1) ^ (b <<< 2) ^ (b <<< 3) ^ (b <<< 4) ^  99
```

Where b is the multiplicative inverse of the S-box input, s is the S-box output, ^ is the bitwise xor operator and <<< is the bitwise rotate left operator.

Calculating the multiplicative inverse in $GF(2^8)$ can be done in different ways [9]. The easiest way is multiplying the input with every element in the field until the result of the multiplication operation is one. Then the multiplicative inverse is found. This is a brute-force search. However, this method suffers from poor performance and a high running time because in the worst case it is required to examine every number in the field. Another way to find the multiplicative inverse is using the extended Euclidean algorithm [10]. This algorithm shows better performance and shorter running time than the previous way.

In this study, both methods along with MixColumns transformation are implemented in C and examined, then many new instructions are added.

3.2.3 Translating the C code to assembly code for our processor

All C language programs have to be converted into the low-level realities of the processors' digital hardware, and our program is not an exception. As there is no compiler to do this job for us, we translated the C code to assembly code by hand.

3.2.4 Drawing the CFGs of the assembly code and examining them to figure out which instructions to be added

Control flow graph (CFG) is a representation of all paths that could be traversed by a program during its execution, using graph notations [11]. Basically, CFGs are mostly used in both code static analysis and compiler implementation and applications. We used CFGs in finding sequential instructions that can be combined and complex sequence of instructions that is created to do one job and can be simplified with hardware components.

Before starting drawing CFGs we examined many compiler-generated CFGs then we chose GCC CFGs to be a template for ours. An example of GCC CFG is shown in Figure 3.1

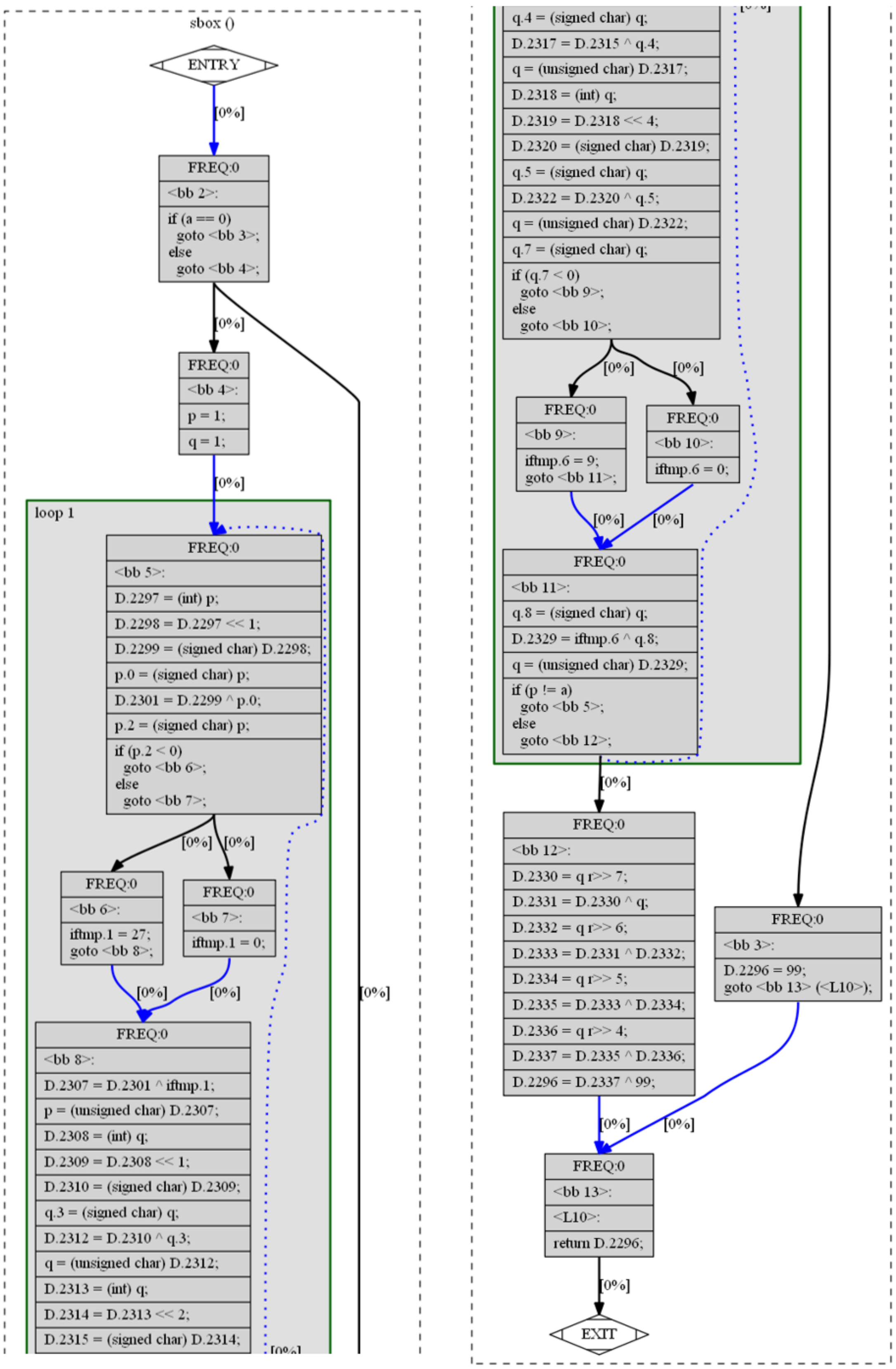
To generate a CFG for a C code in GCC we have to add this compiler option

-fdump-tree-all-graph

Then the compiler will generate many dot files for the compiled code. Dot is a plain text graph description language; it defines a graph but not the layout of the graph. We used Graphviz [12] (an open source graph visualization software) to convert dot files to visualized data like PNG images.

3.2.5 Finding candidate instructions in a CFG and converting them into a new instruction

The candidate instructions should be frequently invoked instructions during the program life cycle, else the consequences of adding a new instruction like the increase



```
FREQ:0
<bb 2>:
if (a == 0)
  goto <bb 3>;
else
  goto <bb 4>;

FREQ:0
<bb 4>:
p = 1;
q = 1;

FREQ:0
<bb 5>:
D.2297 = (int) p;
D.2298 = D.2297 << 1;
D.2299 = (signed char) D.2298;
p.0 = (signed char) p;
D.2301 = D.2299 ^ p.0;
p.2 = (signed char) p;
if (p.2 < 0)
  goto <bb 6>;
else
  goto <bb 7>;

FREQ:0
<bb 6>:
iftmp.1 = 27;
goto <bb 8>;

FREQ:0
<bb 7>:
iftmp.1 = 0;

FREQ:0
<bb 8>:
D.2307 = D.2301 ^ iftmp.1;
p = (unsigned char) D.2307;
D.2308 = (int) q;
D.2309 = D.2308 << 1;
D.2310 = (signed char) D.2309;
q.3 = (signed char) q;
D.2312 = D.2310 ^ q.3;
q = (unsigned char) D.2312;
D.2313 = (int) q;
D.2314 = D.2313 << 2;
D.2315 = (signed char) D.2314;
q.4 = (signed char) q;
D.2317 = D.2315 ^ q.4;
q = (unsigned char) D.2317;
D.2318 = (int) q;
D.2319 = D.2318 << 4;
D.2320 = (signed char) D.2319;
q.5 = (signed char) q;
D.2322 = D.2320 ^ q.5;
q = (unsigned char) D.2322;
q.7 = (signed char) q;
if (q.7 < 0)
  goto <bb 9>;
else
  goto <bb 10>;

FREQ:0
<bb 9>:
iftmp.6 = 9;
goto <bb 11>;

FREQ:0
<bb 10>:
iftmp.6 = 0;

FREQ:0
<bb 11>:
q.8 = (signed char) q;
D.2329 = iftmp.6 ^ q.8;
q = (unsigned char) D.2329;
if (p != a)
  goto <bb 5>;
else
  goto <bb 12>;

FREQ:0
<bb 12>:
D.2330 = q r>> 7;
D.2331 = D.2330 ^ q;
D.2332 = q r>> 6;
D.2333 = D.2331 ^ D.2332;
D.2334 = q r>> 5;
D.2335 = D.2333 ^ D.2334;
D.2336 = q r>> 4;
D.2337 = D.2335 ^ D.2336;
D.2296 = D.2337 ^ 99;

FREQ:0
<bb 3>:
D.2296 = 99;
goto <bb 13> (<L10>);

FREQ:0
<bb 13>:
<L10>:
return D.2296;
```

Figure 3.1: An example of GCC CGFs

in power consumption and the decrease in operating frequency will beat the gains which is the decrease in the run time.

First, the new instructions are built as RTL models, then it's described in VHDL language and added to the processor's ALU. After that, the added instructions functionality is verified by performing behavioral simulation and the required clock cycles to finish the program is measured. Finally, the simulation results are analyzed and reported.

3.3 Adding New Instructions for AES Functions

The AES algorithm is divided to the next functions:

- KeyExpansion function: since this function will be called only once during the life cycle of the AES encryption program and its run time is short relative to the whole program run time, it's not efficient to add instructions for this function exactly. However, this function calls S-box function that will be discussed later and adding new instructions for S-box function leads to an enhancement in this function run time.

- AddRoundKey function: this function uses simple xor instruction only. It's a primitive instruction and cannot be reduced.

- SubBytes function: this function replaces every byte in a state with its S-box equivalent. As discussed before, S-box operation relies on finding the multiplicative inverse in Rijndael's GF which can be implemented in different ways. Firstly, we implemented "sbox_1" function that uses the brute search method and added new instructions for it. However, even though the added instructions reduced the required clock cycles to finish the function, its run time still too long relative to the other functions. So, we decided to shift to another method. "sbox_2" function uses the extended Euclidean algorithm and it gives more satisfying results and more reasonable run-time overhead.

- ShiftRows function: this function does many byte swap operations, and no new instruction can reduce its work at least in our processor architecture.

- MixColumns function: this function multiplies each column (sequential four bytes) in an AES state with a constant matrix in Rijndael's GF. We found that the added instructions for "sbox_1" function can be used for this function too.

3.3.1 Adding new instructions for sbox_1 function

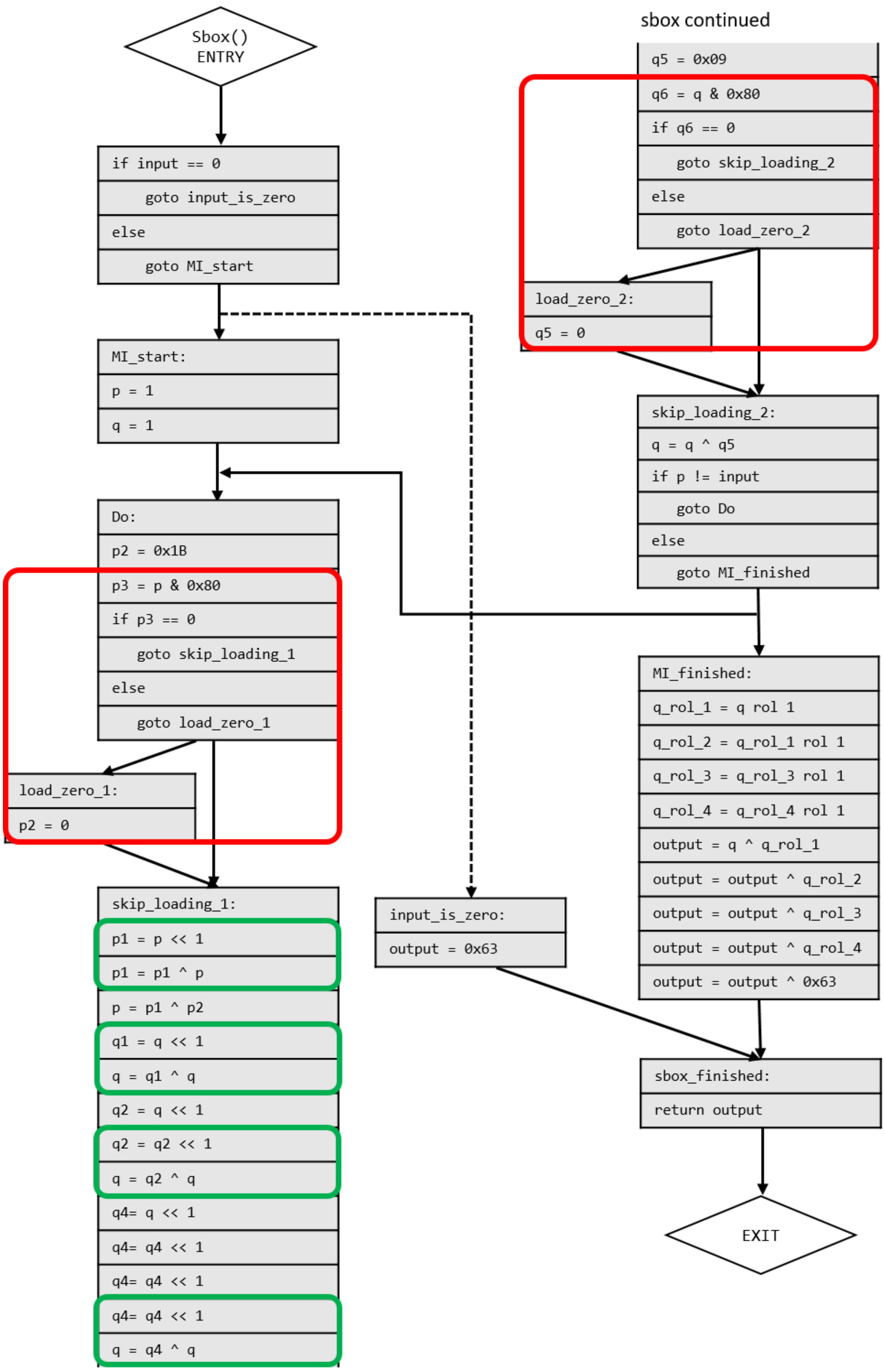
sbox_1 function takes an input "a" and returns its S-box output. Its C code is given in Code 3.1. This code relies on a property that says: for a number p and its multiplicative inverse q in GF, multiplying p with 3 and dividing q with 3 gives another multiplicative inverse pair.

Code 3.1: The C code of sbox_1 function

```
#define ROTL8(x,shift) ((uint8_t) ((x) << (shift)) | \
        ((x) >> (8 - (shift))))
uint8_t sbox_1(uint8_t a)
{
  /* 0 is a special case since it has no inverse */
  if (a == 0)
    return 0x63;

  uint8_t p = 1, q = 1;
  /* loop invariant: p * q == 1 in the Galois field */
  do {
    /* multiply p by 3 */
        p = p ^ (p << 1) ^ (p & 0x80 ? 0x1B : 0);
        /* divide q by 3 (equals multiplication by 0xf6) */
        q ^= q << 1;
        q ^= q << 2;
        q ^= q << 4;
        q ^= q & 0x80 ? 0x09 : 0;
  } while (p != a);
  /* compute the affine transformation */
  return q ^ ROTL8(q,1) ^ ROTL8(q,2) ^ ROTL8(q,3) ^ ROTL8(q,4) ^ 0x63;
}
```

We translated this code to assembly and drew a CFG for it. The CFG is given in Figure 3.2.

Then we searched where we can add a new instruction. We found that we can add one in place of the instructions that are in the red rectangular. These instructions are used to reset a register if the most significant bit of another register is 1. The new instruction ANDs a register (Rz) with the most significant bit of another register (Ry). It has the



```
if input == 0
    goto input_is_zero
else
    goto MI_start

MI_start:
p = 1
q = 1

Do:
p2 = 0x1B
p3 = p & 0x80
if p3 == 0
    goto skip_loading_1
else
    goto load_zero_1

load_zero_1:
p2 = 0

skip_loading_1:
p1 = p << 1
p1 = p1 ^ p
p = p1 ^ p2
q1 = q << 1
q = q1 ^ q
q2 = q << 1
q2 = q2 << 1
q = q2 ^ q
q4= q << 1
q4= q4 << 1
q4= q4 << 1
q4= q4 << 1
q = q4 ^ q

q5 = 0x09
q6 = q & 0x80
if q6 == 0
    goto skip_loading_2
else
    goto load_zero_2

load_zero_2:
q5 = 0

skip_loading_2:
q = q ^ q5
if p != input
    goto Do
else
    goto MI_finished

MI_finished:
q_rol_1 = q rol 1
q_rol_2 = q_rol_1 rol 1
q_rol_3 = q_rol_3 rol 1
q_rol_4 = q_rol_4 rol 1
output = q ^ q_rol_1
output = output ^ q_rol_2
output = output ^ q_rol_3
output = output ^ q_rol_4
output = output ^ 0x63

input_is_zero:
output = 0x63

sbox_finished:
return output
```

Figure 3.2: The CFG of sbox_1 function

same job of the old instructions. The new instruction name is “AMB” (And with most Significant Bit) and its assembly code is shown below.

AMB Rd, Ry, Rz

Some examples of the output of this instruction are given below

Ry = 0x47 (01000111),   Rz = 0x1B,   Rd = (00000000) & 0x1B = 0x00

Ry = 0x91 (10010001),   Rz = 0x09,   Rd = (11111111) & 0x09 = 0x09

The RTL model of this instruction in our ALU can be done by adding a multiplexer to the input of the already existed AND gate. This multiplexer selects the input according to the opcode of the executed instruction. The RTL model is shown in Figure 3.3.

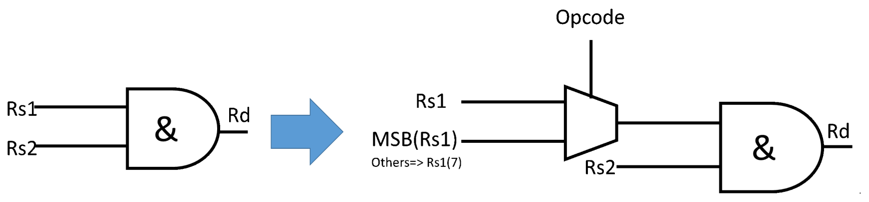


Figure 3.3: The RTL model of AMB instruction

The next instruction that we added to sbox_1 function is “SHLXOR”. It replaces the two instructions that are surrounded with green rectangular “SHL” and “XOR”. These two instructions come four times sequentially in sbox_1 assembly code and the new instruction combines its works. It shifts a register (Ry) to left one bit and then XORs it with another register (Rz). “SHLXOR” instruction Assembly code and two examples explain its work is given below

SHLXOR Rd,Ry,Rz

Ry = 0x40,   Rz = 0x01,   Rd = 0x81

Ry = 0x91,   Rz = 0xFF,   Rd = 0xDD

The RTL model of the new instruction can be presented by a new multiplexer on the input of the already existed XOR gate. This multiplexer selects XOR gate’s first input from a normal register or from the existed shift-left circuit output according to executed instruction opcode. The new RTL model is shown in Figure 3.4

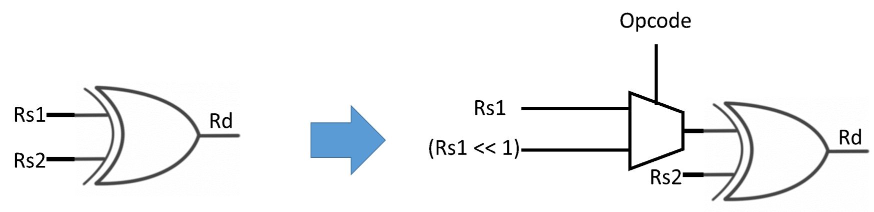


Figure 3.4: The RTL model of SHLXOR instruction

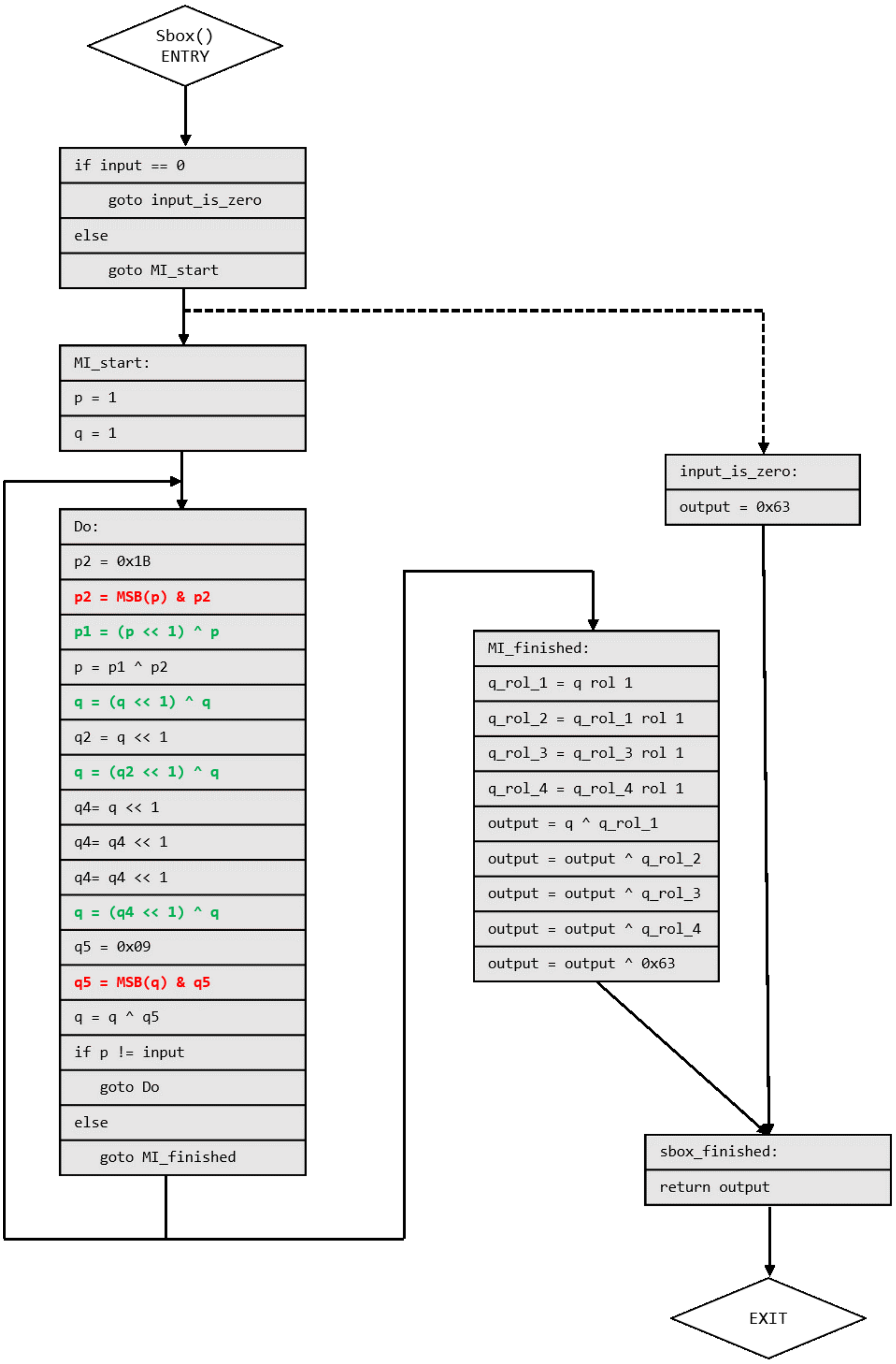
The CFG of sbox_1 function after adding AMB and SHLXOR instructions is shown in Figure 3.5.

Two programs are written to test sbox_1 function on the pipelined processor. In both programs, a 16-byte state has been fed to the function 10 times. The first program doesn't use the added instructions and the second one does. The code size and the required clock cycles to finish both programs are reported in Table 3.1.

Table 3.1: sbox_1 function simulation results

	Pipelined processor	Pipelined processor after adding AMB and SHLXOR
Clock cycles	861,793	510,183
Program size	61	44

The added instructions caused a 40% decrease in clock cycles and 27% decrease in code size.



```
if input == 0
    goto input_is_zero
else
    goto MI_start
```

```
MI_start:
p = 1
q = 1
```

```
Do:
p2 = 0x1B
p2 = MSB(p) & p2
p1 = (p << 1) ^ p
p = p1 ^ p2
q = (q << 1) ^ q
q2 = q << 1
q = (q2 << 1) ^ q
q4= q << 1
q4= q4 << 1
q4= q4 << 1
q = (q4 << 1) ^ q
q5 = 0x09
q5 = MSB(q) & q5
q = q ^ q5
if p != input
   goto Do
else
   goto MI_finished
```

```
input_is_zero:
output = 0x63
```

```
MI_finished:
q_rol_1 = q rol 1
q_rol_2 = q_rol_1 rol 1
q_rol_3 = q_rol_3 rol 1
q_rol_4 = q_rol_4 rol 1
output = q ^ q_rol_1
output = output ^ q_rol_2
output = output ^ q_rol_3
output = output ^ q_rol_4
output = output ^ 0x63
```

```
sbox_finished:
return output
```

Figure 3.5: The CFG of sbox_1 function after using AMB and SHLXOR instructions

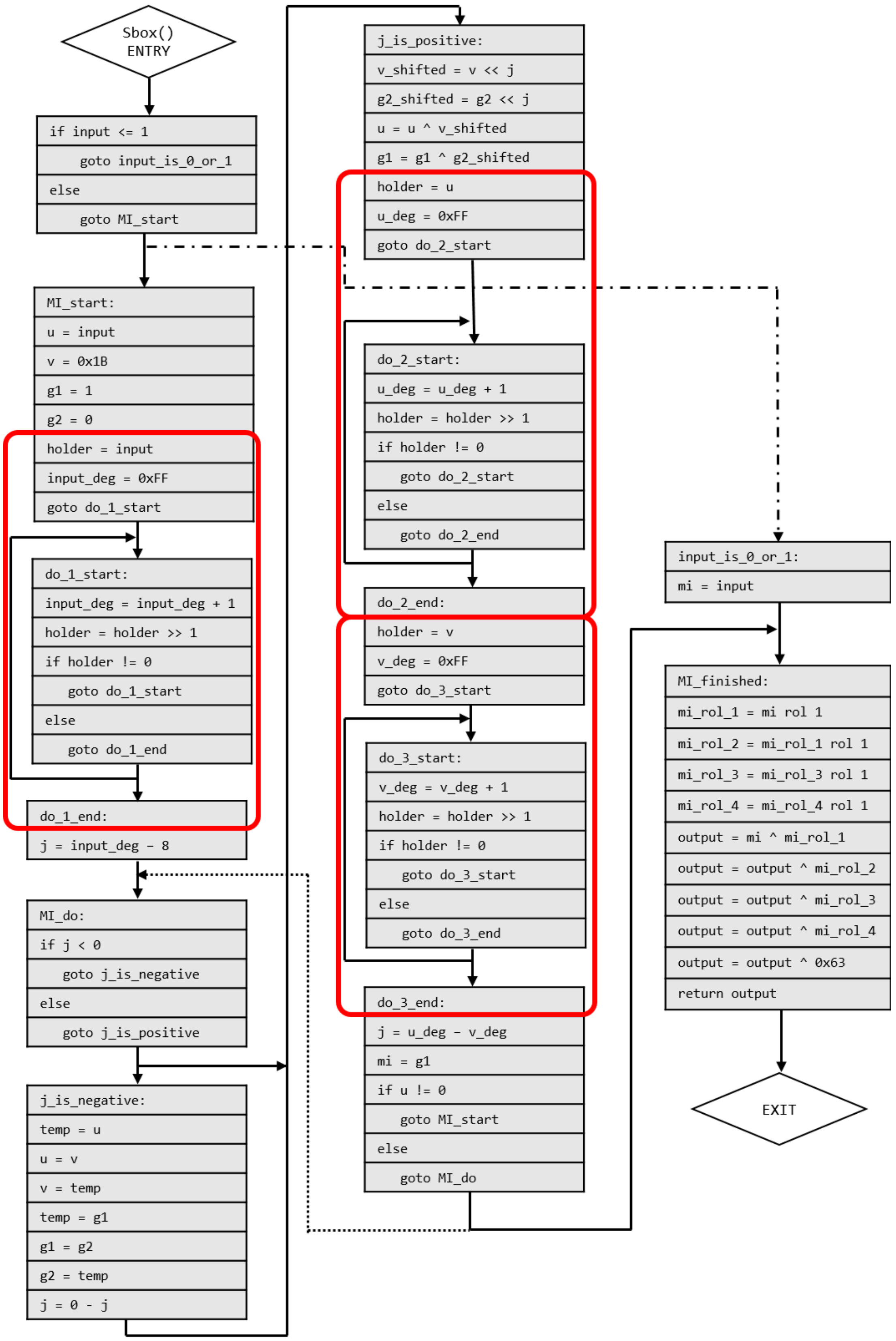
3.3.2 Adding new instructions for sbox_2 function

sbox_2 function is used to replace the inefficient sbox_1 function. In sbox_2 function the extended Euclidean algorithm is implemented to find the multiplicative inverse of the input in in Rijndael's GF. Its C code is given in code 3.2

Code 3.2: The C code of sbox_2 function

```
#define ROTL8(x,shift) ((uint8_t) ((x) << (shift)) | \
        ((x) >> (8 - (shift))))

unsigned char degree (unsigned char a)
{
  unsigned char res = 0xff;
  do
  {
    res++;
    a >>= 1;
  } while  (a != 0);
  return res;
}

unsigned char sbox_2 (unsigned char a)
{
  unsigned char temp;
  unsigned char u = a;
  unsigned char v = 0x1b;
  unsigned char g1 = 1;
  unsigned char g2 = 0;
  char j = degree(u) - 8;
  while (u != 1)
  {
    if (j < 0)
    {
      temp = u;
      u = v;
      v = temp;
      temp = g1;
      g1 = g2;
      g2 = temp;
      j = -j;
    }
    u ^= v << j;
    g1 ^= g2 <<j;
    j= degree(u) - degree(v);
  }
  return g1 ^ ROTL8(g1,1) ^ ROTL8(g1,2) ^ ROTL8(g1,3) ^ ROTL8(g1,4) ^ 0x63;
}
```

The code is translated to assembly and its CFG is given in Figure 3.6



```
if input <= 1
    goto input_is_0_or_1
else
    goto MI_start
```

```
MI_start:
u = input
v = 0x1B
g1 = 1
g2 = 0
holder = input
input_deg = 0xFF
goto do_1_start
```

```
do_1_start:
input_deg = input_deg + 1
holder = holder >> 1
if holder != 0
    goto do_1_start
else
    goto do_1_end
```

```
do_1_end:
j = input_deg - 8
```

```
MI_do:
if j < 0
    goto j_is_negative
else
    goto j_is_positive
```

```
j_is_negative:
temp = u
u = v
v = temp
temp = g1
g1 = g2
g2 = temp
j = 0 - j
```

```
j_is_positive:
v_shifted = v << j
g2_shifted = g2 << j
u = u ^ v_shifted
g1 = g1 ^ g2_shifted
holder = u
u_deg = 0xFF
goto do_2_start
```

```
do_2_start:
u_deg = u_deg + 1
holder = holder >> 1
if holder != 0
    goto do_2_start
else
    goto do_2_end
```

```
do_2_end:
holder = v
v_deg = 0xFF
goto do_3_start
```

```
do_3_start:
v_deg = v_deg + 1
holder = holder >> 1
if holder != 0
    goto do_3_start
else
    goto do_3_end
```

```
do_3_end:
j = u_deg - v_deg
mi = g1
if u != 0
    goto MI_start
else
    goto MI_do
```

```
input_is_0_or_1:
mi = input
```

```
MI_finished:
mi_rol_1 = mi rol 1
mi_rol_2 = mi_rol_1 rol 1
mi_rol_3 = mi_rol_3 rol 1
mi_rol_4 = mi_rol_4 rol 1
output = mi ^ mi_rol_1
output = output ^ mi_rol_2
output = output ^ mi_rol_3
output = output ^ mi_rol_4
output = output ^ 0x63
return output
```

Figure 3.6: The CFG of sbox_2 function

After investigating the CFG we found that we can add an instruction to find the degree of a polynomial in Rijndael's GF and use it instead of the code block that is surrounded in red rectangular. The new instruction "DEG" takes two operands only (Rd and Ry). It finds the degree of the source register (Ry) and stores it in the destination register (Rd). Its assembly code and three examples explain its work are given below.

DEG Rd, Ry

Ry = 0x0A, Rd = 0x03

Ry = 0xF7, Rd = 0x07

Ry = 0x01, Rd = 0x00

The RTL model of the new instruction can be described as 7 2-to-1 multiplexers connected in series where each multiplexer's zero input is connected to the output of the previous one and the first multiplexer zero input is connected to "0" (as 8-bit vector). The one input of the multiplexers is connected to constant vector which its value is the multiplexer order (1, 2, ... ,7). The selector pin of the (i)'th multiplexer is connected to the (i+1)'th bit of the source register Ry. The output of the last multiplexer is the result of DEG instruction. The RTL model of this instruction is given in Figure 3.7

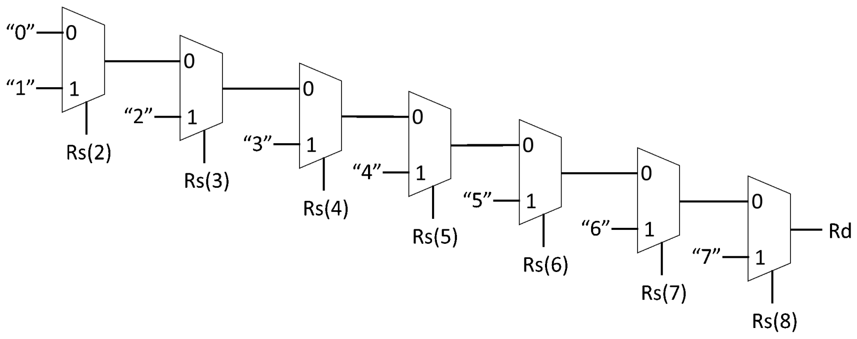


Figure 3.7: The RTL model of DEG instruction

The CFG of sbox_2 function after adding DEG instruction is shown in Figure 3.8.

Note: "v_shifted = v << j" and "g2_shifted = g2 << j" statements in the CFGs are not real instructions. They are actually implemented using a loop because our shift instructions shift only by one. The loop code is represented in this way just to make the

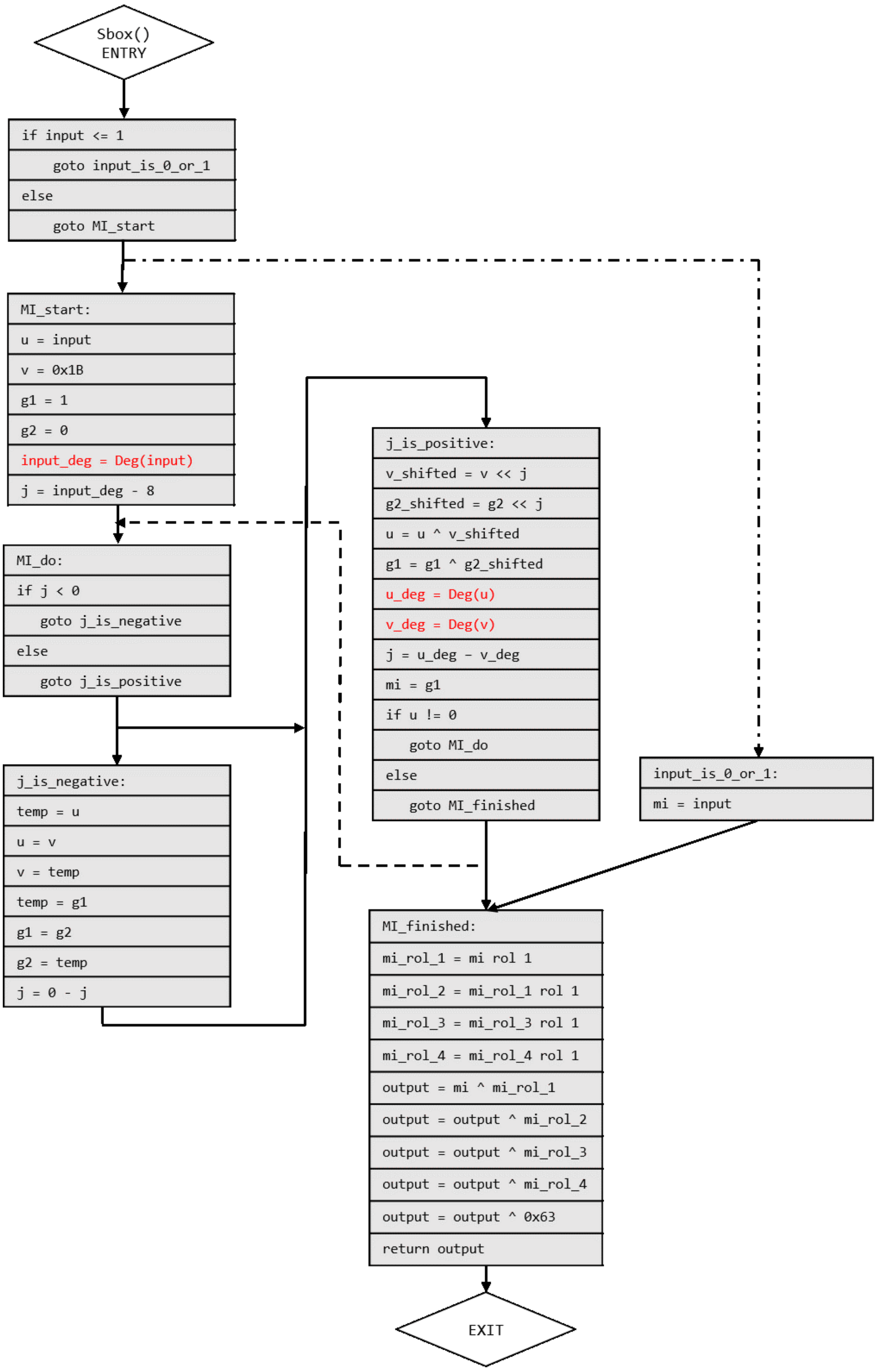


Figure 3.8: The CFG of sbox_2 function after using DEG instruction

CFG clearer. No new instruction is added for shifting by a variable number because the required hardware (a barrel shifter circuit) has a massive area and leads to large power consumption.

As in sbox_1 function, two programs are written to test sbox_2 function on the pipelined processor. In both programs, a 16-byte state has been fed to the function 10 times. The first program doesn't use "DEG" instruction and the second one does. The code size and the required clock cycles to finish both programs are reported in Table 3.2

Table 3.2: sbox_2 function simulation results

	Pipelined processor	Pipelined processor after adding DEG
Clock cycles	106,873	46,033
Program size	85	64

The added "DEG" instruction caused a 56% decrease in clock cycles and 24% decrease in code size.

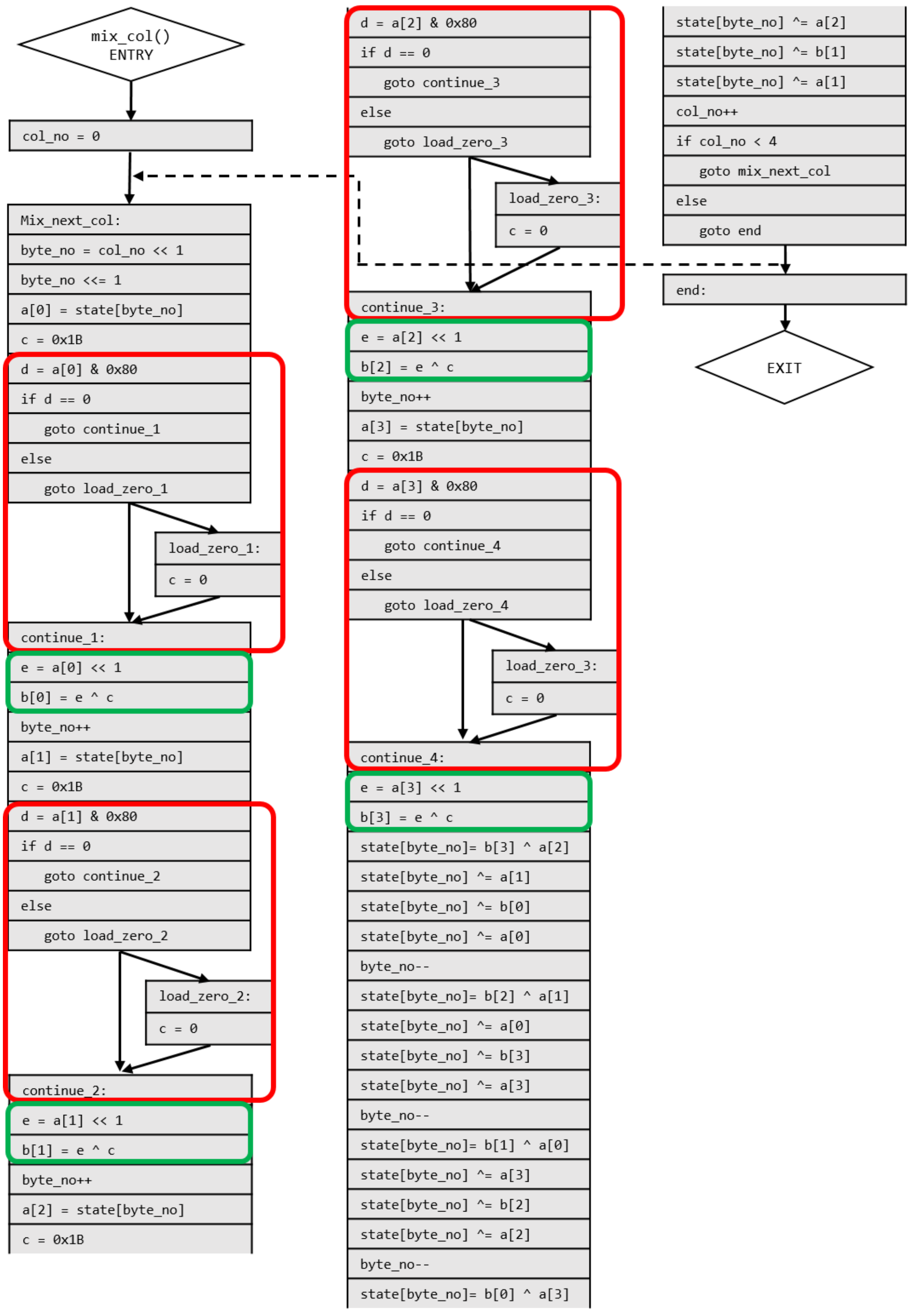
3.3.3 Adding new instructions for mix_col function

mix_col function multiplies each column of an input state with a constant matrix in Rijndael's GF. The C code of this function is given in Code 3.3

Code 3.3: The C code of mix_col function

```
void mix_col(uint8_t* state)
{
  // The array 'a' is simply a copy of a column from the input
  uint8_t a[4];
  //The array 'b' is each element of the array 'a' multiplied by 2
  //in Rijndael's GF
  uint8_t b[4];
  // a[n] ^ b[n] is element n multiplied by 3 in Rijndael's GF
  for (uint8_t columnNumber = 0; columnNumber<4; ++columnNumber)
  {
    uint8_t stateByteIndex = columnNumber*4;
    a[0] = state[stateByteIndex];
    b[0] = (a[0] << 1) ^ (a[0] & 0x80 ? 0x1B : 0);
    stateByteIndex++;

    a[1] = state[stateByteIndex];
    b[1] = (a[1] << 1) ^ (a[1] & 0x80 ? 0x1B : 0);
    stateByteIndex++;

    a[2] = state[stateByteIndex];
    b[2] = (a[2] << 1) ^ (a[2] & 0x80 ? 0x1B : 0);
    stateByteIndex++;

    a[3] = state[stateByteIndex];
    b[3] = (a[3] << 1) ^ (a[3] & 0x80 ? 0x1B : 0);

    //column_byte[3] = 2 * a3 + a2 + a1 + 3 * a0
    state[stateByteIndex] = b[3] ^ a[2] ^ a[1] ^ b[0] ^ a[0];
    stateByteIndex--;

    //column_byte[2] = 2 * a2 + a1 + a0 + 3 * a3
    state[stateByteIndex] = b[2] ^ a[1] ^ a[0] ^ b[3] ^ a[3];
    stateByteIndex--;

    //column_byte[1] = 2 * a1 + a0 + a3 + 3 * a2
    state[stateByteIndex] = b[1] ^ a[0] ^ a[3] ^ b[2] ^ a[2];
    stateByteIndex--;

    //column_byte[0] = 2 * a0 + a3 + a2 + 3 * a1
    state[stateByteIndex] = b[0] ^ a[3] ^ a[2] ^ b[1] ^ a[1];
    }
}
```

The code is converted to assembly and its CFG is given in Figure 3.9.



```
col_no = 0

Mix_next_col:
byte_no = col_no << 1
byte_no <<= 1
a[0] = state[byte_no]
c = 0x1B
d = a[0] & 0x80
if d == 0
    goto continue_1
else
    goto load_zero_1
load_zero_1:
c = 0
continue_1:
e = a[0] << 1
b[0] = e ^ c
byte_no++
a[1] = state[byte_no]
c = 0x1B
d = a[1] & 0x80
if d == 0
    goto continue_2
else
    goto load_zero_2
load_zero_2:
c = 0
continue_2:
e = a[1] << 1
b[1] = e ^ c
byte_no++
a[2] = state[byte_no]
c = 0x1B

d = a[2] & 0x80
if d == 0
    goto continue_3
else
    goto load_zero_3
load_zero_3:
c = 0
continue_3:
e = a[2] << 1
b[2] = e ^ c
byte_no++
a[3] = state[byte_no]
c = 0x1B
d = a[3] & 0x80
if d == 0
    goto continue_4
else
    goto load_zero_4
load_zero_3:
c = 0
continue_4:
e = a[3] << 1
b[3] = e ^ c
state[byte_no]= b[3] ^ a[2]
state[byte_no] ^= a[1]
state[byte_no] ^= b[0]
state[byte_no] ^= a[0]
byte_no--
state[byte_no]= b[2] ^ a[1]
state[byte_no] ^= a[0]
state[byte_no] ^= b[3]
state[byte_no] ^= a[3]
byte_no--
state[byte_no]= b[1] ^ a[0]
state[byte_no] ^= a[3]
state[byte_no] ^= b[2]
state[byte_no] ^= a[2]
byte_no--
state[byte_no]= b[0] ^ a[3]

state[byte_no] ^= a[2]
state[byte_no] ^= b[1]
state[byte_no] ^= a[1]
col_no++
if col_no < 4
    goto mix_next_col
else
    goto end

end:
```

Figure 3.9: The CFG of mix_col function

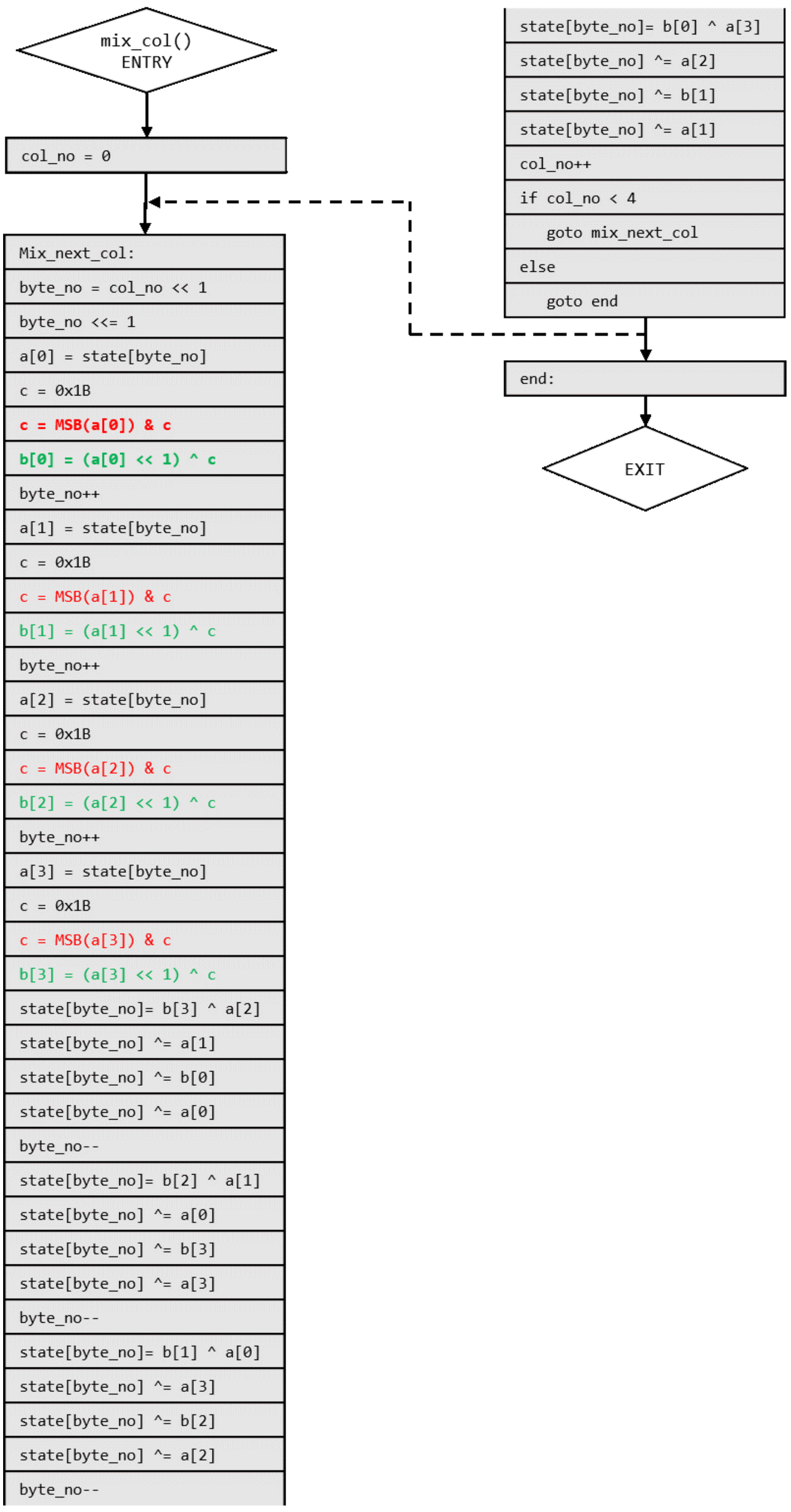
We found that added instructions for the replaced function sbox_1 can be used in mix_col function too, and so no new instructions are needed to be added. The new CFG for mix_col function after using “SHLXOR” and “AMB” instructions is shown in Figure 3.10.

Again, two programs are written to test mix_col function on the pipelined processor. In both programs, a 16-byte state has been fed to the function 10 times. The first program doesn’t use “SHLXOR” and “AMB” instructions and the second one does. The code size and the required clock cycles to finish both programs are reported in Table 3.3.

Table 3.3: mix_col function simulation results

	Pipelined processor	Pipelined processor after using AMB and SHLXOR
Clock cycles	4,309	2,879
Program size	87	56

Using “AMB” and “SHLXOR” instructions caused a 33% decrease in clock cycles and 35% decrease in code size.



```
col_no = 0

Mix_next_col:
byte_no = col_no << 1
byte_no <<= 1
a[0] = state[byte_no]
c = 0x1B
c = MSB(a[0]) & c
b[0] = (a[0] << 1) ^ c
byte_no++
a[1] = state[byte_no]
c = 0x1B
c = MSB(a[1]) & c
b[1] = (a[1] << 1) ^ c
byte_no++
a[2] = state[byte_no]
c = 0x1B
c = MSB(a[2]) & c
b[2] = (a[2] << 1) ^ c
byte_no++
a[3] = state[byte_no]
c = 0x1B
c = MSB(a[3]) & c
b[3] = (a[3] << 1) ^ c
state[byte_no]= b[3] ^ a[2]
state[byte_no] ^= a[1]
state[byte_no] ^= b[0]
state[byte_no] ^= a[0]
byte_no--
state[byte_no]= b[2] ^ a[1]
state[byte_no] ^= a[0]
state[byte_no] ^= b[3]
state[byte_no] ^= a[3]
byte_no--
state[byte_no]= b[1] ^ a[0]
state[byte_no] ^= a[3]
state[byte_no] ^= b[2]
state[byte_no] ^= a[2]
byte_no--
state[byte_no]= b[0] ^ a[3]
state[byte_no] ^= a[2]
state[byte_no] ^= b[1]
state[byte_no] ^= a[1]
col_no++
if col_no < 4
   goto mix_next_col
else
   goto end

end:
```

Figure 3.10: The CFG of mix_col function after using AMB and SHLXOR instructions

3.4 The Extended Instruction Set

As a result of this study, the standard instruction set of the pipelined processor is extended with three instructions. The new instructions' codes are given in Table 3.4.

Table 3.4: The extended instruction set of the designed ASIP

Instruction	Function	Code
SHLXOR Rd,Ry,Rz	Rd = (Ry <<1) ^Rz	11000 dddd 0zzzz yyyy
AMB Rd,Ry,Rz	Rd = MSB(Ry) & Rz	11001 dddd 0zzzz yyyy
DEG Rd,Ry	Rd = DEG (Ry)	11010 dddd 00000 yyyy

The big benefit that came from the extended instructions is reducing the run time for some AES functions in a significant way. Table 3.5 shows a comparison between the designed GPP and the designed ASIP in the number of the required clock cycles to finish many programs.

Table 3.5: Performance simulation results of the designed GPP and the designed ASIP for AES functions

Program	GPP	ASIP
sbox_1 for 10 rounds	861,793	510,183
sbox_2 for 10 rounds	106,873	46,033
mix_col for 10 rounds	4,309	2,879
key_expansion	27,933	14,036
AES round	12,815	6,124
AES total	159,808	76,829

3.5 Simulations Results

In order to compare our ASIP (the pipelined processor with the extended instructions) with our GPP (the pipelined processor without the extended instructions) and get the energy saving outcome the following steps are performed.

First, the AES algorithm was implemented with the standard instructions and was run on the designed GPP. The number of the used FPGA slices and the number of clock cycles that the algorithm takes to complete are reported. The maximum clock frequency and the dynamic on-chip power are already obtained in the previous chapter. The latency, the throughput and the energy consumption are calculated.

Next, the AES algorithm was implemented with the extended instructions and was run on the designed ASIP. After that, the maximum operating frequency, the dynamic on-chip power, the number of the used FPGA slices and the required clock cycles to finish the algorithm code are obtained. In the designed ASIP, the maximum operating frequency was dropped down by 13%, this can be explained as a result of the new instructions' hardware that extended the critical path. Also, the dynamic on-chip power increased 14%, this is because of the additional FPGA slices that are used by the hardware of the new instructions.

Finally, the latency, the throughput and the energy consumption for the designed ASIP are calculated. As a result of decreasing the AES algorithm run time on the ASIP, the latency is decreased and the throughput is increased significantly. The energy consumption of the AES is also decreased 37% although the dynamic on-chip power is increased, because the improvement in latency overcame the downgrade in dynamic power outcome.

Table 3.6: Comparison of the designed GPP and the designed ASIP simulation results

	GPP	ASIP	Ratio of percentage change
Maximum frequency	155 MHz	135 MHz	-13%
Dynamic on-chip power	0.007 W	0.008 W	+14%
Area (number of slices)	70	74	+06%
AES clock cycles	159,808	76,829	-52%
Latency	1031 uS	569 uS	-45%
Throughput	121.24 Kbit/s	219.64 Kbit/s	+81%
Energy	7.22 uW/S	4.55 uW/S	-37%

3.6 Comparing the Proposed Work with Previous Works

AES became a study subject for many researchers and hardware designers due to its importance and its wide usage in many fields. A lot of work is done in designing high performance low-power ASICs for AES [13] [14] [15] [16] [17] [18] [19] [20].

On the other hand, a fewer work is done in making ASIPs or extending an instruction set for AES. In Onur Sahin et al work [21] 6 new complex instructions are added to the 32-bit LEON 2 processor. As reported, the added instructions sped up AES execution 3.12 times. However, no further information is given about the variation in energy consumption or operating frequency.

In Renhai Chen et al work [22] a GPP design is proposed and its instruction set is extended with 4 specific instructions for AES. The presented ASIP achieved 46.5% performance improvement compared to ARM ISA. Although the added instructions' hardware is simple, it caused a 14% increase in the used resources.

Tim Good et al represented a very small 8-bit ASIP for AES on FPGA in their work [23]. As the small area was the main priority of the project, the instruction set of the ASIP is so optimized such the processor isn't capable of doing any work except AES operations.

Our work is based on a novel and genuine processor design not on an open source project. This makes us fully knowing the design details. Also, our added instructions were selected to be simple not complex, complex instructions like one instruction for the whole S-box function or MixColumns function requires more resources on the

FPGA and that causes to decrease the operating frequency and to increase the energy consumption highly. Our instruction set isn't optimized for AES only because we wanted the processor to be used for different applications beside AES encryption.

4. CONCLUSION

In this thesis, a low-power general purpose processor design is presented. Then the processor design is improved by extending the instruction set with instructions for the Advanced Encryption Standard (AES).

First, a simple general-purpose processor was designed by determining its standard instruction set then implementing its data path and control unit using VHDL in Vivado Environment. Then the processor design was improved by pipelining it. Pipeline hazards were avoided without complicating the processor structure. Finally, the simulation results of both designs were compiled with Xilinx PicoBlaze processor. Both designs consumed less power than PicoBlaze processor, and the pipelined processor's maximum frequency was higher than PicoBlaze processor's maximum frequency. Also, the pipelined design finished test programs with clock cycles less than the non-pipelined design and consumed less energy.

After that, the AES algorithm was implemented in C then translated to assembly code. CFGs were drawn for the complex functions of the algorithm and then examined. New candidate instructions that solves software problems faster or combines sequential and related instructions were built as RTL models, then described in VHDL language and added to the processor's ALU. Next, the added instructions functionality was verified by performing behavioral simulation and the required clock cycles to finish test programs were measured.

Finally, the designed GPP and ASIP were compared. It was found that ASIP consumes less energy than GPP by 37% although its dynamic on-chip power is higher, because the improvement in its latency overcame the downgrade in dynamic power outcome.

REFERENCES

[1] **Glokler, T. and Meyr, H.**, 2004. Design of Energy-Efficient Application-Specific Instruction Set Processors (ASIPs), Kluwer Academic Publishers.

[2] **National Institute of Standards and Technology**, 1999. FIPS 46-3: Data Encryption Standard, `https://csrc.nist.gov/csrc/media/publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf`.

[3] **National Institute of Standards and Technology**, 2001. FIPS 197: Advanced Encryption Standard, `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf`.

[4] **Hennessy, J. and Patterson, D.A.**, 2017. Computer Architecture: A Quantitative Approach, Morgan Kaufmann, 6th edition.

[5] **Mano, M.M.R. and Ciletti, M.D.**, 2017. Digital Design: With an Introduction to the Verilog HDL, VHDL, and SystemVerilog, Pearson, 6th edition.

[6] **Churiwala, S.**, editor, 2017. Designing with Xilinx® FPGAs: Using Vivado, Springer, 1st edition.

[7] **Chadha, R. and Bhasker, J.**, 2013. An ASIC Low Power Primer: Analysis, Techniques and Specification, Springer.

[8] **Xilinx**, PicoBlaze 8-bit Microcontroller, `https://www.xilinx.com/products/intellectual-property/picoblaze.html`.

[9] **Wikipedia**, 2020, Finite field arithmetic — Wikipedia, The Free Encyclopedia, `https://en.wikipedia.org/wiki/Finite_field_arithmetic`, [Online; accessed 21-March-2020].

[10] **Wikipedia**, 2020, Extended Euclidean algorithm — Wikipedia, The Free Encyclopedia, `https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm`, [Online; accessed 21-March-2020].

[11] **Allen, F.E.**, 1970. Control Flow Analysis, SIGPLAN Notices.

[12] Graphviz - Graph Visualization Software, `https://www.graphviz.org/`.

[13] **Hamalainen, P.**, **Alho, T.**, **Hannikainen, M. and Hamalainen, T.D.**, 2006. Design and Implementation of Low-Area and Low-Power AES Encryption Hardware Core, 9th EUROMICRO Conference on Digital System Design (DSD'06), pp.577–583.

[14] **Hodjat, A. and Verbauwhede, I.**, 2006. Area-throughput trade-offs for fully pipelined 30 to 70 Gbits/s AES processors, *IEEE Transactions on Computers*, **55(4)**, 366–372.

[15] **Rouvroy, G.**, **Standaert, F..**, **Quisquater, J.. and Legat, J..**, 2004. Compact and efficient encryption/decryption module for FPGA implementation of the AES Rijndael very well suited for small embedded applications, International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004., volume 2, pp.583–587 Vol.2.

[16] **Mozaffari-Kermani, M. and Reyhani-Masoleh, A.**, 2012. Efficient and High-Performance Parallel Hardware Architectures for the AES-GCM, *IEEE Transactions on Computers*, **61(8)**, 1165–1178.

[17] **Good, T. and Benaissa, M.**, 2010. 692-nW Advanced Encryption Standard (AES) on a 0.13-μm CMOS, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **18(12)**, 1753–1757.

[18] **Tsung-Fu Lin**, **Chih-Pin Su**, **Chih-Tsun Huang and Cheng-Wen Wu**, 2002. A high-throughput low-cost AES cipher chip, Proceedings. IEEE Asia-Pacific Conference on ASIC,, pp.85–88.

[19] **Sever, R.**, **Ismailoglu, A.N.**, **Tekmen, Y.C. and Askar, M.**, 2004. A high speed ASIC implementation of the Rijndael algorithm, 2004 IEEE International Symposium on Circuits and Systems (IEEE Cat. No.04CH37512), volume 2, pp.II–541.

[20] **Huang, Y.**, **Lin, Y.**, **Hung, K. and Lin, K.**, 2006. Efficient Implementation of AES IP, APCCAS 2006 - 2006 IEEE Asia Pacific Conference on Circuits and Systems, pp.1418–1421.

[21] **Şahin, O. and Örs Yalçın, B.**, 2012. Kriptoloji Uygulamalarına Özel Bir İşlemcinin Tasarlanarak FPGA Üzerinde Gerçeklenmesi, GÖMSİS 2012 Gömülü Sistemler ve Uygulamaları Sempozyumu.

[22] **Chen, R.**, **Jia, Z.**, **Li, Y.**, **Hui, X. and Li, X.**, 2011. The application specific instruction processor for AES, **4**.

[23] **Good, T. and Benaissa, M.**, 2006. Very small FPGA application-specific instruction processor for AES, *Circuits and Systems I: Regular Papers, IEEE Transactions on*, **53**, 1477 – 1486.

CURRICULUM VITAE

Name Surename: Muhammed ŞAİROĞLU

Place and Date of Birth: Homs - Syria, 1994

E-Mail: ammarshaar94@gmail.com

Education:

- **B.Sc.:** Istanbul University
- **M.Sc.:** Istanbul Technical University

Professional Experience: 2016 - Present : Panasonic Life Solutions, R&D Department, Embedded Software Engineer

Publications, Presentations and Patents on This Thesis

▪ Mohammad Ammar Alshaar and Berna Örs, 2019 : Special Purpose Processor Design for IoT Applications and Implementation on an FPGA
İşlemci Tasarımı Çalıştayı 2019, September 19, 2019 Istanbul, Turkey.