

Faculty of Engineering

Design Methodology and Analysis of Hash Function based-on Field Programmable Gate Array for Hash-based Message Authentication Code Application

Shamsiah binti Suhaili

Doctor of Philosophy
2025

Design Methodology and Analysis of Hash Function based-on Field Programmable Gate Array for Hash-based Message Authentication Code Application

Shamsiah binti Suhaili

A thesis submitted

In fulfillment of the requirements for the degree of Doctor of Philosophy

(Electronic Engineering)

Faculty of Engineering
UNIVERSITI MALAYSIA SARAWAK
2025

DECLARATION

I declare that the work in this thesis was carried out in accordance with the regulations of Universiti Malaysia Sarawak. Except where due acknowledgements have been made, the work is that of the author alone. The thesis has not been accepted for any degree and is not concurrently submitted in candidature of any other degree.



……

Signature

Name: Shamsiah binti Suhaili

Matric No.: 19010121

Faculty of Engineering

Universiti Malaysia Sarawak

Date: 27 August 2025

ACKNOWLEDGEMENT

I am grateful to my supervisor, Assoc. Prof. Dr Norhuzaimin bin Julai, of Universiti Malaysia Sarawak (UNIMAS), for allowing me to conduct research under his guidance and for offering me guidance on research, writing, and presentation. His assistance was indispensable for the timely completion of this thesis.

I would like to extend my gratitude to all those who have made a direct or indirect contribution to this thesis. I am extremely grateful to all of my peers who have provided me with support in some capacity throughout my time at UNIMAS. I am grateful to all professors and lecturers for the knowledge they have shared with me regarding digital designs. Their expertise in this field really encourages and inspires me to continue my research.

Lastly, I would like to extend my sincere gratitude to my family for their unwavering support, encouragement, patience, and love during the entirety of my academic journey. My sincere gratitude to the Centre for Graduate Studies, for the advice and support given during my period of study in UNIMAS. I would also like to express my gratitude to the management and staff of UNIMAS for their assistance and cooperation in enabling me to complete my study in Sarawak. I am grateful to each of you.

ABSTRACT

Security has grown in importance as a study issue in recent years. Many cryptographic techniques have been created to raise the performance of different information-protection systems. One of the categories of cryptography is a hash function. In this research, the field-programmable gate array (FPGA) device from Altera is used because of its higher performance and smaller area implementation. For this purpose, cryptography hash functions are selected as a design target. These algorithms have been studied to improve the speed, area implementation, and power consumption. However, some trade-offs exist among higher performance, area implementation, and low-power design. Cryptographic algorithms are one of the most important aspects of the hardware implementation of embedded system design. They offer methods and systems for executing secure and authenticated transactions via the network. FPGAs are chosen because they can provide high-speed design, are simple to debug, are inexpensive, and have a rapid time to market. The main objective of this research is to formulate unfolding transformation factor 2 and factor 4 methods into hash function design. Additionally, low-power design techniques using state encoding in hash function design also need to be considered. All these design methodologies aim to confirm the effect of HDL coding style on the performance of hardware algorithms implemented on FPGA. This research focuses on unkeyed algorithm hash functions, such as MD5, SHA-1, RIPEMD-160, and SHA-256, to evaluate and compare the design's performance, area implementation, maximum clock frequency, and power consumption. In this research, an FPGA device named Arria II GX from Altera is used because of its higher performance and smaller area implementation compared with others. Pipelining and unfolding methods are applied to obtain high maximum frequency and high throughput of hash functions. The proposed SHA-1 designs, implemented on the Arria II GX EP2AGX45DF29C4,

demonstrates an area of 548 combinational ALUTs and 907 registers. This is a significant reduction on the Stratix II GX. This indicates a 98.38% reduction in ALUTs and a 97.17% reduction in registers, highlighting the significant area efficiency of the proposed design. The most efficient design is the pipelined RIPEMD-160 on Arria II GX, which achieves a ratio of 3.5477 Mbps/ALUT, far greater than all other designs. In addition, the unfolding-based designs (factor 2 and factor 4) demonstrate competitive ratios of 0.5109 Mbps/ALUT and 0.5203 Mbps/ALUT, respectively, reflecting effective trade-offs between performance and resource utilisation. The proposed SHA-256 design with an unfolding factor of 4 had the greatest throughput of 4,196.30 Mbps, indicating a considerable improvement over earlier SHA-256 architecture. The proposed design demonstrates a 294.5% enhancement in throughput and the proposed architecture exhibits a 165.5% enhancement in throughput. A comparative analysis of HMAC-SHA-256 designs implemented on different FPGA platforms. With an area utilisation of 3,953 LUTs and 2,714 registers, the proposed HMAC-SHA256 design, when implemented on Altera Arria II GX with Quartus II 15.0, reaches a maximum frequency of 195.16 MHz.. The novelty and originality of this study lie in the combination of different methodologies that can improve the performance of hash function algorithms and integration of HMAC-SHA-256 designs. Besides, Gray encoding and clock gating can reduce the dynamic power dissipation caused by signal toggling. Furthermore, a combination of unfolding and pipelining can also increase the throughput of hash function algorithms.

Keywords: Digital design, FPGA, SHA-256, Throughput, Unfolding technique

Kaedah Reka Bentuk dan Analisis Fungsi Hash Berasaskan Litar Bersepadu yang Boleh diprogramkan semula untuk Aplikasi Kod Pengesahan Mesej Berasaskan Hash

ABSTRAK

Keselamatan telah menjadi isu penyelidikan yang semakin penting dalam beberapa tahun kebelakangan ini. Pelbagai teknik kriptografi telah dibangunkan untuk meningkatkan prestasi sistem perlindungan maklumat. Salah satu kategori dalam bidang kriptografi ialah fungsi hash. Dalam kajian ini, peranti litar bersepadu yang boleh diprogramkan semula (FPGA) keluaran Altera telah digunakan kerana keupayaannya memberikan prestasi yang tinggi dan penggunaan kawasan perkakasan yang lebih kecil. Oleh itu, fungsi hash kriptografi telah dipilih sebagai sasaran reka bentuk. Algoritma ini dikaji dari aspek kelajuan pemprosesan, pelaksanaan kawasan, dan penggunaan kuasa. Walau bagaimanapun, terdapat kompromi antara prestasi tinggi, saiz pelaksanaan, dan reka bentuk kuasa rendah yang perlu diimbangi. Algoritma kriptografi merupakan komponen penting dalam pelaksanaan perkakasan bagi reka bentuk sistem terbenam kerana ia menyediakan kaedah untuk menjamin transaksi yang selamat dan disahkan melalui rangkaian. FPGA dipilih kerana ia mampu memberikan reka bentuk berkelajuan tinggi, mudah untuk dinyahralat, berkos rendah, dan mempercepatkan masa ke pasaran. Objektif utama kajian ini adalah untuk merumuskan kaedah transformasi unfolding dengan faktor 2 dan faktor 4 dalam reka bentuk fungsi hash. Selain itu, teknik reka bentuk kuasa rendah menggunakan kaedah pengekodan keadaan (state encoding) dalam fungsi hash juga turut diambil kira. Kesemua metodologi ini bertujuan untuk mengesahkan kesan gaya pengekodan bahasa penerangan perkakasan (HDL) terhadap prestasi algoritma perkakasan yang dilaksanakan pada FPGA. Kajian ini memfokuskan kepada fungsi hash tanpa kunci seperti MD5, SHA-1, RIPEMD-160, dan SHA-256, bagi tujuan penilaian dan perbandingan

prestasi reka bentuk dari segi pelaksanaan kawasan, frekuensi maksimum, dan penggunaan kuasa. FPGA Arria II GX daripada Altera telah digunakan dalam kajian ini kerana kelebihannya dari segi prestasi dan kecekapan kawasan berbanding platform FPGA yang lain. Kaedah pengpaipan dan penyingkapan telah diaplikasikan untuk memperoleh frekuensi maksimum dan kadar data yang tinggi. Reka bentuk SHA-1 yang dicadangkan, yang dilaksanakan pada peranti Arria II GX EP2AGX45DF29C4, menunjukkan penggunaan kawasan sebanyak 548 ALUT gabungan dan 907 daftar (register), iaitu pengurangan ketara berbanding pelaksanaan pada Stratix II GX iaitu pengurangan sebanyak 98.38% dalam penggunaan ALUT dan 97.17% dalam daftar, sekali gus menonjolkan kecekapan kawasan yang tinggi. Reka bentuk paling efisien adalah reka bentuk RIPEMD-160 berasaskan pipelining di atas Arria II GX, dengan nisbah prestasi kepada kawasan sebanyak 3.5477 Mbps/ALUT, yang jauh lebih tinggi berbanding reka bentuk lain. Selain itu, reka bentuk berasaskan kaedah unfolding dengan faktor 2 dan faktor 4 menunjukkan nisbah yang kompetitif iaitu masing-masing 0.5109 Mbps/ALUT dan 0.5203 Mbps/ALUT, sekaligus mencerminkan keseimbangan yang efektif antara prestasi dan penggunaan sumber. Reka bentuk SHA-256 yang dicadangkan menggunakan faktor unfolding 4 menunjukkan kadar throughput tertinggi iaitu 4,196.30 Mbps, menandakan peningkatan ketara berbanding seni bina SHA-256 sebelum ini. Reka bentuk ini menunjukkan peningkatan throughput sebanyak 294.5%, manakala seni bina yang dicadangkan pula menunjukkan peningkatan sebanyak 165.5%. Satu analisis perbandingan telah dijalankan ke atas reka bentuk HMAC-SHA-256 yang dilaksanakan di atas pelbagai platform FPGA. Dengan penggunaan kawasan sebanyak 3,953 LUT dan 2,714 daftar, reka bentuk HMAC-SHA256 yang dicadangkan – apabila dilaksanakan pada Altera Arria II GX menggunakan perisian Quartus II versi 15.0 yang mampu mencapai frekuensi maksimum

195.16 MHz. Keaslian dan kebaharuan kajian ini terletak pada gabungan pelbagai metodologi yang mampu meningkatkan prestasi algoritma fungsi hash dan integrasi reka bentuk HMAC-SHA-256. Selain itu, penggunaan pengekodan Gray dan penyekatan jam dapat mengurangkan kehilangan kuasa dinamik akibat pertukaran isyarat. Tambahan pula, gabungan kaedah unfolding dan pipelining dapat meningkatkan throughput algoritma fungsi hash secara signifikan.

Kata kunci: *Reka bentuk digital, FPGA, SHA-256, Truput, Teknik Unfolding*

TABLE OF CONTENTS

		Page
DECLARATION		i
ACKNOWLEDGEMENT		ii
ABSTRACT		iii
ABSTRAK		v
TABLE OF CONTENTS		viii
LIST OF TABLES		xiii
LIST OF FIGURES		xvii
LIST OF ABBREVIATIONS		xxiii
CHAPTER 1: INTRODUCTION		1
1.1	Research Background	1
1.2	Problem Statement	7
1.3	Motivation and Research Gap	8
1.4	Research Objective	11
1.5	Hypothesis	12
1.6	Research Scope	13
1.7	Significance of the Study	15
1.8	Thesis Organisation	17

1.9 Thesis Contribution 18

CHAPTER 2: LITERATURE REVIEW 19

2.1 Introduction 19

2.1.1 Overview of Hash Functions in Cryptography 20

2.1.2 Importance of Hash Function in HMAC Applications 21

2.2 Overview of Common Hash Functions 23

2.2.1 MD5 Design 24

2.2.2 SHA-1 Design 26

2.2.3 RIPEMD-160 Design 27

2.2.4 SHA-256 Design 28

2.3 Design Methodologies of Hash Function 30

2.3.1 Iterative 30

2.3.2 Unrolling and Pipelining 31

2.3.3 Unfolding 31

2.3.4 Low-power Design Approach 32

2.4 HMAC Overview and Comparative Analysis of Hash Function 34

2.4.1 Properties of Cryptographic Hash Functions 35

2.4.2 Comparative Analysis of Hash Functions 38

2.4.3 HMAC Design 62

2.5 Summary 67

CHAPTER 3: METHODOLOGY 69

3.1 Digital Design Flow 69

3.2 Design Overview of Hash Function and HMAC Algorithm 73

3.3 Dataflow and Behavioural Modelling 78

3.3.1 Dataflow Modelling 78

3.3.2 Behavioural Modelling 78

3.4 Hash Function and HMAC Algorithm 79

3.4.1 MD5 Algorithm 80

3.4.2 SHA-1 Algorithm 84

3.4.3 RIPEMD-160 Algorithm 88

3.4.4 SHA-256 Algorithm 93

3.4.5 HMAC Algorithm 98

3.5 Proposed Design Methodology 100

3.5.1 Unfolding Transformation Design Technique 103

3.5.2 Low Power Design Technique 105

3.6 Modification to Hash Function and HMAC Design 107

3.6.1 MD5 Design 108

3.6.2 SHA-1 Design 119

3.6.3 RIPEMD-160 Design 129

3.6.4 SHA-256 Design 147

3.6.5 HMAC Design 159

3.7 Summary 167

CHAPTER 4: RESULTS AND DISCUSSION 169

4.1 Introduction 169

4.2 Results and Discussion of MD5 Design Architecture 172

4.2.1 MD5 Iterative Design 174

4.2.2 MD5 Unfolding and Pipelining Design 190

4.3 Results and Discussion of SHA-1 Design Architecture 197

4.3.1 SHA-1 Iterative Design 199

4.3.2 SHA-1 Unfolding and Pipelining Design 205

4.4 Results and Discussion of RIPEMD-160 Design Architecture 229

4.4.1 RIPEMD-160 Iterative and Pipelining Design 231

4.5 Results and Discussion of SHA-256 Design Architecture 247

4.5.1 SHA-256 Design Architecture 248

4.6 Results and Discussion of HMAC Design Architecture 261

4.6.1 Results of HMAC-SHA1 Design 262

4.6.2 Results of HMAC-SHA256 Design 270

4.6.3 Discussion of HMAC-SHA1 Design 272

4.6.4 Discussion of HMAC-SHA-256 Design 275

4.7 Summary 278

CHAPTER 5: CONCLUSION 280

5.1 Summary of the Research 280

5.2 Future Work 286

REFERENCES 289

APPENDICES 310

Appendix A 310

Appendix B 311

Appendix C 313

LIST OF TABLES

		Page
Table 2.1	Compilation of Hash Functions in HMAC Applications	22
Table 2.2	Compilation of MD5 Hash Function in Different Applications	25
Table 2.3	Compilation of SHA-1 Hash Function in Different Applications	26
Table 2.4	Compilation of RIPEMD-160 Hash Function in Different Applications	28
Table 2.5	Compilation of SHA-256 Hash Function in Different Applications	29
Table 2.6	State Encoding	33
Table 2.7	Compilation of MD5 Hash Function Design	43
Table 2.8	Compilation of SHA-1 Hash Function Design	48
Table 2.9	Compilation of RIPEMD-160 Hash Function Design	53
Table 2.10	Compilation of SHA-256 Hash Function Design	59
Table 2.11	Previous HMAC Design	64
Table 3.1	Behavioural Modelling	79
Table 3.2	Buffer Initialisations	81
Table 3.3	Constant T[i] of MD5 Algorithm	83
Table 3.4	Shift, S, and message value, *k* of MD5 Algorithm	84
Table 3.5	Buffer Initialisation of SHA-1	86
Table 3.6	Constant K_t	87
Table 3.7	Round Function	88
Table 3.8	Initial Value	91
Table 3.9	Non-linear Function of RIPEMD-160	91
Table 3.10	Round Function	91
Table 3.11	Constant of RIPEMD-160	92
Table 3.12	Message Selection	92

Table 3.13	Shift Value	93
Table 3.14	Characteristics of SHA-256 Hash Function	94
Table 3.15	Buffer Initialisation of SHA-256 Algorithm	96
Table 3.16	RIPEMD-160 Input for Pipelined Design	134
Table 3.17	Comparison of Binary and Gray Encoding	159
Table 4.1	ASCII Code of Message Input of MD5 Design	172
Table 4.2	512-bit message input	173
Table 4.3	Message in the Little-Endian Format	173
Table 4.4	Round 1 of Non-linear Function F	175
Table 4.5	Round 4 of Non-linear Function I	177
Table 4.6	Final MD5 Hash Function	178
Table 4.7	Maximum Frequency and Area Implementation of MD5 Design	179
Table 4.8	Maximum Frequency of MD5b Design	180
Table 4.9	Area Implementation of MD5 Design (Arria II GX: EP2AGX45DF25C4)	180
Table 4.10	Simulation-based Power Estimation of Four Different Types of MD5 Designs	186
Table 4.11	Synthesis and Implementation of MD5 Design	188
Table 4.12	Comparison of MD5 Designs and Their Implementation	194
Table 4.13	ASCII Code of Message Input for SHA-1 Design	198
Table 4.14	512-bit Message Input	198
Table 4.15	Message in the Big-Endian Format	198
Table 4.16	Ain Output (t = 1 to t = 10)	200
Table 4.18	Ain Output (t = 71 to t = 80)	202
Table 4.18	Comparative Overview of Early SHA-1 Iterative Hardware Designs (2002–2009).	203
Table 4.19	A Output (t = 1 to t = 10)	207

Table 4.20	A Output (t = 31 to t = 40)	208
Table 4.21	Final SHA-1 Hash Function	210
Table 4.22	FPGA-based SHA-1 Implementation	216
Table 4.23	Area Implementations of SHA-1 and SHA-1 Unfolding	217
Table 4.24	Maximum Frequency of SHA-1 Design	218
Table 4.25	Maximum Frequency of SHA-1 Unfolding	220
Table 4.26	Synthesis and Implementation based on Altera Arria II GX	227
Table 4.27	Synthesis and Implementation based on Xilinx Virtex5 XC5VLX50T	227
Table 4.28	ASCII Code of Message Input RIPEMD-160 Design	229
Table 4.29	512-bit Message Input	230
Table 4.30	Message in the Little-Endian Format	230
Table 4.31	Output B_in0 and B_in1 for $1 \leq t \leq 16$	232
Table 4.32	Output of outb and routb for $1 \leq t \leq 11$	236
Table 4.33	Output outb and routb for $78 \leq t \leq 80$	238
Table 4.34	Synthesis and Implementation of RIPEMD-160 Hash Function	239
Table 4.35	Power Analysis of Both RIPEMD-160 and Pipelined RIPEMD-160 Designs	240
Table 4.36	Comparison of RIPEMD-160 Design with Previous Studies	241
Table 4.37	Synthesis and Implementation Results of RIPEMD-160	243
Table 4.38	Synthesis and Implementation Results of SHA-256 Design and SHA-256 Unfolding Factor 2 Design	252
Table 4.39	Comparison of Implementation Results with Other SHA-256 Designs	253
Table 4.40	Comparison of Synthesis and Implementation Results with Other SHA-256 Designs	255
Table 4.41	ASCII Code of Message Input “Sample #1”	263
Table 4.42	Message in Hexadecimal Format	263
Table 4.43	Key Input for HMAC-SHA1	263

Table 4.44	Key ⊕Ipad for HMAC-SHA1	263
Table 4.45	(Ko⊕Ipad) || Text for HMAC-SHA1	264
Table 4.46	Output of the First SHA-1 Design	264
Table 4.47	Ko ⊕ Opad	264
Table 4.48	(Ko⊕Opad) || Hash((Key ⊕Ipad)||Text)	265
Table 4.49	Output of the Second SHA-1 Design	265
Table 4.50	20-byte of HMAC-SHA-1 Output	265
Table 4.51	First SHA-1 Hash Function	266
Table 4.52	HMAC-SHA-1 Output	268
Table 4.53	HMAC-SHA-1 Design	273
Table 4.54	Power Analysis of HMAC-SHA-1 Design	273
Table 4.55	Comparison of FPGA-based Implementation of Previous HMAC Designs	274
Table 4.56	Comparison of FPGA-based Implementation Comparison of Previous HMAC-SHA-256 Designs	276

LIST OF FIGURES

		Page
Figure 1.1	Network encryption and authentication	4
Figure 2.1	Clock Gating (SemiEngineering, n.d.)	34
Figure 2.2 (a) and (b)	Pre-image Resistance or One-Wayness	36
Figure 2.3	Second Pre-image Resistance or Weak Collision Resistance	37
Figure 2.4 (a) and (b)	Collision Resistance or Strong Collision Resistance	37
Figure 2.5	Hash Function	39
Figure 3.1	Typical Design Flow	71
Figure 3.2	FPGA Design Flow	72
Figure 3.3	Design Overview of Hash Function and HMAC Algorithm	73
Figure 3.4	Design Overview of Hash Function and HMAC Algorithm	74
Figure 3.5	Flowchart for the Overall Hash Function and HMAC Design	76
Figure 3.6	Dataflow Modelling	78
Figure 3.7	MD5 Architecture	80
Figure 3.8	Compression Function of MD5	82
Figure 3.9	SHA-1 Compression Function	86
Figure 3.10	Compression Function of RIPEMD-160 Block Diagram	89
Figure 3.11	RIPEMD-160 Top-Level	90
Figure 3.12	Message Preprocessing of SHA-256 Hash Function	95
Figure 3.13:	Message Schedule of SHA-256 Algorithm	96
Figure 3.14	Compression Function of SHA-256 Algorithm	97
Figure 3.15	Message Authentication using Keyed- HMAC	99
Figure 3.16	Illustration of HMAC construct	100
Figure 3.17	Iterative Looping (Henriguez et al., 2006)	101

Figure 3.18 Inner-Round Pipelining (Henriguez et al., 2006) 101

Figure 3.19 Outer-Round Pipelining (Henriguez et al., 2006) 102

Figure 3.20 Pipelining and Unfolding Transformation 102

Figure 3.21 An Example of an Original DSP Programme (Parhi, 1999) 104

Figure 3.22 A DFG Corresponding to a DSP Programme (Parhi, 1999) 104

Figure 3.23 The Unfolded DFG Corresponding to the 2-Unfolded DSP Programme (Parhi, 1999) 105

Figure 3.24 State Encoding: (a) Binary Encoding and (b) Gray Encoding 106

Figure 3.25 Gating the Clock: (a) Normal and (b) Clock-Gated Implementation 107

Figure 3.26 Design Methodology of Hash Function 107

Figure 3.27 Top-Level Architecture of MD5 Binary Design 108

Figure 3.28 Top-Level Architecture of MD5 Gray Design 109

Figure 3.29 Top-Level Architecture of MD5 Binary with Gating Design 110

Figure 3.30 Top-Level Architecture of MD5 Gray with Gating Design 111

Figure 3.29 Top-Level Architecture of MD5 Binary with Controller Design 112

Figure 3.32 MD5 Compression Function Architecture 113

Figure 3.33 Counter MD5 module 115

Figure 3.34 Top Level of Unfolding MD5 Design 116

Figure 3.35 Compression Function of Unfolding MD5 Design 117

Figure 3.36 MD5Pipelining and Unfolding Compression Function 119

Figure 3.37 Top-Level Architecture of SHA-1 Design 120

Figure 3.38 SHA-1 Unfolding Compression Function 121

Figure 3.39 Top Level Architecture of SHA-1 Unfolding Design 123

Figure 3.40 Top Level of SHA-1 Inner Pipelining Design 124

Figure 3.41 Top Level of SHA-1 Four-stage Pipelining Design 125

Figure 3.42 Top Level of SHA-1 40-stage Pipelining Design 127

Figure 3.43 Top Level of SHA-1 Unfolding 40-stage Pipelining Design 128

Figure 3.44 RIPEMD-160 Iterative Architecture Design 131

Figure 3.45 Top-Level Architecture of RIPEMD-160 Design 132

Figure 3.46: Pipelined Design 133

Figure 3.47 RIPEMD-160 Pipelined Architecture Design for a Single Block 135

Figure 3.48 Top-Level Architecture of Pipelined RIPEMD160 Design 137

Figure 3.49 Proposed RIPEMD-160 Unfolding Factor 2 Design 139

Figure 3.50 Coder Module of RIPEMD-160 Unfolding Factor 2 140

Figure 3.51 KConst Module of RIPEMD-160 Unfolding Factor 2 141

Figure 3.52 Message Module of RIPEMD-160 Unfolding Factor 2 141

Figure 3.53 Function Parallel of RIPEMD-160 Unfolding Factor 2 142

Figure 3.54 Coder Module of RIPEMD-160 Unfolding Factor 4 144

Figure 3.55 KConst Module of RIPEMD-160 Unfolding Factor 4 144

Figure 3.56 Message Module of RIPEMD-160 Unfolding Factor Four 145

Figure 3.57 Function Parallel Module of RIPEMD-160 Unfolding Factor 4 145

Figure 3.58 Top Level of SHA-256 Design 149

Figure 3.59 $Temp_{1o}$ Block Diagram Architecture 150

Figure 3.60 $Temp_{2o}$ Block Diagram Architecture 150

Figure 3.61 Cho (next_e,e,f) Function Architecture 151

Figure 3.62 Majo(next_a,a,b) Function Architecture 151

Figure 3.63 $0o(next\_a)$ Architecture 151

Figure 3.64 $\sum_{1O}(next\_e)$ Architecture 151

Figure 3.65 σ_{0_0} Architecture 152

Figure 3.66 σ_{1o} Architecture 152

Figure 3.67 Architecture of Temp11 Block Diagram 153

Figure 3.68 Architecture of $Temp21$Block Diagram 154

Figure 3.69 Architecture of $Temp12$ Block Diagram 156

Figure 3.70 Architecture of $Temp22$ Block Diagram 156

Figure 3.71 Architecture of σ_{0o} 157

Figure 3.72 Architecture of σ_{1o} 157

Figure 3.73 Message Schedule of SHA-256 Unfolding Factor 4 Design 158

Figure 3.74 Message Authentication using HMAC 160

Figure 3.75 Implementation of Proposed HMAC-SHA-1 161

Figure 3.76 Top-Level Architecture of HMAC-SHA1 Design 162

Figure 3.77 Illustration of HMAC Construction 164

Figure 3.78 Implementation of Proposed HMAC-SHA-256 165

Figure 4.1 CAD Tool Simulation Flow Design 170

Figure 4.2 Non-Linear Function F for B_t (t = 0 to t = 8) 175

Figure 4.3 Non-linear Function I for B_t (t = 57 to t = 63) 176

Figure 4.4 Output of MD5 Hash Function 178

Figure 4.5 Simulation Results of MD5 Binary Design 181

Figure 4.6 Simulation Results of MD5 Gray Design 181

Figure 4.7 Simulation Results of MD5 Binary with Gating Design 182

Figure 4.8 Simulation Results of MD5 Gray with Gating Design 182

Figure 4.9 Simulation Results of MD5 Design with Input “a” 183

Figure 4.10 Functional Simulation of MD5 Unfolding Design 191

Figure 4.11 Timing Simulation of MD5 Unfolding Design 191

Figure 4.12 Functional Simulation of MD5 Unfolding with Four-stage Pipelining Design 191

Figure 4.13 Timing Simulation of MD5 Unfolding with Four-stage Pipelining Design 192

Figure 4.14 Functional Simulation of MD5 Unfolding with 32-stage Pipelining Design 192

Figure 4.15 Timing Simulation of MD5 Unfolding with 32-stage Pipelining Design 192

Figure 4.16 Comparison of Performance-to-Area Ratio of MD5 Designs 196

Figure 4.17 FMax of MD5 Design 197

Figure 4.18 Throughput of MD5 Design 197

Figure 4.19 Ain(t= 1 to t = 10) 200

Figure 4.20 Ain (t = 71 to t = 80) 202

Figure 4.21: A (t = 1 to t = 10) 207

Figure 4.22: A (t = 31 to t = 40) 208

Figure 4.23 Final Simulation Output of SHA-1 Hash Function 209

Figure 4.24 Output of SHA-1 Hash Function 210

Figure 4.25 Functional Simulation of SHA-1 Inner Pipelining Design 212

Figure 4.26 Timing Simulation of SHA-1 Inner Pipelining Design 212

Figure 4.27 Functional Simulation of SHA-1 with Four-stage Pipelining Design 212

Figure 4.28 Timing Simulation of SHA-1 with Four-stage Pipelining Design 213

Figure 4.29 Functional Simulation of SHA-1 with 40-stage Pipelining Design 213

Figure 4.30 Timing Simulation of SHA-1 with 40-stage Pipelining Design 213

Figure 4.31 Functional Simulation of SHA-1 Unfolding with 40-stage Pipelining Design 214

Figure 4.32 Timing Simulation of SHA-1 Unfolding with 40-stage Pipelining Design 214

Figure 4.33 Area Implementations for Both Proposed SHA-1 Designs 221

Figure 4.34 Maximum Frequency and Throughput for Both Proposed SHA-1 Designs 222

Figure 4.35 Total Thermal Power Dissipation for Both Proposed SHA-1 Designs 222

Figure 4.36 Unfolding Transformation of SHA-1 Algorithm Compression Function 224

Figure 4.37 Four Stages of SHA-1 Algorithm Architecture 225

Figure 4.38 Output B_in0 and B_in1 for $1 \leq t \leq 16$ 232

Figure 4.39 Final Output of RIPEMD-160 Design 235

Figure 4.40 Output of outb and routb for $1 \leq t \leq 11$ 236

Figure 4.41 Output outb and routb for $78 \leq t \leq 80$ 237

Figure 4.42 Output of RIPEMD-160 Hash Function 238

Figure 4.43 Comparison of performance-to-area ratio 245

Figure 4.44 Simulation of RIPEMD-160 unfolding with factor 4 design 246

Figure 4.45 Functional Simulation of SHA-256 Design 249

Figure 4.46 Timing Simulation of SHA-256 Design 249

Figure 4.47 Functional Simulation of SHA-256 Unfolding Design 251

Figure 4.48 Timing Simulation of SHA-256 Unfolding Design 251

Figure 4.49 Timing Simulation Waveform of SHA-256 Design 257

Figure 4.50 Timing Simulation Waveform of Unfolding SHA-256 with Factor 2 Design 258

Figure 4.51 Timing Simulation Waveform of Unfolding SHA-256 with Factor Four Design 258

Figure 4.52: (a) FPGA Output Implementation Display and (b) Hardware Implementation of SHA-256 Hash Function 259

Figure 4.53 HMAC-SHA-1: Hash ((Ko⊕Ipad) || Text) 267

Figure 4.54 HMAC-SHA-1: Hash((Ko⊕Opad) || Hash((key⊕Ipad)||Text)) 269

Figure 4.55 Output of HMAC- SHA-1 270

Figure 4.56 Simulation Waveform of HMAC-SHA-256 Design 271

Figure 4.57 HMAC Calculation of SHA-256 Design with a Specific Key 272

Figure 4.58 Maximum Clock Frequency of HMAC-SHA-256 272

LIST OF ABBREVIATIONS

AH	Authentication Header
ALUTs	Adaptive Look-Up Tables
BlockRAM	Block Random Memory Access
CAD	Computer-Aided aided Design
CLBs	Configurable Logic Blocks
CMOS	Complementary Metal-Oxide Semiconductor
DFG	Data Flow Graph
DSP	Digital Signal Processing
ESP	Encapsulating Security Payload
FPGAs	Field Field-Programmable Gate Arrays
FSM	Finite State Machine
HDLs	Hardware Description Languages
HMAC	Hash-based Message Authentication Code
I/O	Input/Output
IEEE	Institute of Electrical and Electronic Engineers
IP	Internet Protocol
IPSec	Internet Protocol Security
IR	Industrial Revolution
ISO	International Organization Organisation for Standardization
LE	Logic Element
MAC	Message Authentication Code
Mbps	Megabits per second

MD5	Message Digest 5
OSI	Open System Interconnection
PLDs	Programmable Logic Devices
RIPEMD-160	RACE Integrity Evaluation Message Digest 160
RTL	Register-Transfer Level
SDGs	Sustainable Development Goals
SHA-1	Secure Hash Algorithm 1
SHA-256	Secure Hash Algorithm - 256
SoC	System on Chip
VHDL	Very High-Speed Integrated Circuit Hardware Description Language
WEF	World Economic Forum
WLTS	Wireless Transport Layer Security

CHAPTER 1

INTRODUCTION

1.1 Research Background

Cryptography protects digital communication by converting information into encoded representations that prohibit unauthorised access and maintain data integrity. As modern societies become more reliant on networked technologies, maintaining the confidentiality, authenticity, and dependability of transmitted data has become critical. Cryptographic hash functions, in particular, have developed as an essential component of secure communication protocols, notably Internet Protocol Security (IPSec), where they facilitate authentication and data integrity methods across network layers. Cryptography is the discipline of encoding messages in secret codes, where only the designated recipient can decipher an encrypted message. It is a critical tool for safeguarding information while maintaining confidentiality and integrity during data transmission. Several essential cryptographic requirements for computer security systems are as follows:-

i. Confidentiality or secrecy: Only the intended user can access an encrypted message, which must first be decrypted to understand the information received.

ii. Data integrity: To avoid data alteration, some algorithms incorporate additional procedures to protect the message from forgery. The intended receiver receives the original data.

iii. Authentication: The procedure of verifying one's identification; it is essential to establish that the key remains confidential.

Cryptographic algorithms are one of the most important applications of hardware implementation for embedded security system design and electronic data transmission (Henriguez et al., 2006). Cryptographic algorithms can be classified into three types: symmetric, asymmetric, and unkeyed algorithms. Cryptographic algorithms represent a fundamental component in the hardware implementation of embedded security systems and electronic data transmission. Symmetric cryptography (secret key), asymmetric cryptography (public key), and hash function are three categories of cryptographic algorithms. The input message, such as characters or letters, is referred to as plaintext, and characters or letters of the secret form of the message are referred to as ciphertext. The transformation of plaintext into ciphertext is referred to as encryption, whereas the process of reverting ciphertext back to plaintext is known as decryption (Nichols, 1999). Symmetric cryptography employs the same key for both encryption and decryption of the plaintext; in contrast, asymmetric cryptography utilises different keys for these processes. A hash function does not utilise a key to generate its output.

The increasing use of embedded systems and Internet of Things devices has increased the need for effective hardware implementations of cryptographic algorithms that can achieve excellent performance while using the fewest resources and power possible. Since Field-Programmable Gate Arrays (FPGAs) are flexible, reconfigurable, and suitable for parallel processing, implementing hash functions on them offers a promising solution. However, there are still issues with designing to balance power, performance, and area, particularly when using various hash algorithms like MD5, SHA-1, RIPEMD-160, and SHA-256 for applications that need strong message authentication. Hash function, known as one-way cryptography, has no key and is widely used in the protocol scheme. The cryptographic hash function algorithms on reconfigurable hardware are widely used for data protection in

the automotive, multimedia content, or any cryptographic materials. A study and analysis of cryptographic hash function algorithms involving encryption techniques are essential to the embedded digital security system design for the next generation. The cryptographic hash function, also known as the message digest algorithm, is an algorithm that translates a random string of characters into hash code output. This function is extensively utilized in cryptographic applications and is particularly valuable for message authentication due to its robustness against collision attacks. Ronald Rivest developed the MD5 message digest, which is one of the most renowned hash functions (Henriguez et al., 2006). The MD5 algorithm is designed for digital signature applications, in which a large file must be securely compressed prior to encryption. In this project, a variety of methodologies are employed to design additional hash functions, including MD5, SHA-1, RIPEMD-160, and SHA-256.

Some applications, such as message authentication code (MAC), rely on different hashing operations. There are various hash functions, including Message-Digest 5 (MD5), Numerous hash functions exist, such as Message Digest 5 (MD5), RACE Integrity Primitives Evaluation Message Digest 160 (RIPEMD-160), Secure Hash Algorithm 1 (SHA-1), and Secure Hash Algorithm 256 (SHA-256). The network layer constitutes the third of the seven layers in the Open Systems Interconnection (OSI) concept, commonly referred to as the Internet. It handles network addressing and physical data routing. Nowadays, enhanced internet security is necessary to safeguard networks from illegal surveillance. Consequently, internet protocol security (IPSec) facilitates secure communication over the internet by encrypting and/or authenticating network traffic at the internet protocol (IP) layer. IPsec is a security protocol utilised on the internet. Encapsulating security payload (ESP) and authentication header (AH) protocols are different entities. The MAC value is retained in the authentication data files of the AH and ESP.

Computer communication architecture and protocol standards were modelled after the Open System Interconnection (OSI) model by the International Organisation for Standardisation (ISO) (Stallings, 1996). Physical, presentation, application, session, data link, network, and transport are the seven layers that make it up. A network layer is considered as an internet layer, where the transmission of data from one terminal to another terminal occurs based on address and routing network. Figure 1.1 illustrates the sending of message from a transmitter to a receiver. Encryption occurs before the IP layer, which is at the IPSec layer; subsequently, the application passes the message through the network. The IP layer consists of an IP address to route the message through the network to arrive at its destination. The destination address will be determined by the router using the IP address of the sender. Subsequently, the data that have been transmitted are obtained by decrypting the packet.

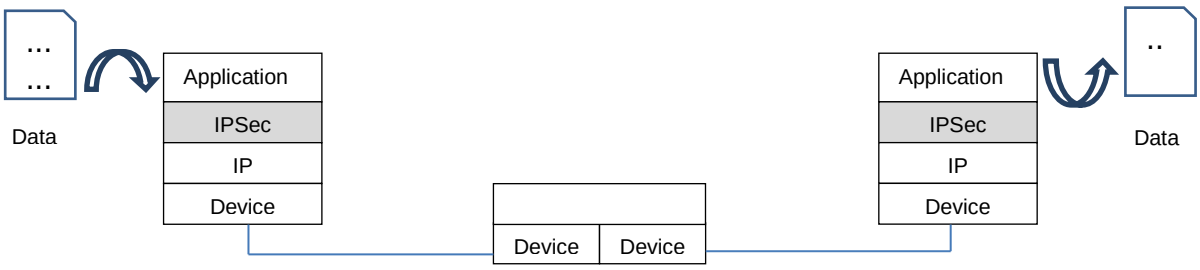


Figure 1.1 Network encryption and authentication

The internet community has developed the IPSec to provide cryptographic protection for internet traffic (Nichols, 1999). The network layer, which is the third layer of the seven-layer OSI model, is where IPSec is implemented. IPSec is the most recent IP-based protocol. It is also referred to as network data encryption protocols. The IP provides authentication and encryption facilities to the network to control the network and protect it from unauthorised monitoring. Nowadays, a traffic network is exposed to eavesdropping and unauthorised users if there is no security element embedded in the network. Hardware

implementation of cryptographic algorithms is one of the most crucial components of embedded system architecture. It offers approaches and systems for conducting safe and authenticated network transactions. Therefore, the solution to this problem is installing an appropriate encryption and authentication product in the network. Implementations of cryptographic hash function algorithms on reconfigurable hardware for both high speed and low area are practical solutions for embedded systems, which can produce different results based on the structure of reconfigurable logic.

When evaluating the reconfigurable hardware of field-programmable gate array (FPGA) implementation, numerous factors need to be considered, notably area implementation, as today's society focuses on making devices as compact as feasible, subsequently increasing device efficiency. In other words, design performance is improved. However, there are trade-offs among increased performance, implementation in a smaller area, and low-power design. Therefore, advanced design strategies are required to balance speed, area implementation, and power usage. When algorithms are involved, the FPGA is frequently an excellent choice; however, it is hindered by the issue of high-power consumption. Hardware description language (HDL) is generally used in digital LSI design, including FPGA design. Furthermore, HDL designs are portable, as well as easy to write and debug. They do not depend on a specific technology for hardware implementation.

Hardware designs by HDL can be classified into different types of HDL modelling, such as structural, dataflow, and behavioural modelling. Structural modelling is used for the hierarchical modular design approach. The dataflow modelling style is similar to the Boolean equation, allowing a digital system to be designed in terms of function. Meanwhile, behavioural modelling uses procedural statements to construct the design in the always

block, and this style is identical to a natural language description of the circuit functionality. This research uses and evaluates a combination of all types of HDL modelling. The performance of digital circuits depends on HDL modelling and the structure of FPGA family devices.

In general, several types of FPGAs are provided by many FPGA vendors. For instance, for Altera Co., one of the well-known FPGA vendors, the Stratix device is used for high-performance implementation, and the Arria device is a low-cost transceiver FPGA. Meanwhile, the Cyclone device is the best choice for the design that requires low-cost implementation. Embedded designs are represented by a top-level module unit consisting of several modules, where a combination of the module units can be arranged using different methodologies and modelling for performance enhancement. The critical path is the primary route via which the entire system of the design operates correctly. Therefore, the critical path can be shortened in order to create the high-performance hardware. During the compilation process, three forms of sequential system delays are used to achieve the maximum frequency of the design: pin-to-pin combinational delay, clock-to-output delay, and register-to-register delay.

This research is driven by the necessity to create cryptographic hash function architectures that are based on FPGAs and that resolve these performance trade-offs, thereby facilitating secure data transmission in critical domains such as network security, embedded routers, and secure IoT applications. This project seeks to enhance high-efficiency hardware solutions by systematically assessing various design processes and employing advanced optimisation techniques to address current security requirements.

1.2 Problem Statement

Traffic networks are prone to eavesdropping and illegal access in the absence of a network-integrated security element. In addition, slow data transmission results in limited throughput and poor performance in terms of speed, area, and power. As a result, selecting a suitable encryption and authentication product for the network solves this problem. The third network layer of the seven-layer OSI architecture employs the IPSec protocol. Nowadays, security on the network is a major issue in data transmission. Hence, it is important to improve methods for developing FPGA-based systems by making better use of HDLs for design optimisation (Ma et al., 2023). A network layer needs to be secure enough with cryptographic algorithms so that it can be used to accommodate encryption and authentication processes.

Some applications, including MAC and digital signature, can make use of the hash function. Most previous designs utilise iterative methods to create hash function algorithms, which result in low throughput (Jayapandian et al., 2018a). A previous study applied the unfolding factor-2 to a hash function. Consequently, increasing the number of unfolding leads to a higher performance design (Chen & Li, 2020). Therefore, high-performance design of cryptographic hash function algorithm is one of the important aspects of security algorithm. Hence, designing a cryptographic hash function algorithm on reconfigurable hardware with high speed, low power, and small area implementation needs to be considered. Efficient implementation of cryptographic hash function algorithm on reconfigurable hardware is a technical aspect that must be considered thoroughly. A combination of two design methods, such as pipelining and unfolding, into a single hash function design can enhance hardware implementation (Makkad & Sahu, 2017). Consequently, using design

techniques that optimise performance, power consumption, timing, and throughput, high-performance hash functions can be developed in terms of area, speed, and power.

Cryptographic hash functions play a vital role in ensuring data security within digital systems, especially in the context of HMAC-based authentication. Contemporary applications, including IoT, embedded security, and automotive networks, are progressively utilising FPGA-based implementations owing to their reconfigurability and high-performance attributes. Existing hash function hardware designs primarily utilise iterative architectures, leading to constraints in throughput, elevated latency, and less than optimal power and area efficiency. Hash functions become unsuitable for real-time, resource-constrained environments without optimised hardware architectures, resulting in performance bottlenecks. In this study, performance is defined as the sum of throughput (measured in Mbps), latency (cycles per operation), power consumption (mW), and area utilisation (logic elements and registers). The identified limitations result in increased energy requirements and possible weaknesses in authentication procedures. Consequently, it is essential to establish optimised and systematic design methodologies for cryptographic hash functions. These methodologies should aim to enhance throughput, decrease latency, minimise area and power consumption, and provide robust and efficient support for secure HMAC operations in contemporary FPGA-based systems.

1.3 Motivation and Research Gap

Numerous studies have been conducted on the usage of cryptographic hash functions in HMAC to guarantee data authenticity and integrity. Recent studies have demonstrated significant advancements in optimizing hash functions for hardware platforms (El Hadj Youssef et al., 2020). However, relatively less focus has been placed on creating high-

performance hash function architectures especially suited for FPGA-based HMAC implementations. Most current research focusses on general-purpose hardware implementations or software-based solutions, which frequently fall short of the demanding resource and performance requirements of contemporary embedded systems. This research aims to overcome existing restrictions by exploring the incorporation of a high-performance hash function architecture which utilizing techniques such as pipelining, unfolding, and dynamic power reduction into an HMAC design executed on FPGA (Chen & Li, 2020). The study intends to contribute to the larger field of cryptographic hardware security by assessing the effects of this integration on crucial performance measures, such as maximum operating frequency, throughput, area utilisation, and power consumption. Enhancing hardware-based cryptographic security and enabling useful applications in resource-constrained environments like Internet of Things devices, embedded network routers, and effective secure communication systems where energy efficiency and real-time data protection are critical is the aim (Sideris et al., 2024).

The cryptographic hash function algorithms on reconfigurable hardware are widely used for data protection in the automotive, multimedia content, or any cryptographic materials. A study and analysis of cryptographic hash function algorithms involving encryption techniques are essential to the embedded digital security system design for the next generation. The cryptographic hash function, also known as the message digest algorithm, is an algorithm that translates a random string of characters into hash code output. This function is extensively utilized in cryptographic applications and is particularly valuable for message authentication due to its robustness against collision attacks. An improvement in design methodology was proposed by several previous researchers, utilizing the unfolding factor-2 methodology in the context of hash functions. Increasing the number

of unfolding leads to high-performance design (Chen & Li, 2020). Therefore, a combination of two design methods, such as pipelining and unfolding, into a single hash function design can enhance hardware implementation (Makkad & Sahu, 2017).

The ability to design any circuit that satisfies the high-speed systems and hardware cost limitations makes FPGA devices one of the most preferred media implementations for digital systems (Ma et al., 2023). Thus, based on previous design, several methodologies have been proposed to improve and enhance the performance of hash functions to obtain high-performance design. Design analysis for each different type of hash function is proposed in this research to solve the low-performance design, such as unfolding design with high number of factors, pipelined design with different number of stages, state encoding design, and clock gating, which focuses on low-power design. Altera Quartus II and ModelSim are two computer-aided design (CAD) programmes that are used to implement and simulate these designs, respectively.

Although there have been notable improvements in the designs of FPGA-based cryptographic hash functions, several key limitations continue to persist. Current methodologies frequently experience constraints related to maximum operating frequencies and throughput, primarily due to the use of iterative or partially unfolded architectures. Although certain studies have examined pipelining and others have focused on unfolding, there has been a lack of comprehensive research on the systematic integration of pipelining with elevated unfolding factors aimed at optimising both throughput and latency. Additionally, previous studies rarely integrate dynamic power reduction methods like Gray encoding into a unified pipelining and unfolding framework, particularly regarding HMAC implementations for IP security applications. This thesis addresses existing gaps by

proposing an innovative design methodology that integrates pipelining and unfolding with optimised parameters, resulting in improved performance metrics characterised by enhancements in throughput, area efficiency, and dynamic power consumption. Furthermore, the performance gains are enhanced through the use of higher unfolding factors, which have been systematically optimized to maximize parallel processing efficiency and minimize latency.

1.4 Research Objective

This research investigates how the HDL coding style impacts the performance of FPGA-based hardware algorithms. To achieve this goal, hash-based message authentication code (HMAC) and cryptographic hash functions are chosen as design targets. Research into the development of these algorithms aims to improve power consumption, implementation area, and processing time. Therefore, designing the cryptography several hash functions and HMAC application is a good example for this study. Therefore, for optimal performance of the target system, it is recommended to use an improved HDL coding style. The objectives of this research are:

i. To formulate unfolding transformation factor 2 and factor 4 methods into hash function design.

ii. To develop low-power design techniques using state encoding and clock gating into hash function design.

iii. To integrate cryptographic hash functions into the HMAC design and to analyse their performance quantitatively with respect to maximum frequency,

resource utilization (area), power consumption, and throughput on the FPGA platform.

iv. To apply and validate the efficient implementation of cryptographic hash function algorithms on FPGA in terms of frequency, area utilisation and power consumption.

Design performance can be improved in terms of throughput and speed by enhancing hash function design methodology using unfolding techniques. Applying low-power approaches can increase the power efficiency of cryptographic hash function implementations. This is important since enhancing FPGA performance design involves addressing the essential issue of low power. This aligns with the goal of improving power efficiency in cryptographic hash function implementations, which is crucial for optimising FPGA performance in HMAC applications. The purpose of the research is to specifically examine the utilisation of hash functions within HMAC, which is a key aspect of the research's scope. The final objective is key to the practical application of this research, emphasizing the validation and efficiency of implementing hash functions on the FPGA, which relates to the overall goal of optimising performance for HMAC applications.

1.5 Hypothesis

Data traffic is particularly susceptible to eavesdropping and unauthorized access in modern networks as most of the traffic does not have integrated security measures. Thus, information's secrecy, integrity, and authenticity are greatly jeopardised while passing via traffic networks due to the increased likelihood of interception, manipulation, and other forms of illegal access. Therefore, this research is concerned with developing design methodologies for enhancing the performance of cryptographic hash functions in digital

design. Using techniques like pipelining and unfolding to design the cryptographic hash function allows for high performance with significantly reduced area utilisation in terms of speed, power efficiency, and throughput. Thus, it can result in cryptographic hash functions with higher throughput design capability.

One would consider an area, resource usage, and maximum frequency in the synthesis compilation report of a design in order to attain efficiency in the design of hash functions. The technique evaluates dynamic power consumption with and without Gray encoding using the PowerPlay Power Analyzer, with the goal of improving the power efficiency of hash algorithms. The application of Gray encoding for designing hash functions will reduce dynamic power consumption considerably due to reduced switching activity in the hardware implementation. The application of unfolding techniques will contribute to the reduction of latency in the processing of hash functions. Reduction in latency can be obtained from design simulation using ModelSim with testbench files for various hash functions. This is done to ensure high-throughput, resource-efficient implementation of the results that will meet critical security requirements in network systems and ensure optimal encryption and authentication processes.

1.6 Research Scope

This study focusses on the implementation of cryptographic hash function algorithms on FPGA platforms to determine their performance and appropriateness for embedded security applications. FPGAs were chosen as the hardware medium because of their configurability, support for parallel processing, and ability to expedite cryptographic procedures within stated resource and cost limitations. However, FPGA devices have intrinsic limits, such as limited logic element availability, constrained memory resources,

and rigorous timing requirements, which must be addressed during the design process to ensure efficient and dependable operation.

This research selects a cryptographic hash function algorithm as an application for digital implementation to assess the impact of Verilog code. For digital systems, FPGAs are a common media implementation because they can realise any circuit that satisfies both high-speed system requirements and hardware cost limits. This research seeks to design and build a cryptographic hash algorithm utilising Verilog code for hardware modelling to achieve optimal results. The chosen hash function architectures are modelled by the project using Verilog HDL, which is the main hardware description language. Altera Quartus II is used as a synthesis and place-and-route tool to convert high-level Verilog descriptions into optimised gate-level implementations that are compatible with the selected FPGA family. Device-specific limitations, such as area utilisation, clock distribution, and routing congestion, must be carefully considered during this procedure. To confirm the accuracy of every design and examine waveform behaviour, detailed functional and timing simulations are carried out using ModelSim.

The scope of this research includes the design, implementation, and comparative evaluation of several widely used cryptographic hash functions, including MD5, SHA-1, SHA-256, and RIPEMD-160. These functions are investigated as standalone implementations and within HMAC configurations to facilitate authentication. Additionally, the study incorporates and evaluates advanced design methodologies such as unfolding, pipelining, clock gating, and Gray encoding, with the goal of enhancing throughput, reducing latency, and minimizing power consumption under the constraints imposed by FPGA hardware. Through this investigation, the research aims to demonstrate effective techniques

for overcoming FPGA limitations and achieving high-performance cryptographic hardware solutions suitable for modern embedded systems.

1.7 Significance of the Study

This research is significant as it focuses on achieving a high-performance hash function design by analysing the structures of several cryptographic hash functions, including MD5, SHA-1, RIPEMD-160, and SHA-256. These hash functions are critical components for applications in MACs and digital signatures. In modern security systems, it is often desirable to have short digital signatures for efficiency. To address this issue, signing long messages using a hash function is a practical solution, as it reduces computational overhead while ensuring security.

Numerous previous designs have explored different methods, such as iterative, pipelining, and unfolding techniques, to optimise the performance of hash functions. It is still necessary to determine the optimal combination of approaches to optimise performance with regard to area efficiency, power consumption, and throughput, as each of these methods has its limitations when applied alone. For secure data transmission, especially in situations with limited resources like high-speed banking systems and Internet of Things (IoT) devices, high-performance hash function designs are crucial.

In order to address that research gap, this work investigates and proposes a design methodology for improving the performance of cryptographic hash functions by combining unfolding and pipelining techniques. The unfolding technique enables higher throughput by restructuring the hash function algorithm to execute multiple iterations simultaneously, significantly reducing latency. Meanwhile, pipelining increases the maximum operating frequency of the design by enabling continuous data flow across processing stages. By

combining these two techniques, the proposed methodology achieves a high-performance hash function design, ensuring both high throughput and efficient resource utilisation.

Additionally, power efficiency is a critical factor in designing hash functions for modern applications. This research incorporates low-power design techniques, such as Gray encoding and clock gating, to minimise dynamic power dissipation caused by signal toggling. Gray encoding reduces switching activity in the logic design, while clock gating disables unused portions of the circuit, resulting in a power-efficient implementation. This approach ensures that the proposed hash function designs are suitable for energy-sensitive applications, such as IoT devices and mobile systems.

This research is significant due to its capacity to produce efficient designs for cryptographic hash functions through:

i. Higher throughput: Achieved by increasing the unfolding factors of the hash function algorithm, reducing latency proportional to the number of factors used.

ii. Low-power consumption: Attained using dynamic power optimisation techniques like Gray encoding and clock gating.

iii. Improved design methodology: Achieved through modifications to the hash function structure, enabling parallel input and output handling for better performance.

This study contributes to advancing cryptographic design methodologies by addressing existing limitations and providing scalable, high-performance solutions for secure communication and data transmission in modern security systems.

1.8 Thesis Organisation

The project outlines are organised into five chapters in this section: Chapter 1, 2, 3, 4, and 5. Each chapter is described in detail in the following paragraphs:

The project's overview and introduction are detailed in Chapter 1. A brief discussion on the problem statement of the project, as well as the objectives and research outlines, are explained in this chapter.

The second chapter discusses the project's literature review. Chapter 2 consists of a description of software that used during the design process. This chapter discusses the designing of the project using Verilog HDL code. To implement the design, Altera Quartus II and ModelSim are used as the CAD design tool and simulator, respectively. Furthermore, this chapter includes an explanation of FPGAs. As the evaluation of the output design emphasises maximum frequency, a brief discussion of maximum frequency is presented. Finally, a literature review of hash function is described briefly.

Chapter 3 describes the methodology of the design. The flow of the hash function and HMAC design structure is explained step by step, based on different types of hash functions. This chapter provides the descriptions of all non-linear functions for various forms of hash functions. Brief descriptions of Gray code, clock gating, unfolding, and pipelined design are given.

Chapter 4 discusses the results of hash function and HMAC design. The explanation of the hash function and HMAC results covers the progression from the beginning of the design to the final stage of obtaining output. ModelSim is used as the simulator tool for the design. A variety of hash functions are designed, and each simulation is briefly described.

This thesis concludes with Chapter 5. This chapter focuses on the results that reflect the success of this research. Besides, several recommendations for future work are suggested in this chapter.

1.9 Thesis Contribution

This study examines a fast implementation of the HMAC, which employs its algorithm to ensure data authenticity and integrity for optimisation of design and hardware efficiency by applying the hash function algorithm. During data transfer, HMAC is critical for message authentication. Using Verilog HDL code, it was successfully developed with the implementation of an FPGA device. Furthermore, the Altera Quartus II CAD tool was employed to improve the design's maximum frequency.

The following are the contributions made by this thesis:

i. Investigate the throughput of hash function design using various types of techniques.
ii. Formulate equation for each non-linear function for hash function with unfolding transformation with factor 2 and factor 4.
iii. Design the combination of unfolding transformation and pipelining methods to obtain the high-performance design of hash function.
iv. Develop a low-power design with state encoding for dynamic power of hash function design.
v. Propose HMAC design with a wide variety of hash function designs.
vi. Using the FPGA to verify the design by implementing the hash function design.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

The Industrial Revolution (IR) has brought about a significant transition and evolution in terms of different fields. The IR 1.0 first emphasised mechanisation and the utilization of steam power, while the IR 2.0 subsequently transitioned to the utilisation of electrical energy. Automated and robotic systems are employed in place of human labour in the context of the IR 3.0. It can be distinguished by a change from manual to machine-based industry and from an agricultural to an industrial economy. Integrating modern technologies like cyber security, automation, robotics, the Internet of Things (IoT), and artificial intelligence defines the current era of technological innovation and advancement, known as the fourth Industrial Revolution (IR 4.0).

Among the nine pillars of the IR 4.0 is cybersecurity. The new technologies integrate the physical, digital, and biotechnology on every sector of society and economy. These innovations will keep bringing billions more people online and more users to the internet. Thus, cybersecurity significantly improves the organization and online business. It refers to the prevention of malicious attacks and unauthorised access to digital systems, networks, and devices. Among the United Nations' Sustainable Development Goals (SDGs), Goal 16—"Peace, Justice, and Strong Institutions"—deals specifically with security. This objective aims to foster peaceful and inclusive societies, guarantee equitable access to justice, and establish effective, responsible, and transparent institutions at all levels.

This research focuses on cryptographic security of hash function design using Altera Quartus II and ModelSim Simulator tools on field-programmable gate array (FPGA) as hardware implementation. Embedded system design refers to the process of building hardware and software systems specifically tailored to execute a designated purpose or set of functions within a larger system. This technique is also known as embedded computing. Embedded systems can be discovered in a wide variety of items, including home appliances, automobiles, medical devices, industrial control systems, and consumer electronics. Electronics boards like the Arduino Uno, FPGA, microcontroller, and microprocessor are categorised under the embedded system design. Consumer electronics, automobile systems, medical devices, and factory automation are among the applications using embedded systems. Smart thermostats, digital cameras, and industrial control systems are examples of embedded systems. This study focuses on FPGA-based digital design implementation. This is because FPGA implementation can be used in the development of a wide variety of digital systems, including those involving RFID, filters, robotics, medical monitoring, and security (Saju & Sreehari, 2021).

2.1.1 Overview of Hash Functions in Cryptography

There are substantial performance and resource management benefits to implementing cryptographic hash functions on FPGA using Verilog HDL. Various algorithms, particularly Secure Hash Algorithm 256 (SHA-256) and AES, have been explored for their efficiency in cryptographic applications, especially in blockchain technology. The former is essential for blockchain applications to guarantee data integrity and authenticity . FPGA architectures for SHA-256 have been synthesised to optimise power and time consumption, demonstrating improved resource management compared to software

implementations. Additionally, SHA-256 has been implemented on FPGAs for IoT devices, utilising parallel execution with clustered cores to enhance throughput (Santos et al., 2024).

Hash function can be used for security enhancement by combining hash function with Rivest-Shamir-Adleman algorithm to improve security in blockchain applications (Peroumal et al., 2022). Data encryption and authentication can be enhanced through the use of FPGA implementations of hash functions. An FPGA implementation platform may introduce resource allocation and power consumption, which can provide benefits to cryptographic applications in various fields. It is crucial to balance these aspects to optimise cryptographic applications across various sectors.

2.1.2 Importance of Hash Function in HMAC Applications

A hash function, which is crucial for data authentication, is used by hash-based message authentication codes (HMACs) to ensure the authenticity and integrity of messages. Hash functions and secret keys are components of an HMAC structure. Improving the hash function increases the performance of HMAC application design. In HMAC applications, the hash function ensures that the integrity and authenticity of data. Furthermore, the hash function in HMAC ensures data integrity, authentication, and efficiency.

According to Uriawan et al. (2024), hash functions are used in HMAC applications to generate hash values. These values can be used to identify data alterations and ensure data integrity. The HMAC structure also makes use of the secret key. According to(Uriawan et al., 2024), hash functions are used in HMAC applications to generate hash values. These values can be used to identify data alterations and ensure data integrity. The HMAC structure also makes use of the secret key. Consequently, it can prevent unauthorised access by utilising a mechanism to verify the source of the message (Castellon et al., 2022).

Additionally, the efficiency of the design is influenced by the hash function selected in the HMAC design. Nowadays, the SHA-256 hash function has been selected due to its security and performance (Anwar et al., 2021). The efficiency of HMAC can be enhanced while preserving the security of design by optimising the energy consumption of hash functions. Table 2.1 lists the use of hash functions in HMAC applications.

Table 2.1 Compilation of Hash Functions in HMAC Applications

No.	Overview	Finding	Reference
1	HMAC-SHA256 improves file security when properly implemented. Understanding cryptographic concepts is critical to prevent vulnerabilities.	Data integrity is effectively protected, and manipulation is detected by HMAC-SHA256. Furthermore, SHA256 generates unique file signatures that prevent unauthorised modifications.	(Uriawan et al., 2024)
2	Applying an energy-efficient algorithmic engineering approach to the HMAC fundamental hash function.	The use of HMAC achieved potential energy savings of up to 17%. Keeping functionality while reducing power usage.	(Castellon et al., 2022)
3	Hash methods enable the secure transfer and integrity of certificate data. SHA-256 is preferred in verification due to its efficiency and security.	A hash algorithm function is executed on the certificate. Transaction verification can be conducted efficiently.	(Anwar et al., 2021)
4	Protocols rely on HMAC because it generates a safe message authentication code by combining a cryptographic key with the hash function.	Addresses the effectiveness and widespread application of HMAC in well-known protocols, such as IPSec. HMAC is commonly utilised in popular security protocols.	(Mousavi & Shakour, 2019)
5	Examines the significance and practical applications of hash functions.	A focus is placed on the significance of hash functions in the context of network security.	(Arvind K. Sharma & Dr.S.K. Mittal, 2019)

In summary, hash functions in HMAC applications emphasize their crucial role in preserving the authenticity and integrity of data. In addition, HMAC combines a secret key with a cryptographic hash function, offering a strong method for ensuring message integrity

and safeguarding against unauthorised modifications. For the purpose of data integrity and authenticity, HMAC utilises hash functions like SHA256 to generate unique signatures on data, enabling the detection of any modifications (Uriawan et al., 2024). The combination of a secret key and a hash function improves security by making it difficult for attackers to fake messages (Mousavi & Shakour, 2019). Reconfigurable HMAC-hash units can improve performance by combining different hash functions and increasing throughput for large messages (Khan et al., 2005;Khan et al., 2007). According to Jin and Wang (2014), enhanced hash functions can further improve message integrity and authentication while accommodating different security requirements. Although HMAC greatly enhances data security, it is still important to be alert about any weaknesses and make sure that cryptographic techniques are constantly changed to lower risks.

This chapter briefly explains about HDLs and FPGAs. HDL is a language to design digital logic for hardware implementation. Digital logic for hardware implementation can be designed using HDL. In this study, Verilog is used as a design concept. FPGAs are one of the most popular media to implement digital logic. A brief overview of FPGA is provided in this chapter as FPGA implementation is the focus of the research. Hence, literature reviews from other researchers are considered to obtain the efficient cryptographic hash function algorithm in terms of speed, area, and power.

2.2 Overview of Common Hash Functions

Hash functions are crucial in cryptography, serving as one-way functions that transform inputs of variable length into fixed-length outputs. They are vital for maintaining data integrity, ensuring authentication, and providing security across a range of cryptographic applications. Hash functions are specifically designed to be collision-resistant,

pre-image resistant, and second pre-image resistant, thereby safeguarding against numerous threats. These functions find extensive use in digital signatures, password protection, and blockchain technology, among various other uses (Mohammed et al., 2020).

2.2.1 MD5 Design

A well-known cryptographic hash algorithm that produces a 128-bit hash value is Message Digest 5 (MD5), which was developed in the early 1990s. Although it was intended for security applications, its weaknesses have raised concerns about its ability to protect sensitive data. Despite these limitations, MD5 remains popular due to its speed and low computational requirements, especially in low-power devices and applications that require little processing power.

A possible usage of MD5 is in population data management (Nistrina et al., 2024). In this case, only authorised personnel will be able to access critical population data owing to MD5's security features, thereby improving data management efficiency (Jayapandian et al., 2018). Furthermore, the approach is used to generate unique hash keys for digital documents, ensuring their integrity and authentication, as proven in a certificate issuance system (Albahadily et al., 2023). MD5 is an appropriate choice for low-power devices due to its increased speed and reduced processing capacity (Kumar et al., 2022). Nkouankou et al. (2022) discovered that MD5 is vulnerable to collision and pre-image attacks, which compromise its reliability for secure applications. Alternative algorithms are required because its security level is deemed inadequate for high-level data confidentiality (Horpenyuket al., 2024). Although MD5 is still used in many different applications, its inherent weaknesses require careful consideration and the search of more safe substitutes for

the security of private information. Table 2.2 presents a compilation of MD5 hash function in different applications.

Table 2.2 Compilation of MD5 Hash Function in Different Applications

No.	Hash Function	Overview	Finding	Reference
1	MD5	It implements the MD5 algorithm for account security in a population data management application to improve data management and prevent unauthorised access.	Eases the process of managing population data for village offices. Improves account security by employing the MD5 algorithm.	(Nistrina et al., 2024)
2	MD5	Constructs models for the computation of trigonometric coefficients in MD5.	Formulated a mathematical and software model for the growth of the MD5 sine function.	(Horpenyuk et al., 2024)
3	MD5	It uses the MD5 message digest method to secure digital signatures and authenticate digital documents and certificates.	Created MD5-secured digital signature mechanism. Verified digital certificate integrity.	(Albahadily et al., 2023)
4	MD5	MD5 method secures low-power consumption sensors and is used for data base portioning.	MD5 ensures security for low-power sensors. Enables database partitioning to protect sensitive data.	(Kumar et al., 2022)
5	MD5	Enhanced pre-image attack on the MD5 hash function utilising propositional logic. The Boolean satisfiability problem is solved with SAT solvers.	Study MD5 pre-image attack using SAT solver. SAT solvers solve faster with new encoding.	(Nkouankou et al., 2022)
6	MD5	Secure networks with efficient hash functions. The improved MD5 algorithm improves retrieval efficiency.	Optimise the internal structure of MD5.	(Zhuoyu & Yongzhen, 2022)

2.2.2 SHA-1 Design

For cryptographic purposes, a 160-bit hash value can be generated by the Secure Hash Algorithm 1 (SHA-1). It is a well-known method for verifying the authenticity and integrity of data, which was developed by the National Security Agency and published by the National Institute of Standards and Technology. A 160-bit hash value is the output size of SHA-1. Although its attributes have been undermined over time, they still offer pre-image resistance, second pre-image resistance, and collision resistance (Al-Nawashi et al., 2024; Wagner & Lugrin, 2023). The US government recognises it as a Federal Information Processing Standard (Al-Nawashi et al., 2024). This algorithm is employed in many different applications to confirm the integrity of files, such as documents and audio files (Nasution, 2022; Toruan, 2021). Additionally, to offer identity verification and non-repudiation services, it is frequently combined with digital signatures (Khan et al., 2022). There is a cryptographic flaw in SHA-1 that makes it insecure against contemporary threats (Al-Nawashi et al. 2024). The advent of SHA-2 and SHA-3 emphasises the necessity of stronger hashing standards to reduce security threats. Although SHA-1 has contributed much to cryptography, due to its weaknesses, new secure hashing algorithms are required, which reflects the changing nature of cybersecurity threats. The SHA-1 hash function is compiled across multiple applications, as shown in Table 2.3.

Table 2.3 Compilation of SHA-1 Hash Function in Different Applications

No.	Hash Function	Overview	Finding	Reference
1	SHA-1	Analysis and development of the SHA-1 algorithm - a comprehensive review of the algorithm's theoretical components and practical implementation.	A thorough examination of the SHA-1 algorithm's theoretical aspects. The demonstration of implementation through a variety of hash configurations.	(Al-Nawashi et al., 2024)

Table 2.3 Continued

2	SHA-1	The application of SHA-1 for the verification of audio file authenticity	SHA-1 determines the validity of audio files well. Prevents unauthorised alterations and fraud.	(Nasution, 2022)
3	SHA-1	Standardised hash functions are regarded as secure for a variety of cryptographic applications.	No substantial technological advancements in hash functions are anticipated.	(Mulder et al., 2023)
4	SHA-1	Includes background on the SHA hash algorithm family and how they have developed through time. Evaluates SHA-1, SHA-2, and SHA-3 in terms of their respective security, performance, and limitations.	SHA-1 raises serious security concerns. Newer algorithms might provide improved security and performance as technology develops.	(Khan et al., 2022)
5	SHA-1	Using the SHA-1 algorithm to ensure documents' authenticity, confidentiality, integrity, and authentication.	The SHA-1 algorithm improves document authenticity and integrity. Detects unauthorised changes to confidential documents effectively.	(Toruan, 2021)

2.2.3 RIPEMD-160 Design

Notable for its usage in combination with SHA-256, the ISO/IEC standard RACE Integrity Evaluation Message Digest 160 (RIPEMD-160) hash function is notable in the production of Bitcoin addresses. Despite the emergence of different collision attacks, RIPEMD-160 has proven resilient, with the most effective attacks only completing a fraction of its 80-step procedure. Furthermore, RIPEMD-160 employs a double-branch structure, which strengthens its security against collision attacks (Li et al., 2023). It uses a combination of logical functions and permutations to generate a 160-bit hash output after 80

steps of processing data. In recent development of RIPEMD-160, the emergence of mixed integer linear programming (MILP)-based approaches has facilitated the detection of differential traits, making these attacks more successful (Li et al., 2023). Even though RIPEMD-160 is still a strong hash function, the continued study of collision attacks emphasises how important it is to continuously assess cryptographic standards, particularly as computing power increases. Table 2.4 lists a compilation of RIPEMD-160 hash function in different fields.

Table 2.4 Compilation of RIPEMD-160 Hash Function in Different Applications

No.	Hash Function	Overview	Finding	Reference
1	RIPEMD-160	The method utilises MILP for the search of signed differential features.	Explores RIPEMD-160 hash function collision attacks. Strongly improves collision and semi-free-start attacks.	(Li et al., 2023)
3	RIPEMD-160	Recommended pre-image attacks on RIPEMD-160 and HAS-160. Attacks enhanced through intermediary measures and message padding.	Pre-image attacks have been proposed against RIPEMD-160 and HAS-160. Both of the hash functions have had their attack steps improved.	(Shen & Wang, 2018)
4	RIPEMD-160	Design on RIVYERA S6-LX150 performs well. Compared to PC and GPU, FPGA design saves energy.	Attained 245,000 passwords per second on multi-FPGA. FPGAs used 77 times less energy than PCs.	(Abbas et al., 2014)

2.2.4 SHA-256 Design

Data integrity and security are guaranteed by the well-known SHA-256 hash function, which computes a fixed-length hash from variable-length input dataClick or tap here to enter text.(Dewi et al., 2019; Fauziah et al., 2018). It is a dependable solution for many applications, including blockchain technology and password storage, thanks to

qualities such as collision resistance. Recent research has investigated related vulnerabilities and advancements, highlighting its importance in modern cryptography.

It is computationally impossible to identify two distinct inputs that produce the same hash, which is a critical security feature of SHA-256 (Alamgir et al., 2024). Besides, SHA-256 reliably generates the same hash for identical inputs, assuring data verification accuracy. Some applications of SHA-256 are routinely used to hash passwords, preventing unauthorised access. It also protects Bitcoin transactions by maintaining the integrity of transaction data (DiNardi & Radhakrishnan, 2023). New architectures, such as those based on binary tree topologies, have been proposed to improve processing speed by 23% on FPGA systems (Sy et al., 2023). Furthermore, SHA-256 has been successfully implemented into systems, such as QR identification to ensure secure attendance tracking (Sy et al., 2023). Although SHA-256 is strong, ongoing research identifies possible shortcomings, such as collision attacks and its efficacy (Alamgir et al., 2024). This emphasises the importance of continuously evaluating and improving cryptographic processes. Table 2.5 shows a compilation of RIPEMD-160 hash function in various applications.

Table 2.5 Compilation of SHA-256 Hash Function in Different Applications

No.	Hash Function	Overview	Finding	Reference
1	SHA-256	Satisfiability (SAT) solver for collision detection. SAT solvers are guided using a computer algebra system (CAS).	Implemented SAT + CAS SHA-256 collision solution. Found 38-step modified initialisation vector collision.	(Alamgir et al., 2024)
2	SHA-256	Blockchain data layer implementation employing SHA256 hashing algorithm.	Improved data integrity, confidentiality, and authentication on the blockchain network.	(Sondakh et al., 2024)

Table 2.5 Continued

3	SHA-256	Pipeline with multiple levels, utilising techniques of reusing registers and sharing adders.	Propose the B2$HA coprocessor utilising SHA-256, BLAKE-256, and BLAKE2s hash functions. Quick, energy-efficient, and versatile.	(Luan et al., 2023)
4	SHA-256	Multiple-level Verilog implementation analysis. Novel hardware architecture using binary tree adder trees.	A novel SHA-256 hardware architecture targets Intel DE10-Lite FPGA and speeds up hash processing by 23% using binary tree structured adder trees.	(DiNardi & Radhakrishnan, 2023)
5	SHA-256	A safe and efficient QR identification system with SMS notification using SHA-256.	QR identification scheme with SHA-256. SMS-based automated attendance tracking.	(Sy et al., 2023)

2.3 Design Methodologies of Hash Function

The hash function can be optimised in a variety of ways to make it faster, more efficient, and less power consumption. Iterative approaches, pipelining, unfolding, low-power designs, retiming, and resource sharing are several ways to optimise the hash algorithm. These techniques make hash function appropriate for security applications by improving performance, throughput, and efficiency of implementation.

2.3.1 Iterative

The iterative method is one of the conventional methods used to design hash functions. In other words, the data will be updated from the output data to the previous state. This condition will be repeated until the final round of each type of hash function. This method is good, simple, and requires low area usage as only one round function is executed at a time and has manageable power consumption. This is a conventional approach of

implementing hash functions, in which the algorithm sequentially processes data. It is challenging to parallelise because each round or compression function depends on the output of the previous round. This method may be slower than pipelined or unfolded implementations, particularly for high-throughput applications, due to its lack of parallelism. However, iterative processing reduces performance, as each block waits for the previous block to finish. This type of method is typically suitable for applications that prioritise low hardware complexity over high throughput (Sandeep Kumar, 2016).

2.3.2 Unrolling and Pipelining

By separating the hash function into several stages, a method called pipeline can process data in parallel, allowing fresh data to flow into each stage without blocking the processing of earlier data. By enabling simultaneous processing of multiple data blocks in distinct phases, the throughput is substantially increased. The necessity of registers and control circuitry for each stage results in a higher consumption of power and space in pipelines. This can be a trade-off on FPGA platforms, particularly when power is a concern. Furthermore, this technique is ideal for high-throughput applications, such as real-time data processing in secure communication systems, where latency and performance are of the highest priority (Goos et al., 2005; Jiang et al., 2009; Li et al., 2019; Sideris & Dasygenis, 2023; Y. Wang et al., 2010).

2.3.3 Unfolding

Unfolding repeats the logic of several rounds within a single clock cycle, allowing the hash function to execute numerous rounds at once. The number of cycles required to complete the hash computation is minimised using this strategy. Reducing the hash function's latency results in great performance. Unfolding is especially beneficial in latency-

sensitive applications. Unfolding requires a big area since hardware resources are replicated for each unfolded round, which might increase power consumption. This approach is effective for applications that require rapid hashing with low latency, such as cryptographic authentication systems in secure protocols (Jiang et al., 2009; Lee et al., 2008; Wang et al., 2010).

2.3.4 Low-power Design Approach

Due to its rapid implementation, hash function implementation on reconfigurable hardware is a crucial component nowadays. When it comes to implementing hardware, one solution is the FPGA. It is composed of several programmable interconnections linking tens of thousands of building blocks called CLBs (Rodrguez-Henriguez et al., 2006). The input output (I/O) block acts as a bridge between the external environment and the internal logic structure. Two ways to define FPGA-based digital logic design are schematic design and HDL. The main advantage of FPGA is its reconfigurability. Furthermore, it incurs low development cost and is easy for verification but suffers the disadvantage of high-power consumption. Therefore, two techniques for reducing power consumption in hash function designs have been proposed, namely state encoding and clock gating.

Low-power approaches optimise the hash function to use as little power as possible, frequently by minimising switching activity and making better use of logic gates. The techniques used in this research are clock gating and Gray encoding. Table 2.6 shows a simple example of encoding transition for seven states, for both binary and Gray encoding. From this table, the maximum transition per clock cycle for Gray encoding is one, while for Binary encoding is three. This is because Gray encoding only changes 1-bit per state during transition. Gray encoding reduces switching activity by assuring that only 1-bit changes

between successive states, which is very useful in state machines. By applying this technique to finite state machine design, power consumption can be reduced.

Table 2.6 State Encoding

State	**Gray**	**Binary**
S0	000	000
S1	001	001
S2	011	010
S3	010	011
S4	110	100
S5	111	101
S6	101	110
S7	100	111
Maximum transition per clock cycle	1	3

Clock gating reduces dynamic power by blocking the clock signal in inactive areas. Clock gating can be used to lower power consumption. Reduced power consumption is another benefit of clock gating, which occurs when designs only execute operations when input is available; this input is ANDed with the clock input to form clock gating (Zhang et al., 2014). A clock gating example is shown in Figure 2.1. Its lower power consumption makes it ideal for battery-powered and energy-sensitive applications. This method is essential in IoT, mobile and portable devices, as well as FPGAs, where power efficiency is critical, such as research into dynamic power reduction.

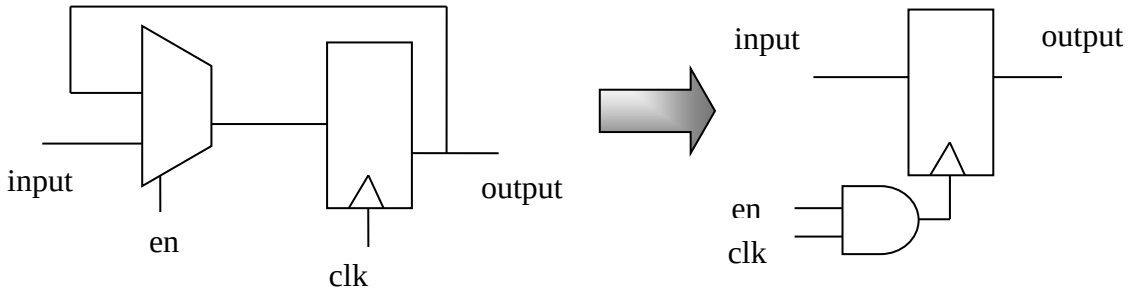


Figure 2.1 Clock Gating (SemiEngineering, n.d.)

2.4 HMAC Overview and Comparative Analysis of Hash Function

Hash-based message authentication code (HMAC) is a commonly employed cryptographic method that guarantees the integrity and validity of messages (Mousavi & Shakour, 2019). It adds a one-of-a-kind code to the message by combining a cryptographic key with a hash function. To make sure the message has not been changed, the recipient can check this code. Transport layer security and other secure communication protocols rely on HMAC's security, particularly in applications such as internet protocol security (IPSec), where it is crucial for safeguarding data integrity and authenticity (Mousavi & Shakour, 2019). The HMAC is generated or verified only by those with the key; hence, the procedure hashes the message together with the key. The robustness of the hash function used determines the degree of security offered by HMAC. Stronger hash functions, such as SHA-256, are favoured due to their superior security attributes (Anwar et al., 2021;Sharma & Mittal, 2019).

Furthermore, HMAC is an excellent method for maintaining message integrity, but its efficacy depends significantly on the underlying hash function's quality and the cryptographic key's confidentiality. As cryptographic needs change, HMAC must be constantly improved and adapted to meet emerging threats. Although hash functions play a crucial role in HMAC, it is important to consider how security and performance could be

compromised, particularly in situations with limited resources where energy efficiency is vital. Recent studies suggest improvements to HMAC efficiency by decreasing computational overhead while preserving security, which is crucial for applications in network coding (Uriawan et al., 2024).

2.4.1 Properties of Cryptographic Hash Functions

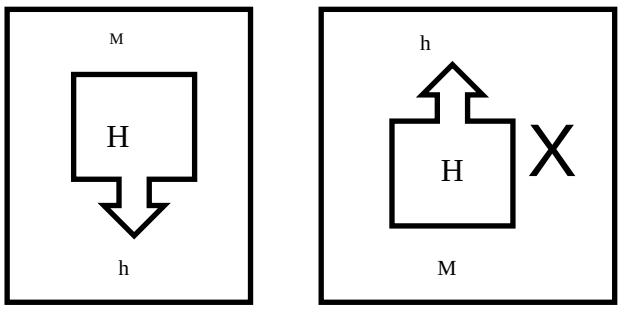
A hash function is characterised by its one-way nature, its output having a set length, and its resistance to collisions (Wagner & Lugrin, 2023). It is known as a one-way function because it is impossible to utilise the output to obtain the initial input. In other words, hash functions are designed to be irreversible. Different kinds of hash functions provide different lengths of output. For instance, SHA-256 produces a hash value of 256 bits. The condition when it is computationally impossible to identify two distinct inputs that give the same hash output is also known as collision resistance (Wagner & Lugrin, 2023). All these characteristics make hash functions secure and appropriate for carious cryptographic applications.

There are numerous applications in cryptography, such as password hashing, digital signature, and data integrity (John & Jose, 2023). Here, the idea is to check if data has not been altered during transmission. Although hash systems are generally seen as safe, the rise of lightweight cryptography emphasises the need of effective algorithms fit for devices with little resources, suggesting a possible change of direction for future innovations (John & Jose, 2023).

Cryptographic hash functions serve as critical elements in contemporary security protocols, offering a range of characteristics that guarantee data integrity and authenticity. As a digital fingerprint, the functions transform inputs of varying lengths into consistent

outputs of set lengths. Collision resistance, second pre-image resistance, and pre-image resistance are the key features of cryptographic hash functions. These properties are important for function effectiveness in various applications, including digital signatures and message authentication.

Collisions occur when two messages generate identical message digests. The hash function does not exhibit collision behaviour. Thus, hash function, H is strong enough to withstand collision. It is useful for digital signature algorithm and message authentication. There are several properties of hash function. It works with any block message and returns an output of a set length. Furthermore, it is easy to compute the hash function for every message. Therefore, it is practical for both software and hardware platforms.



(a) **(b)**

Figure 2.2 (a) and (b) Pre-image Resistance or One-Wayness

Pre-image resistance, also known as one-wayness; second pre-image resistance, or weak collision resistance; and collision resistance, or strong collision resistance; are the three central criteria that a hash function must have. The requirement of hash function is illustrated in Figure 2.2 (a). Based on this figure, input message, M produces the output hash, h, but it cannot reverse the process of hash to obtain message M, as shown in Figure 2.2 (b). Hence, it is referred to as one-wayness property of hash function. Furthermore, second pre-image resistance is computationally infeasible to determine y, if there is given input x with

hash code, $h(x) = h(y)$. It means that it is technically impossible to find another input message with the same hash value, as presented in Figure 2.3.

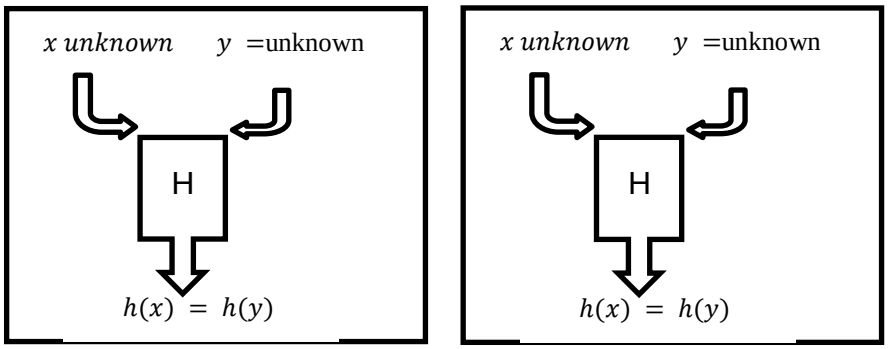
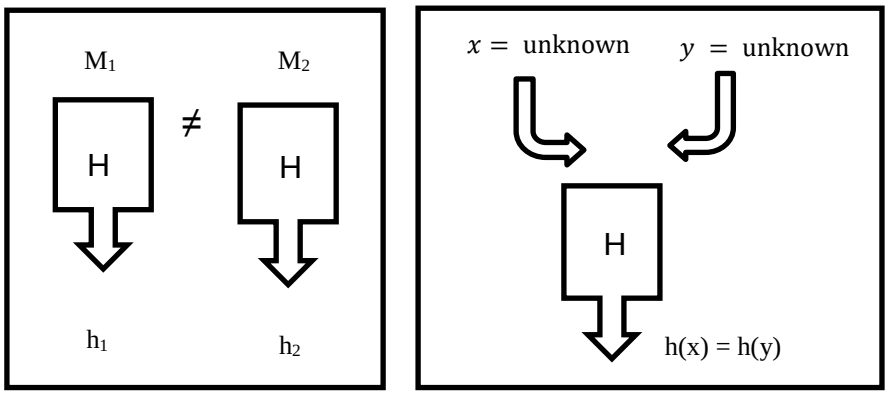


Figure 2.3 Second Pre-image Resistance or Weak Collision Resistance



(a) **(b)**

Figure 2.4 (a) and (b) Collision Resistance or Strong Collision Resistance

Also, the output hash code h(x)= h(y), $h(x) = h(y)$ which is computed for any two message inputs x and y, is computationally impossible. As illustrated in Figure 2.4 (a) and (b), this state denotes having a high level of collision resistance.

Pre-image resistance, second-pre-image resistance, and collision resistance are three important features of cryptographic hash functions. Due to the pre-image resistance, finding an input that generates a hash output is computationally impossible. This is essential for the preservation of data security and confidentiality (Wagner & Lugrin, 2023). Second, pre-image resistance makes it difficult to find a different input that produces the same hash value

as the current input. This prevents attackers from replacing genuine data with harmful data (Wagner & Lugrin, 2023).

If two distinct inputs that yield the same output from a hash function cannot be identified, then it is said to be collision resistant. This property is vital for maintaining the uniqueness of hash outputs, which is crucial in applications like blockchain (D. Sharma & Saxena, 2023). The fundamental properties of security are crucial, but the design of hash functions must also consider performance and efficiency, particularly in resource-constrained environments, as emphasised in lightweight cryptography (John & Jose, 2023). Lightweight cryptography is the field of cryptographic methods and systems intended to ensure data security in low-power FPGAs, embedded systems, RFID tags, IoT devices, and low-resource conditions.

2.4.2 Comparative Analysis of Hash Functions

As one-way mechanisms that convert inputs of different lengths into outputs of a fixed size, hash functions are essential to cryptography. They are essential for various applications, such as securely storing passwords, creating digital signatures, and verifying data integrity. The effectiveness of these functions in cryptographic frameworks is dependent on properties such as pre-image resistance, second pre-image resistance, and collision resistance.

Hash function, H is a non-linear function that transforms fixed binary chains of arbitrary length to fixed length; it takes all message bits as input. Let M be a message of an arbitrary length and returns a fixed-length value, h. Hash codes are the results of hash functions. Hash value or message digest are other possible meanings. The structure of a hash function is shown in Figure 2.5. Hash function is an efficient computational function that

does not rely on a specific key. The output hash code depends on the input message; any change of the bits message will change the hash code. Message digests with short bit lengths are the strength of modern hash algorithms (Franciso R. H. et al., 2006).

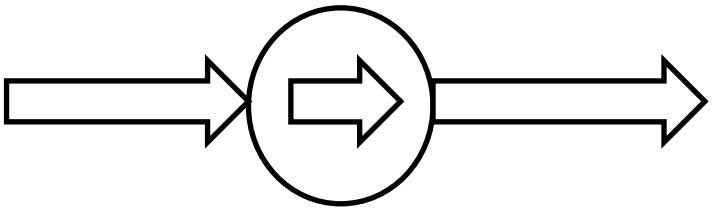


Figure 2.5 Hash Function

The implementation of cryptographic hash algorithms like SHA-1, MD5, RIPEMD-160, and SHA-256 on FPGA systems utilising Verilog HDL has received significant attention because of their essential purpose in ensuring data integrity and security across various applications. Every algorithm presents distinct attributes and performance compromises when implemented in hardware, especially on FPGA architectures that provide parallel processing and reconfigurability. FPGA realisations are ideal for embedded systems and time-sensitive settings because they can provide noticeably better throughput, lower latency, and enhanced resource efficiency when compared to software-based implementations.

FPGAs consist of many architectural types that affect design decisions and performance results. For instance, high-performance FPGA families like Xilinx Virtex and Intel Stratix have a lot of logic parts, special DSP blocks, and fast transceivers, which allow for more sophisticated cryptographic pipelines with larger frequency margins. Conversely, mid-range devices such as Intel Arria and Xilinx Kintex offer a balance of cost and performance, delivering sufficient resources for moderately complicated designs while preserving lower power consumption. Economical families like Intel Cyclone or Xilinx

Artix are targeted for applications with strict area and budget limits, while they impose restrictions on attainable performance and accessible memory.

Although FPGAs offer flexibility, their implementations present issues concerning resource allocation, timing closure, and dynamic power consumption. Lightweight design approaches, including pipelining, unfolding, Gray encoding, and clock gating, are utilised to reduce these issues by decreasing switching activity and enhancing throughput within the device's architectural constraints. The justification for choosing FPGA technologies and lightweight techniques is covered in this part, which also examines earlier research that looked into FPGA-based cryptographic hash function designs for HMAC applications. The reported performance enhancements, the architectural factors unique to various FPGA families, and typical challenges faced during synthesis, optimisation, and deployment are also highlighted.

2.4.2.1 MD5 Design

Several studies have implemented MD5 on FPGA. Some of the researchers implemented the MD5 design using two different types of FPGA devices, which are Altera and Xilinx CAD tools. An overview of several papers regarding MD5 is discussed briefly in this chapter. A throughput of 586 Mbps was achieved with logic requirements of only 647 slices and 2 BlockRAMs when Jarvinen et al. (2005) developed MD5 using a Xilinx Virtex-II XC2V4000-6 FPGA. Xilinx was chosen because it provides large logic resources and produces very fast performance. The MD5 algorithm's structure, suggested by Jarvinen et al. (2005), permitted both iterative and pipelined implementations. Pipelining can reduce delay and utilise more logic area implementations. An approach to increase the performance of the design is by using pipelined architecture and the use of BlockRAMs. It was discovered that

hardware with extremely few logic resources could efficiently implement the MD5 algorithm.

A previous study introduced two types of MD5 hash algorithms that were implemented using VHDL on Virtex V1000FG680-6 with a clock rate of up to 200 MHz, namely iterative and full loop unrolling (Deepakumara et al., 2001a). Based on this paper, the designs were assessed according to their maximum frequency and throughput. The throughput and maximum frequency for iterative design were 165 Mbps and 21 MHz, respectively, while the throughput and maximum frequency for full loop unrolling were 354 Mbps and 71.4 MHz, respectively.

FPGA implementation of hash function for cryptographic protocol is also one of the solutions to the cryptographic protocol issue (Díez et al., 2002). The authors proposed two types of cryptographic algorithms designed in their study: MD5 and SHA-1. A maximum frequency of 60.2 MHz and a throughput of 467.3 Mbps were achieved using the MD5 architecture executed on the Virtex 2V3000.

High-throughput design is important for data transmission. Therefore, FPGA implementation with high throughput needs to be considered. Wang et al. (2010) conducted an implementation of the MD5 hash function algorithm on FPGA with ultra-high throughput. Based on the study, the maximum frequency of iterative MD5 design was 102.7 MHz, with 810 Mbps of throughput of the design. This design used Altera Stratix II EP2SGX90FF. The throughput of the MD5 design with 32 stages of pipelining with unfolding was increased by Wang et al. (2010) to 32.03 Gbps. He & Xue (2010) used Cyclone II for MD5 implementation, producing 4.4 Gbps throughput. No information regarding speed, area, or

throughput was provided when (Noaman, 2013) and (Landge & Mishra, 2016), who used VHDL to design MD5. Besides, Zhong et al.(2016) designed MD5 using Java.

There are several other FPGA family devices, such as Stratix for high-performance FPGAs and Cyclone for low-cost FPGAs. Hence, to obtain a balance between power and performance, the Arria II GX device has been chosen because the Arria series provides a low-cost transceiver for FPGA devices. Moreover, the device also delivers optimal performance and power efficiency. The throughput of the MD5 design increases significantly due to the latency of the design. The latency for MD5 iterative design is 66 cycles. Meanwhile, the latencies of the MD5 unfolding design and MD5 unfolding with pipelining are 33.5 and 34.5, respectively.

Altera and Xilinx FPGAs, among others, have been used for MD5 designs. The throughput of the MD5 design obtained by Deepakumara (2001) and Diez et al. (2002) was 165 Mbps and 467.3 Mbps, respectively, and the MD5 designs used Virtex V1000FG680-6 and Virtex 2V3000, respectively. The implementation by Jarvinen et al. (2005) using Virtex-II achieved a throughput of 607 Mbps and a maximum frequency of 78.3 MHz. The iterative design achieved a maximum frequency of 102.9 MHz and a throughput of 810 Mbps according to the MD5 design by Wang et al. (2010). This design used Altera Stratix II EP2SGX90FF. The throughput of the MD5 design with 32 stages of pipelining with unfolding was increased by Wang et al. (2010) to 32.03 Gbps. He and Xue (2010) used Cyclone II for MD5 implementation and gave 4.4 Gbps throughput. The preceding MD5 hash function designs are summarised in Table 2.7 according to the highest frequency, area, and throughput that could be achieved using different methods. This compilation provides a comparative analysis of the methods used in designing MD5 design. It shows that pipelining

and unfolding transformations can enhance and improve the performance of MD5 design. Besides, this method can also provide small-area implementation, as well as high speed.

Table 2.7 Compilation of MD5 Hash Function Design

No.	Design Method	FPGA Device	Authors/ year	FMax (MHz)	Throughput (Mbps)	Area	Power (mW)
1	MD5 iterative (high/low adder)	Altera (no device mentioned)	(Fan et al., 2023)	133.6	1068.8	2,334	-
2	MD5 iterative by encoding ribonucleic acid	(No software/device mentioned)	(Mohammed Ali & Kadhim Farhan, 2020)	-	-	-	-
3	MD5 iterative with pseudorandom number generator	Altera Cyclone II EP2C35F672C6	(Tian et al., 2018)	50	-	4,895	-
4	MD5 iterative	-ModelSim -No device mentioned	(Landge & Mishra, 2016)	Simulation time of 6,900 ns	-	-	-
5	MD5 iterative	Software	(Zhong et al., 2016)	-	-	-	-
6	MD5 iterative	Xilinx Spartan 3-xc3s1400a-4fg484	(R. Saeed, 2016)	-	806.596	-	-
7	MD5 Iterative	Xilinx IC XC3S1400A	(Noaman, 2013)	-	-	-	-
8	MD5 iterative (Application OTP)	Software	(Sediyono et al., 2013)	-	-	-	-

Table 2.7 Continued

9	MD5 iterative (12 MD5 multi-parallel architecture)	Altera Cyclone II EP2C35F672C6	(D. He & Xue, 2010)	48.2	4,300 (3,343)	30,134	-
10	MD5 iterative	Altera Stratix II GX EP2SGX90FF1508C3	(Y. Wang et al., 2010)	102.7	810	462	-
	MD5 pipeline (4p)			97.95	3,039	3,484	-
	MD5 pipeline (32p)			86.32	21,428	25,767	-
	MD5 pipeline (32p) - unfolding factor 2			66.48	32,035	267	-
11	Unified MD5, SHA-1, RIPEMD-160 (iterative)	ASIC 0.18u	(Y. K. Lee et al., 2008)	584.8	3,651	19,767	-
	Unified MD5, SHA-1, RIPEMD-160 (unfolding)			299.4	3,738	23,167	-
12	MD5 iterative	Xilinx Virtex-II XC2V4000-6	(Jarvinen, Tommiska, & Skytta, 2005)	75.5	586	647	-
	MD5 full pipelining			93.4	725	7,997	-
13	MD5 iterative	Xilinx Virtex 2V3000	(Díez et al., 2002)	60.2	467.3	1,369	-
14	MD5 iterative	Xilinx Virtex	(Deepakumara et al., 2001)	21	165	880	-
	MD5 full loop unrolling			71.4	354	165	-

2.4.2.2 SHA-1 Design

There are several other publications of SHA-1 implementation based on FPGA. Jarvinen (2004) focused on the implementation of SHA-1 on Xilinx Virtex-II XC2V2000-6. The structure of SHA-1 was based on the iterative structure that could be used in

combination with MD5 design in the future. The design was written in VHDL codes, and it provided a good balance between performance and area implementation. The maximum frequency of the design was 117.5 MHz, while the throughput of the design was 734 Mbps. For the area implementation, the design utilised 1,275 slices and the equivalent gate count was 25,467.

Kang et al. (2002) implemented three different types of hash function (SHA-1, HAS-160, and MD5) in a single chip. One of the main objectives of this design was to minimise the use of multiple gates. Using a speed grade 3 Altera EP20K1000EBC652-3, all designs were synthesised, placed, and routed. The maximum frequency of SHA-1 based on a hash processor with 81 clock cycles was 18 MHz, while the throughput of the design was 114 Mbps. In this design, the SHA-1 module was combined with the HAS-160 module into a one-unit module. Thus, the utilisation of the logic element for resource sharing processor was 10,573. The proposed architecture offered an effective solution for the implementation of an IPSEC chip (Kang et al., 2002).

The reconfigurable cryptographic chip is applicable for encryption and decryption processes. It can implement a diverse array of algorithms with agility and (Miao et al., 2009). They proposed five different types of hash functions that had been implemented on Stratix II-EP2S15F484C3, namely SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512. To boost the speed of message (W_t generation), the reconfigurable design incorporated carry save adder (CSA) and carry lookahead adder (CLA) in the data channel. Based on the findings, the architecture could support a maximum frequency of 171.821 MHz with a total of 874 registers and 1,131 ALUTs. The throughput of the design was 727.853 Mbps.

Cryptographic algorithms implemented on reconfigurable hardware, like FPGA, showed promise as an alternative to the more conventional hardware implementation. It provided powerful mechanisms to protect data(Díez et al., 2002). Two basic techniques were implemented: loop unrolling and parallelisation of resources and pipelining. High throughput is an advantage of the pipelined architecture. The researchers designed both MD5 and SHA-1 architectures on Xilinx Virtex 2V3000. For area implementation, SHA-1 used 1,550 slices which were equivalent to 29,900 gates. The maximum frequency of SHA-1 design was 38.6 MHz with a throughput of 899.8 Mbps.

The internet protocol security (IPSEC) protocol has specified a number of algorithms, one of which is the FPGA implementation of the SHA-1 algorithm (Dai & Zhou, 2003). They proposed the FPGA implementation design, which was divided into three parts: function block diagram, expansion block, and data path. A 3-to-2 CSA was used, followed by a carry propagate adder. The designs were synthesised, placed, and routed on a EP1K100QC208 target device. The results achieved 1622 of logic elements. with a maximum frequency of the design and a throughput of 43.08 MHz and 268.99 Mbps, respectively.

Altera Stratix II GX EP2SGX90FF FPGA has successfully implemented the SHA-1 hash algorithm, resulting in a high throughput of up to 76.195 Gbps (Jiang et al., 2009). A pipeline architecture of the design leads to an increment of the throughput. Jiang et al. (2009) implemented four different types of SHA-1 designs: three iterative and one loop unrolling. The three iterative designs were divided into three parts: iterative, iterative with four pipelines, and iterative with 40 pipelines. The maximum frequency of the iterative design was 253.07 MHz, while for loop unrolling, the maximum frequency was 163.70 MHz. Even

though the results showed a high throughput, it utilised large area implementation of about 33,764 ALUTs. The total number of registers and ALUTs for the iterative design was 32,048 and 947, respectively.

Hash functions belong to one of the most important categories of encryption algorithms. They are widely used in security applications and have been included in the specifications of communication protocols (Sklavos et al., 2003). In this paper, Sklavos et al. (2003) introduced two types of hash functions that have been implemented based on FPGA: MD5 and SHA-1. The designs focused on pipelining architecture with a four-step design for every 20 rounds. A Xilinx FPGA device was used to implement, place, and route the designs written in VHDL. The results showed 1,578 slices with an operating frequency of 72 MHz. This increased the throughput to 1.7 Gbps.

Lee et al. (2006) examined the throughput of an optimised SHA-1 architecture utilising unfolding transformation. This strategy lowered the needed cycles for a single block hash. They proposed the implementation of SHA-1 architecture based on FPGA and ASIC implementation. The optimal results of SHA-1 based on FPGA implementation (Virtex2 XCV1000) was 12 cycles without pipeline, while for the design with pipeline, the optimal result was 24 cycles. The designs were written in VHDL. The results for 12 cycles showed a maximum frequency of 20.9 MHz with 893 Mbps of throughput.

Kim et al. (2012) reduced the design's critical path by concentrating on loop unfolding and pre-processing to execute two hash operations. Xilinx ML605, an FPGA based on the Virtex-6 LX240T, was used to implement the designs. The results showed that the maximum frequency for the original design was 165.2 MHz, while for the unfolding design with 40 cycles, it was 142.6 MHz. To obtain a high throughput, the pipelined

architecture of the design was proposed. The results showed that the proposed SHA-1 designs needed 1,649 slices to reach a throughput of 7.35 Gbps at 150.45 MHz. The previous SHA-1 hash function designs are summarised in Table 2.8, which lists the maximum frequencies, areas, and throughputs achieved using different methods. This compilation provides a comparative analysis of the methods used in designing SHA-1. It shows that pipelining and unfolding transformations can enhance and improve the performance of SHA-1 design. Besides, this method can also provide small-area implementation and high speed.

Table 2.8 Compilation of SHA-1 Hash Function Design

No.	Design Method	FPGA Device	Authors/ year	FMax (MHz)	Throughput (Mbps)	Area	Power (mW)
1	SHA-1 iterative (high/low adder)	Altera (no device mentioned)	(Fan et al., 2023)	171.821	1,099.65	874	-
2	SHA-1 iterative	Xilinx 14.0 (no device name)	(Surya Prashanth et al., 2021)	85.815	-	4,656	-
3	SHA-1	Xilinx Virtex-6 LX240T	(Makkad & Sahu, 2017)	213.14	1,347.2	449	-
	SHA-1 unfolding (factor 2)			154.35	1,927.5	518	-
	SHA-1 pre-computation			207.90	2,596.2	546	-
	SHA-1 four-stage pipelining			172.32	8,607.6	1230	-
4	SHA-1 iterative	Implementation of SHA-1 algorithm on FPGA	(Sandeep Kumar, 2016)	-	-	1,029	118.69

Table 2.8 Continued

5	SHA-1 iterative	Parallelising SHA-1	(Lee et al., 2015)	165.207	1,057	1,242	-
	SHA-1 loop - unfolding			142.633	1,825	1,270	-
	SHA-pre-processing			161.212	2,013	1,250	-
6	SHA-1	Xilinx Virtex-6	(Kim et al., 2012)	165.2	1,057	-	-
	SHA-1 unfolding			142.6	1,825	-	-
	SHA-1 pre-processing			161.2	2,013	-	-
	SHA-1 pipeline (4p)			150.7	7,351	1,649	-
7	SHA-1 pipelining	Altera Stratix III EP3SE260F1 517C4	(Qiong & Jianwu, 2012)	270	13,824	-	-
8	Multiple SHA-1	Xilinx Virtex-5 XC5VSX95T	(Liu et al., 2012)	-	42,700	-	-
9	SHA-1	ASIC	(Dochert y & Koelman s, 2011)	100	162	6,836	-
10	SHA-1	Cyclone EPIC6Q240C 8	(Zhou & Liu, 2011)	59.4	-	1,560	-
11	SHA-1 unfolding + delay bound	Xilinx Virtex-2 XC2V1000	(Lee et al., 2011)	84	935	-	-
	SHA-1 unfolding + delay bound			36.5	1,170	-	-
12	SHA-1 iterative	Altera Stratix EP1S80F150 8C7	(Cheng & Deng, 2010)	-	-	1,584	-

Table 2.8 Continued

13	SHA-1 pipelining (4p)	Xilinx Virtex-2 XC2v1000	(Lee et al., 2009)	98.3	5,032.9	3,442	-
	SHA-1 unfolding factor 2 and pipelining (40p)			98.3	50,329	28,741	-
14	SHA-1 iterative	Altera Stratix II GX EP2SGX90FFF	(Jiang et al., 2009)	253.07	782	1,561	-
	SHA-1 pipelining (4p)			341.16	3,054	5,880	-
	SHA-1 pipelining (40p)			212.40	29,756	51,785	-
	SHA-1 unfolding and pipelining(40p)			163.70	32,048	76,195	-
15	SHA-1 iterative with CSA	Altera Stratix II EP2S15F484 C3	(Miao et al., 2009)	105.664	727.853	2,124	-
16	SHA-1 unfolding factor 2	Xilinx Virtex-2 XC2V1000	(Lee et al., 2009)	118	5,900	2,894	-
17	SHA-1 Partial unrolling	Xilinx Virtex-II	(Michail et al., 2009)	91	4,715	1,212	-

2.4.2.3 RIPEMD-160 Design

In recent years, many researchers have involved in designing optimised RIPEMD160 with high maximum frequency and high throughput (Dobbertin et al., 1996; Dominikus, 2002; Ganesh et al., 2007; Knežević et al., 2008; Michail et al., 2008; Ng et al., 2004; Satoh & Inoue, 2001; Sklavos & Koufopavlou, 2005). An optimisation for RIPEMD-160 on Xilinx

Virtex-E with a maximum frequency of 87.6 MHz and a throughput of 2,803 Mbps was performed. The proposed design keeps the area small enough using five-stage pipeline for each round. Operating frequency and delay are both improved by optimising the critical path. The design consists of two-unit blocks, such as pre-computation and final calculation, to obtain the output. Sklavos & Koufopavlou (2005) proposed the implementation of a processor FPGA, which obtained a throughput of up to 2 Gbps. A pipelined design with five stages of transformation and two parallel data streams formed the basis of this design. The Xilinx FPGA device (v200bg256) was used for placing and routing the code written in VHDL. The proposed processor, which made use of both RIPEMD-128 and RIPEMD-160, could operate at a maximum frequency of 67 MHz.

A unified solution for three distinct hash functions namely MD5, SHA1, and RIPEMD160 was introduced by Ganesh et al. (2017) and is known as HashChip. By using resource sharing, the area implementation of hash function can be reduced. Ng et al., (2004) also introduced a unified design for both MD5 and RIPEMD160. Any time an application requires a different hash algorithm; a unified design is the most effective way to implement them. To unify the two algorithms, it is necessary to make the following adjustments: increase the length of the chaining variable to 160 bits by adding the multiplexer and 10-bit cyclic shifter that are necessary for the design of RIPEMD-160. Altera EPF10K50SBC356-1 was used to implement the design that was done in VHDL. The design could only handle frequencies up to 26.66 MHz. The amount of memory bits was 5,376 (13%), whereas the number of logic cells used was 1,964 (68%). It is imperative that cryptographic hash algorithms are crucial in the field of cryptography. Therefore, by designing various types of hash functions, the comparison in terms of speed and throughput for each design can be analysed. An integrated device that could execute various MD4 hash algorithms was

proposed by Dominikus (2002). Various components of the MD4 family such as MD5, RIPEMD-160, SHA-1, and, most recently, SHA-256 were detailed in this article. It used VHDL for construction, and the calculation of their compression function was different. The design was divided into two parts: calculation of temporary t, which consisted of arithmetic and logic functions, and new assignment of the working variables, which was called update. Satoh and Inoue (2001) developed MD5, RIPEMD-160, and SHA-1/224/256/384/512, implementing them with a 0.13 μm CMOS standard cell library. All of these hash function architectures used CLA and CSA for their implementation. Knezevic et al. (2018) presented two designs: a design without an optimised loop and a design with an optimized loop, resulting in throughputs of 568 Mbps and 624 Mbps, respectively.

There are several other FPGA family devices, such as Stratix for high-performance FPGAs and Cyclone for low-cost FPGAs. Hence, to obtain balance of power and performance, Arria II GX device was chosen because Arria series provide a low-cost transceiver of FPGA device. Besides, it also delivers optimal performance and power efficiency. Moreover, the architecture of FPGA device also plays important roles in the RIPEMD-160 design. By selecting an appropriate FPGA device, the performance of RIPEMD-160 design can be enhanced. Table 2.9 presents the previous implementation of RIPEMD-160 design. It is difficult to find the same device for the same design due to the limitation of budget and devices. The table compiles prior RIPEMD-160 hash function designs according to the maximum frequencies, areas, and throughputs. This compilation provides a comparative analysis of the methods used in RIPEMD-160 design. Based on other hash function designs, it shows that pipelining and unfolding transformations can enhance and improve the performance of RIPEMD-160 design. In addition, this method can also provide small-area implementation and high speed.

Table 2.9 Compilation of RIPEMD-160 Hash Function Design

No.	Design Method	FPGA Device	Authors/ year	FMax (MHz)	Throughput (Mbps)	Area	Power (mW)
1	RIPEMD-160 iterative	HashCat software and GPU	(Kuznetsov et al., 2019)	-	-	-	-
2	RIPEMD-160 iterative	Xilinx Virtex	(H. Michail et al., 2010)	68.2	2,180	2,057	-
		Xilinx Virtex-E		86.6	2,750	2,256	-
		Xilinx Virtex-II		94.6	3,020	2,548	-
3	RIPEMD-160 pipelining	Xilinx Virtex	(Michail et al., 2008)	51.6	1,650	1,798	-
		Xilinx Virtex-E		64.6	2,080	1,856	-
		Xilinx Virtex-II		74.6	2,380	1,985	-
4	RIPEMD-160 iterative with CSA	Xilinx XC2VP30	(Knežević et al., 2008)	100.05	625	4,410 LUTs	
5	2008	Xilinx Virtex-E	(Michail et al., 2008)	87.6	2,803	-	-
6	MultiHash: SHA-1, MD5, and RIPEMD-160 (iterative multi-cycle)	Xilinx X2) XC2VP30FF896-7	(Ganesh et al., 2007)	104.875	203	2,837	-
7	Unified RIPEMD-160, SHA-1, and MD5	Xilinx Virtex XCV300E	(Khan et al., 2005)	37	116.94	3584	637

Table 2.9 Continued

8	RIPEMD-160 processor iterative	Xilinx Virtex XCV300E	(Sklavos & Koufopavlou, 2005)	67	2,017	-	-
9	RIPEMD-160 iterative	Altera EPF10K50	(Ng et al., 2004)	26.6	84	1,964	-
10	RIPEMD-160 iterative	Xilinx 2V500fg456	(Sklavos et al., 2003)	55	1,656	2,245	-
11	RIPEMD-160 iterative	FPGA Virtex XCV300E	(Dominikus, 2002)	42.9	65	1,004 CLBs	
12	RIPEMD-160 iterative	ASIC (0.13 um CMOS)	(Satoh & Inoue, 2001)	270.3	1,442	24,775 gates	-
13	RIPEMD-160 iterative	Software (Assembly)	(Dobbertin et al., 1996)	-	39.8	-	-
		Software ©		-	19.3	-	-

2.4.2.4 SHA-256 Design

There are four distinct hash functions that make up the SHA-2 hash function: SHA-224, SHA-256, SHA-384, and SHA-512. These hash algorithms provide outputs with lengths varying from 256 bits to 512 bits, which is determined by the SHA-2 length. In this study, the design focuses only on SHA-256 algorithm because of the extension from SHA-1 algorithm. Several SHA-256 designs have been proposed (Ahmad & Das, 2005; Chaves et al., 2006; McEvoy et al., 2006a; Mestiri et al., 2015; Miao et al., 2009; Sharif et al., 2011; Sklavos & Koufopavlou, 2003, 2005; Sun et al., 2007), depending on different types of FPGA devices. The SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512 cryptographic chip was developed and implemented on the EP2S15F484C3 Stratix II device by Miao et al. (2009). The design was written using Verilog code. Modelsim 6.0c simulator tool was

used to simulate and verify the design. According to these findings, SHA-256 offered a maximum frequency of 143.16 MHz and a throughput of 909.8 Mbps. To improve the execution performance of SHA-256, this design integrated the CSA and CLA. A SHA-256 design with a maximum frequency and throughput of 83 MHz and 326 Mbps, respectively, was proposed by Sklavos and Koufopavlou (2003). This design was implemented on Xilinx Virtex v200pq240. From this result, the maximum frequency of SHA-256 algorithm decreased slightly compared to the previous design. Digital signature algorithms, keyed-hash message authentication codes, and random number generators can all benefit from this design.

Two variants of the SHA-2 algorithm, SHA-256 and SHA-512 were proposed by Ahmad and Das (2005). These algorithms were designed and tested based on Altera Quartus II CAD tool. This work did not mention specific programming that was used to design and implement the SHA-256 algorithms. The designs were evaluated and integrated using Verilog HDL and VHDL, and subsequently installed and routed in Altera APEX II, Stratix, and Mercury family FPGAs. The maximum frequency of SHA-256 designs was 41.97 MHz, which reduced decrease significantly compared to other designs. Xilinx Virtex II xc2v2000-bf957 was used to implement six SHA processors designed by McEvoy et al. (2006a) using VHDL. Three processors were used for SHA-256: one with a 2x-unrolled core, one with a 4x-unrolled core, and one with a basic quasi-pipelined core. To make the computations in the SHA core faster, these methods have been suggested. The SHA core's long critical path, unrolling, quasi-pipelining, BlockRAM, and parallel counter are all part of the CSA. From the results, unrolled SHA-256 provided low latency compared to the basic design. However, the basic SHA-256 design gave 133.06 MHz of maximum frequency with a throughput of 1,009 Mbps.

The efficiency of SHA-2 algorithm was enhanced by Chaves et al. (2006). Xilinx VIRTEX II Pro (XC2VP30-7) was equipped with SHA-256, which achieved a maximum frequency of 174 MHz. In order to save space and resources, this design suggested rescheduling techniques and hardware reutilization methods for a pipelined structure. The results showed that the maximum frequency of SHA-256 was 174 MHz and a throughput of 1,370 Mbps. Sun et al. (2007) proposed SHA-2 (256, 384, and 512) designs, and the algorithms were written using Verilog code. ModelSim 6.0a was utilised for designing and simulating the algorithms. The target FPGA device for these designs was similar to the device used in Sklavos & Koufopavlou (2003), which was Xilinx Virtex v200pq240-6. The design of SHA-256 was based on CLA and CSA. The goal of this method was to enhance the design's critical path. The results showed that the maximum frequency of SHA-256 decreased significantly, with only 74 MHz and a throughput of 291 Mbps.

Sharif et al. (2011) introduced different types of hash function algorithms, in which SHA-2 algorithms were modelled using VHDL. The designs were synthesised and implemented using Altera Quartus II and Xilinx. There were two categories of designs: basic and embedded with DSP units and BlockRAMs. When compared to basic designs, the high frequency achieved by implementing SHA-2 using embedded FPGA resources is good. The maximum frequency of SHA-2 increased significantly on Virtex 5 and Stratix III, with 218.2 MHz and 205.8 MHz, respectively. Mestiri et al. (2015) focused on SHA-256 and SHA-512 designs. The designs produced the results based on a finite state machine (FSM) with a padded process unit. The FSM performed five states (i.e., pad 0, pad 1, pad 2, pad 3, and pad F). The SHA-256 design resulted in a maximum frequency of 202.54 MHz and a throughput of 1.58 Gbps.

An improved pipelined architecture of the SHA-256 hash algorithm was designed by Padhi and Chaudhari (2018). The Xilinx Virtex-4 FPGA was programmed to implement the design in Verilog HDL. Achieving better architectural performance was the goal of this design. The maximum frequency of the design was 170.75 MHz with a throughput of 1,344.98 Mbps (Padhi & Chaudhari, 2018). Michail, Milidonis, et al. (2005)proposed SHA-256 design with a maximum frequency of 64.1 MHz and a throughput of 2,052.1 MHz. This design was implemented on Virtex E using the pre-computation technique to improve design throughput Michail et al. (2005) improved the throughput of optimised SHA-256 design using VHDL based on Virtex FPGA implementation. This design used partially unrolled operation with pre-calculation and pre-computation. Furthermore, CSA was also applied in this design to increase its throughput (Michail et al., 2010).

He et al. (2018) proposed the SHA-256 design using three-stage pipeline with Cyclone II FPGA implementation. With the masking scheme, the design achieved a throughput of 655.66 MHz. Wu et al. (2020) and Zhang et al. (2020) put out the idea of designing SHA-256 with ASIC implementation. The 14 nm technique was used to implement and synthesise both concepts. Finally, Li et al. (2019) proposed an asynchronous SHA-256 implementation design in SMIC 40 nm technology. The result of the design was simulated and verified using Synopsys VCS (Li et al., 2019).

Li et al. (2020) proposed FPGA-based implementation and achieved the results of implementation with a maximum frequency of 100 MHz and a throughput of 787 Mbps. This design used neural network and blockchain fusion based on image copyright protection and implemented DNN and SHA-256 on FPGAClick or tap here to enter text.. Brazhnikov (2020) proposed SHA2 with 28 nm CMOS technology. This design produced different

results by applying different adder types to SHA-256 design, such as RCA, CLA, Han-Carlson, Brent-Kung, Kogge-Stone, and Sklansky in terms of delay and area (Brazhnikov, 2020). Maximum frequency and throughput were not covered in this paper, while the main objective was to examine SHA-256's performance with regard to high throughput. Yimeng (2020) proposed a rescheduling method to enhance the throughput. This design was implemented on Virtex-4, which has an area of 979 slices and a throughput of 1,984 MHz.

Bensalem et al. (2021) proposed the latest implementation with improvement in throughput. They improved the SHA-256 design implementation on FPGA using OpenCL optimisation techniques, which comprise the insertion of local memories, as well as loop splitting, unrolling, and pipelining. This design obtained a throughput of 3,973 Mbps (Bensalem et al., 2021). Each of these designs was an earlier implementation of SHA-256 that used ASIC or FPGA implementation. Through the use of unfolding transformation techniques, this work presents an improved design for the SHA-256 hash function and its implementation on FPGA. The goal is to enhance performance. Table 2.10 presents a compilation of previous SHA-256 designs. It compiles prior work on SHA-256 hash function design according to the maximum frequencies, areas, and throughputs for different approaches. This compilation provides a comparative analysis of the methods used in designing SHA-256. It shows that pipelining and unfolding transformations can enhance and improve the performance of RIPEMD-160 design. Besides, this method can also provide small-area implementation and high speed.

Table 2.10 Compilation of SHA-256 Hash Function Design

No.	Design Method	FPGA Device	Authors/ year	FMax (MHz)	Throughput (Mbps)	Area	Power (mW)
1	SHA-256 iterative (binary tree structured adder)	Altera DE-10	(DiNardi & Radhakrishnan, 2023)	82.69	-	3,600 LE	-
2	SHA-256 iterative	Xilinx Vivado	(Goyal et al., 2023)	-	-	-	-
		Artix-7		-	-	-	122
		Kintex-7		-	-	-	84
		Spartan-7		-	-	-	90
3	AES-SHA3 iterative	Spartan 3AN XC3s500e	(Jasim et al., 2023)	50 MHz	64,000	-	-
4	SHA-256 iterative	Xilinx Artix- 7 XC7A200 TL	(Thomas & Bhakthavatchalu, 2021)	302.186	1,190.6	2,282	-
5	SHA-256 iterative	Altera Arria 10 (OpenCL)	(Bensalem et al., 2021)	243	3,970	67,150	-
6	SHA-256 iterative	Xilinx XC7Z020	(Kammoun et al., 2021)	135	1,063	1,305	-
7	SHA-256 iterative (RCA, CLA, Han-Carlson, Brent-Kung, Kogge-Stone, and Sklansky)	Xilinx Zynq7000 XC7Z020 (C-HLS)	(Kammoun et al., 2020)	181	917	6,367	-
8		Cyclone V 5CSXFC6 DF32	(Kieu-Do-Nguyen et al., 2021)	139.04	1,020	1,159	-

Table 2.10 Continued

9	SHA-256 iterative	CMOS technology	(Brazhnikov, 2020)	-	-	-	-
10	SHA-256 pipelining	Xilinx Vivado Kintex Ultrascale	(Martino & Cilardo, 2020)	2.646 ns	-	4,986	0.434
11	SHA-256 rescheduling and iterative	Xilinx Virtex-4	(Chen & Li, 2020a)	255.7	1,984	979	-
12	SHA-256 iterative, resource sharing, and power gating	Xilinx Virtex-7 Xc7v2000 tfhg1761-2	(Gad et al., 2020)	83.33	637	275	13
13	SHA-256 Parallel pipelining	ASIC (Intel 14 nm Technology)	(Wu et al., 2020)	-	-	14,272.6 um^2	6.855
14	Double SHA-256 iterative	Artix 7 XCtA200T	(Pham et al., 2020)	132.5	135.168	-	-
15	SHA-256	Xilinx Vivado Alpha-Data ADM-PCIE-7V3	(Li et al., 2020)	100	787	-	-
16	2019	Zynq-7000 XC7Z030	(Florin & Ionut, 2019)	151.5	79,900	-	-
17	SHA-256 iterative	Synopsys IC compiler SMIC 40 nm technology	(Li et al., 2019)	376.5	3,000	23,490 um^2	-
	SHA-256 cascade structure (unfolding factor 2)			227.27	3,600	38,885 um^2	-

Table 2.10 Continued

18	SHA-256 folded 5	Virtex -4	(Wong et al., 2018)	238	379.6	382	- -
	SHA-256 folding and 4-2 adder	Virtex-4		222	881.1	485	-
19	Multi SHA (SHA-512, SHA-512/224, SHA-512/256)	Xilinx Virtex-5	(Lee & Shin, 2018)	129	826	1,080	-
20	SHA-256 iterative	Xilinx Virtex-VI	(Thakur, 2018)	-	-	1,803 LUTs	-
21	2018	Virtex-4 XC4VLX 100-12	(Padhi & Chaudhari, 2018b)	170.75	1,344.98	610	
22	SHA-256 iterative	Xilinx Virtex-6 xc6vlx550 t-2ff1759	(Rote et al., 2016)	135.14	1,081.12	736	-
	SHA-256 round pipelined			271	2,040.47	905	-
23	SHA-256 iterative	Xilinx Virtex-5 XC5VFX7 0T	(Mestiri et al., 2015)	202.54	1,580	387	-
24	2015	SHA-2 hardware core for Virtex-5 FPGA	(Mohamed & Nadjia, 2015)	170	1,359.6	1,203	-
25	SHA-256 iterative	Xilinx Virtex 5 XC5VLX 330	(Michail et al., 2015)	112	3,880	9,012	-

2.4.3 HMAC Design

2.4.3.1 HMAC-SHA-1

The implementation of HMAC-SHA-1 based on FPGA has been proposed by different researchers with different objectives. Some publications on HMAC-SHA-1 have introduced several objectives, such as small size, high performance, and resource sharing. McLoone & McCanny (2002) proposed a novel single-chip hardware IPSec comprising Rijndael and HMAC-SHA-1. The maximum frequency for this design was 50 MHz. Selimis et al. (2003) observed an increase in maximum frequency, where HMAC-SHA1 was designed for the wireless transport layer security, and the design was implemented on FPGA v150bg352, achieving a maximum frequency of 82 MHz. Wang et al. (2004) introduced HMAC with SHA-1/MD5 and adopted a clock frequency, achieving a maximum frequency of 21.96 MHz. The advantage of this design is that it reduces hardware complexity but still provides similar performance. Besides, an efficient structure of hash function can be developed by sharing the usage of hardware. Michail et al. (2004)showed an implementation on FPGA device v3200efg1156, providing a maximum frequency of 62.0 MHz. Developing a compact and high-speed implementation of HMAC-SHA1 was the primary goal of the work conducted by Yiakoumis et al. (2005). A maximum frequency of 111 MHz was obtained using Xilinx Virtes E -8. In order to enhance the design's throughput, Khan et al. (2007) implemented parallelism and pipelining in the HMAC-hash unit. The work integrated the hash function into one reconfigurable unit, with a maximum frequency of 43.47 MHz on Xilinx XC2V4000. The unit consisted of MD5, SHA-1, RIPEMD-160, HMAC-MD5, HMAC-SHA-1, and HMAC-RIPEMD-160.

2.4.3.2 HMAC-SHA-2

Security has recently emerged as a hot topic among researchers. To enhance the effectiveness of these information-protecting processes, various cryptography algorithms have been developed. Message digests are generated using hash function techniques during data transmission. Therefore, it becomes an essential instrument for the integration of security into applications such as internet banking, email, and other software. A hash function produces a fixed-length output from an arbitrary-length message input. The one-way nature of hash functions makes it challenging to convert a hash value to a message input. Hash functions, such MD5, RIPEMD160, or SHA-1, are a type of keyless cryptography. In this application, HMAC was used to build and modify an SHA family to fit the needs of cryptographic algorithms in terms of performance. The SHA-2 hash algorithm supports four distinct hash types: 224, 256, 384, and 512. These hash algorithms use SHA-length, which can be anywhere from 256 bits to 512 bits, to calculate the length of their output. The application of HMAC focuses on SHA-256 hash function.

This research optimises hardware resources and performance utilising hash function of SHA-256 with MAC. Meanwhile, IPSec and HMAC-SHA-256 are the focus for several projects related to the optimisation of hardware size, performance, and hardware consumption. McLoone & McCanny (2002) proposed IPsec hardware on a single chip comprising Rijndael and HMAC-SHA-256. A wireless design increases the maximum frequency, as shown in a previous study (Selimis et al., 2003). On the other hand, the HMAC with SHA-1/MD5 was initially presented in earlier research (Wang et al., 2004), where hardware complexity was minimised, and an efficient hash function structure was devised to share hardware. Additionally, Michail et al. (2004) demonstrated the HMAC-SHA-1 implementation on an FPGA device, whereas Yiakoumis et al. (2005) demonstrated the

execution of a small-sized, high-speed HMAC-SHA-1. The HMAC-hash unit was built and combined using pipelining and parallelism as part of the improvement approaches adopted in Khan et al. (2007a). Even though these ideas worked well, the greatest frequencies that they could reach were only a few tens of MHz, or up to 111 MHz, as obtained in Yiakoumis et al. (2005). In the meantime, prior research by Juliato & Gebotys (2011) and Rubayya & Resmi (2015) designed an FPGA implementation of HMAC based on SHA-256. The results were obtained using a Xilinx device in both designs. Furthermore, the HMAC design in Rubayya & Resmi (2015) exhibited significant improvement. This suggests that in order to meet the demands of current systems, greater performance is needed.

Table 2.11 summarises previous HMAC designs employed with several hash functions, including RIPEMD-160, MD5, and SHA-1. Numerous studies have been conducted on HMAC, but not all of them have focused on the frequency maximum of FPGA design (Choi et al., 2020), (Oku et al., 2018), (Lin et al., 2017), (Ravilla & Putta, 2015b). Kieu-Do-Nguyen et al. (2022) combined all SHA-2 families into one core, including HMAC-SHA2-224/256/384/512, whereas Pham et al. (2022) designed a combination of SHA-256/512/256d hash function. This research focuses on SHA-256 due to its wide implementation in security design, such as Bitcoin, which is also known as cryptocurrency.

Table 2.11 Previous HMAC Design

No.	Authors/ Year	FPGA Device	HMAC Design	FMax (MHz)	Throughput (Mbps)	Area
1	(McLoone & McCanny, 2002)	Xilinx XCV1000E	HMAC-SHA-1	24.2	30–37.8	-
2	(Selimis et al., 2003)	Xilinx Virtex XC1000E	HMAC-SHA-1	82	-	1,018 CLBs

Table 2.11 Continued

3	(Wang et al., 2004)	Altera EP20K100EBC652-1X	HMAC-SHA-1	21.96	27.7–34.7	5,329
			HMAC-MD5	21.96	34.6–43.2	
4	(Michail et al., 2004)	Xilinx	HMAC-SHA-1	62	1,587	6,011
5	(Yiakoumis et al., 2005)	Xilinx Virtex (-6)	HMAC-SHA-1	91	582.4	686
			HMAC-MD5	55	440.0	612
		Xilinx Spartan-3 (-4)	HMAC-SHA-1	87	543	750
			HMAC-MD5	62	480	665
6	(Yiakoumis et al., 2006)	Xilinx Virtex-II (-6)	HMAC-SHA-1	162	1,024.0	854
			HMAC-MD5	96	756.0	797
		Xilinx Virtex-E (-8)	HMAC-SHA-1	111	701.6	686
			HMAC-MD5	65	512.0	612
7	(Khan et al., 2005)	Xilinx Virtex II	HMAC-MD5	43.21	171.2	7,911 LUTs
			HMAC-SHA-1		137.4	7,484 LUTs
			HMAC-RIPEMD-160		137.4	9,650 LUTs
8	(Kakarountas et al., 2006)	Xilinx Virtex XCV150BG352	HMAC-SHA-1	98.7	2,526.7	950
		Xilinx Virtex-E		111.5	2,854.40	950
		Xilinx Virtex-II		163.3	4,180.48	854
9	(Juliato & Gebotys, 2011)	Altera Apex 20K, EP20K1000EBC652	HMAC (SHA-256)	35.55		

Table 2.11 Continued

10	(Juliato & Gebotys, 2011)	Xilinx Virtex-E, XCV1600 EBG1156	HMAC (SHA-256)	48.12		
11	(Juliato & Gebotys, 2011)	Xilinx Virtex-II, XCV2V4 000BF95 7	HMAC (SHA-256)	59.66		
12	(Chen & Yuan, 2012)	No FPGA implemen tation	HMAC (SHA-256)	-		
13	(Abbas et al., 2014)	RIVYERA S6-LX150 Spartan6-LX150	HMAC-multi RIPEMD-160 pipeline	-	-	-
14	(Rubayya & Resmi, 2015)	Xilinx Device (no device name)	HMAC (SHA-256)	110.0 09		
15	(Ravilla & Putta, 2015b, 2015a)	No FPGA implementat ion	HMAC (SHA-256)	-		
16	(Lin et al., 2017)	No FPGA implementat ion	HMAC (SHA-256)	-		
17	(He et al., 2018)	Altera Cyclone II	HMAC-SHA-256 pipeline (3p)	87.08 MHz	655.66	6,131
18	(Oku et al., 2018)	No FPGA implementat ion	HMAC (SHA-256)	-		
19	(Choi & Seo, 2020)	No FPGA implementat ion	HMAC (SHA-256)	-		

Table 2.11 Continued

20	(Kieu-Do-Nguyen et al., 2022)	Virtex 4/Virtex 5	HMAC-SHA-256	188		
21	(Pham et al., 2022)	Virtex2 XC2VP20	SHA-256	165		

2.5 Summary

Contemporary cryptographic hash functions encounter various challenges and constraints, largely due to their fundamental architectural designs and persistent security weaknesses. The challenges are substantial as they directly affect the integrity and reliability of data security across various applications, including Internet of Things (IoT) ecosystems and blockchain platforms. Several significant research gaps and flaws in the body of current work have been highlighted by the critical review included in this chapter:

i. Low Operating Frequency Maximum

Many existing hash function designs fail to achieve high clock frequencies, which inherently restricts their suitability for time-critical applications and limits overall system responsiveness. In particular, iterative architectures typically employed in hash function realizations contribute to this constraint by introducing sequential dependencies.

ii. Low Throughput

A significant number of previous designs demonstrate low data processing rates resulting from non-optimized sequential implementations and limited use of pipelining or unfolding techniques. While several studies have looked into unfolding, they usually use low unfolding factors, which results in only slight throughput gains. The integration of pipelining

and unfolding is inadequately explored, with limited comprehensive studies examining their combined potential to improve performance.

iii. Insufficient Optimisation of HMAC for FPGA in IP Security Applications

Although numerous studies have introduced generic hash function accelerators, there is a significant lack of research focused on the efficient integration of hash functions in HMAC constructions specifically optimised for FPGA deployment, particularly within Internet Protocol (IP) security frameworks.

iv. Insufficient techniques for power optimisation.

Many current studies predominantly emphasise throughput improvement or area reduction, frequently overlooking sophisticated approaches for dynamic power reduction. There is a lack of exploration regarding techniques such as Gray encoding, clock gating, and unfolding transformations that are specifically designed for low-power cryptographic implementations. Recent research trends have focused on innovative approaches, including parallel processing architectures, pipelined designs, and lightweight hash function constructions, which show significant potential for enhancing security robustness and performance efficiency in next-generation implementations. These methodologies have not yet undergone comprehensive validation or standardisation for FPGA-based cryptographic hardware aimed at high-assurance applications. While FPGA-based implementations are acknowledged for their potential to overcome conventional constraints and deliver high-throughput, energy-efficient hashing capabilities, however, the systematic application of advanced optimisation strategies across frequency, throughput, and power domains is still underdeveloped.

CHAPTER 3

METHODOLOGY

3.1 Digital Design Flow

This study utilises a systematic digital design methodology to construct, synthesise, and evaluate cryptographic hash function and HMAC implementations on FPGA hardware. Hardware Description Languages (HDLs), especially Verilog HDL, are employed to describe the functional behaviour, Register-Transfer-Level (RTL) architecture, and timing attributes of each design. Verilog HDL was chosen for its extensive industry acceptance, robust support in FPGA design tools, and alignment with IEEE standards for electrical hardware description.

By adhering to the standards for the development of high-quality and efficient HDL coding styles, it is possible to achieve better results in the aspects of speed, power, and area for hash functions and HMAC design. Therefore, writing HDL codes with proper guidelines for any application can contribute to an efficient method. One of the advantages of HDL is technology independence. It is possible to build the design without choosing a specific fabrication technique. Nowadays, the complexity of digital strategy is an essential issue in designing a large structure. Therefore, HDL is an appropriate language for resolving this issue. A typical design sequence for a very-large scale integration integrated circuit is illustrated in Figure 3.1. Digital design circuits are described by the utilisation of HDL at the RTL. The levels of representation and processes in the design flow are represented by the unshaded and shaded rectangles, respectively. The digital design process starts with the specification of the algorithmic functionality and target architecture for the hash functions and HMAC structures. The designs are initially expressed in Verilog HDL at the RTL level,

defining the data pathways, control logic, and modular structure. Functional verification is conducted with simulation tools to confirm that the HDL description yields accurate outputs in response to representative input vectors.

Upon verification, synthesis tools in Altera Quartus II convert the RTL code into a gate-level netlist optimised for the designated FPGA device. This technique translates the abstract HDL representation into physical resources, including logic parts, registers, and routing connections. The synthesis phase is directed by limitations on timing, area utilisation, and power consumption to meet the performance objectives of the project. Once synthesis is complete, place-and-route operations are carried out to optimise interconnect pathways and allocate distinct FPGA locations to each logic component. After that, timing analysis and static verification are carried out to make sure the design satisfies setup/hold and critical path criteria. ModelSim simulations are employed to verify the behaviour of the post-synthesis netlist and to produce waveform data for further analysis.

The entire design flow is characterised by iterative refinement, which involves the modification of the HDL code and the repetition of the verification stages to resolve functional or timing issues that are identified in simulation or synthesis. This methodology ensures that the final implementation meets the design parameters for speed, area, and power efficiency. This research use FPGA devices as the target platform, allowing for the prototyping and evaluation of various cryptographic algorithms under uniform hardware limitations. The integration of Verilog HDL, Quartus II synthesis, and ModelSim simulation facilitates a reliable and repeatable design methodology appropriate for the implementation of high-performance, resource-efficient cryptographic primitives.

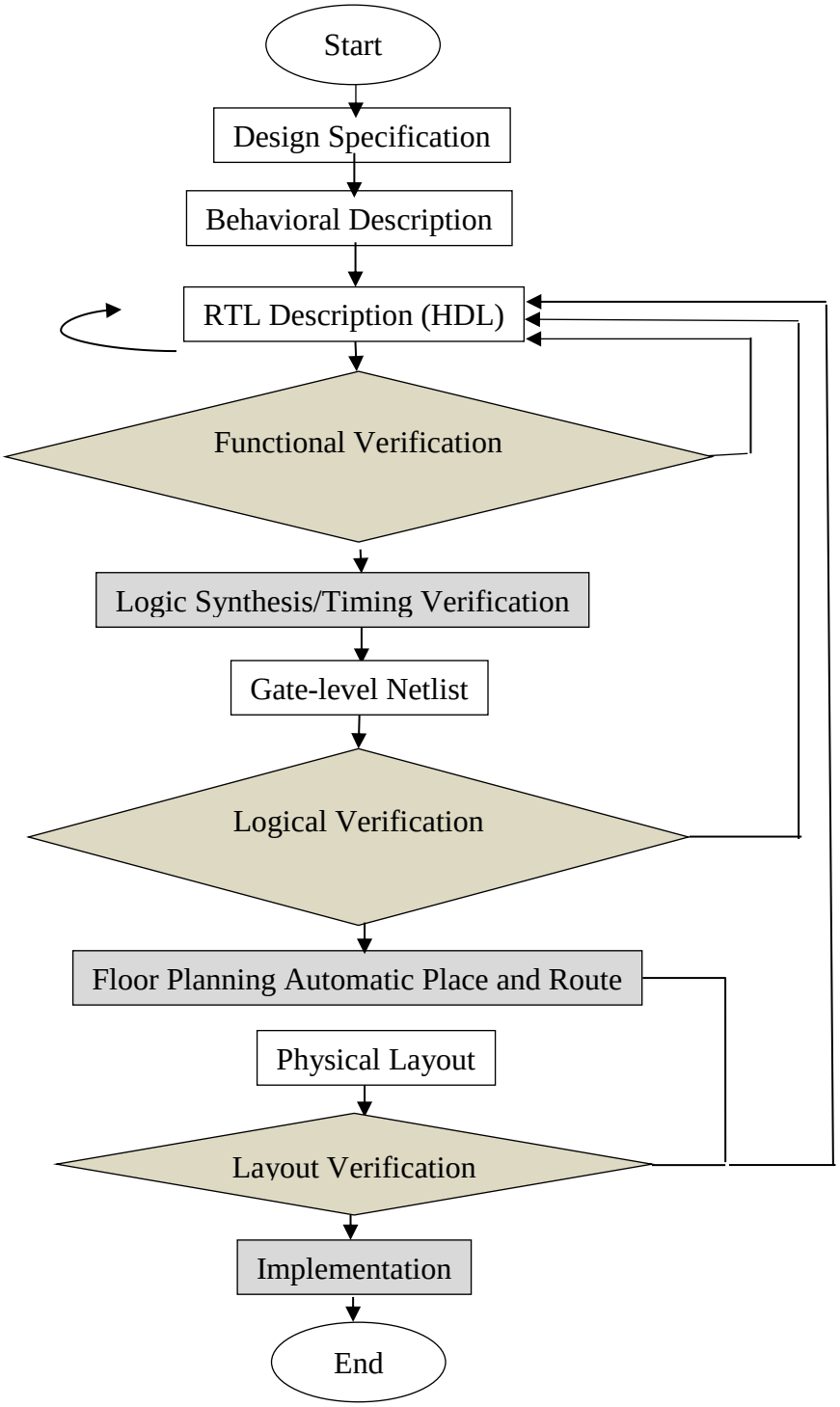


Figure 3.1 Typical Design Flow

Figure 3.2 illustrates an FPGA implementation design flow of a digital design. First, the designer needs to write the digital logic in Verilog. This design entry phase declares all the connection wires and buses for input and output. Furthermore, this top-level architecture includes Verilog code instantiation modules. The output design is validated by simulating the consequences of the design using a testbench file. The efficiency of the concept is now verified using functional simulation. The RTL design, Verilog code, must be converted into gate level based on an FPGA family device chosen in the design's early phase. The design will undergo compilation and synthesis processes to translate the Verilog code into a netlist representing the physical hardware device.

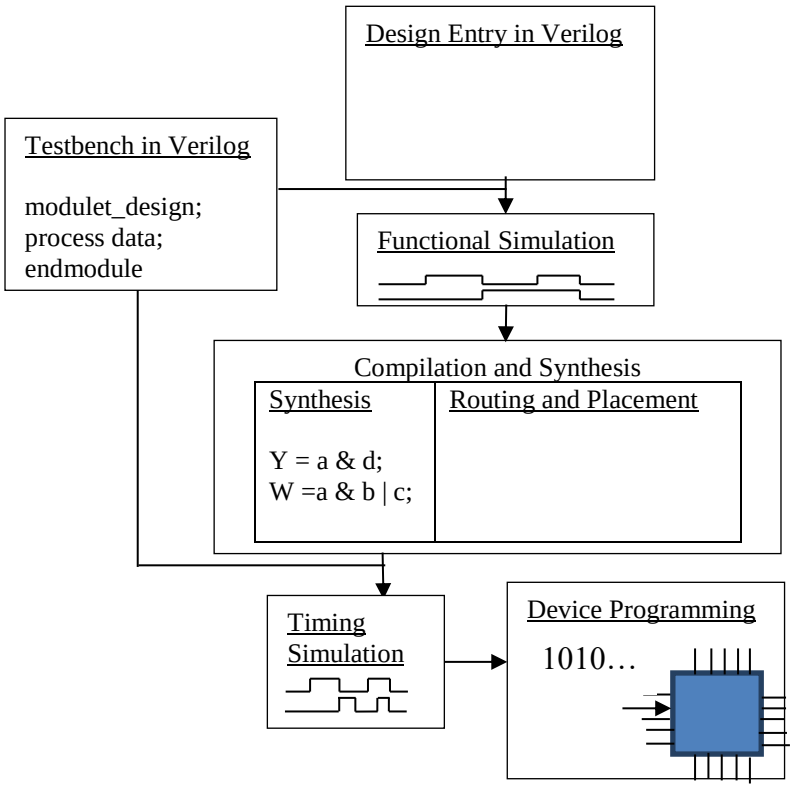


Figure 3.2 FPGA Design Flow

Logic synthesis tools play essential roles in digital electronic design automation. A timed simulation is done to assess the outcomes in terms of time setup and time hold once the synthesis process ends. Then, the design can be transferred to the FPGA device.

3.2 Design Overview of Hash Function and HMAC Algorithm

Several hash function algorithms such as MD5, SHA-1, RIPEMED-160, and SHA-256 are constructed in this work using Verilog code. The core structure of the algorithm in terms of technique is modified, which leads to increased hash function performance. In this study, HMAC is an application that employs the hash function. Figure 3.3 depicts an overview of the hash function and the HMAC algorithm. This chapter discusses the algorithm of the fundamental design loop used in building digital hardware design in the following section.

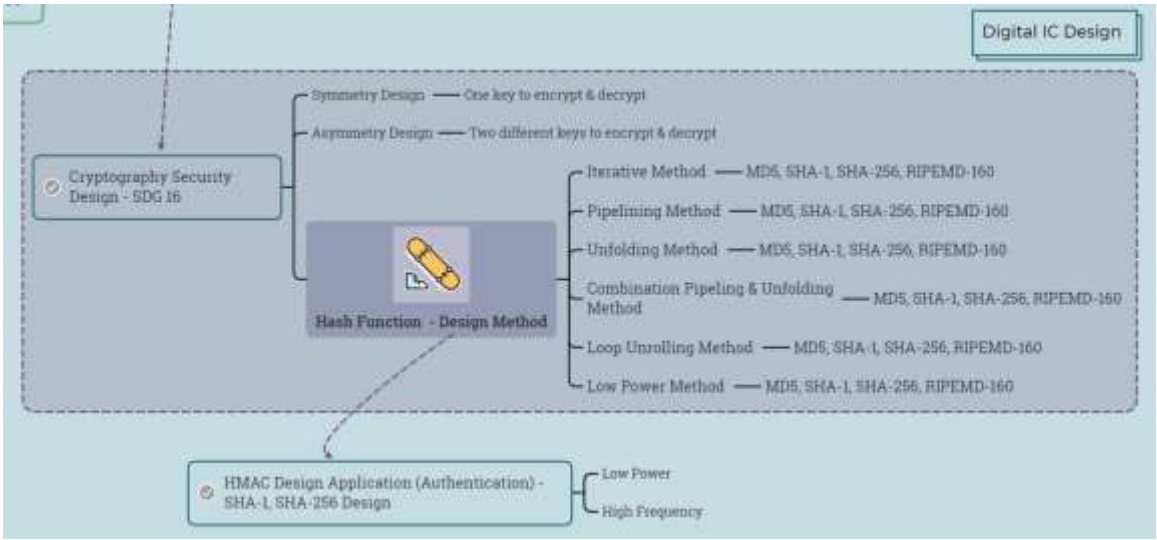


Figure 3.3 Design Overview of Hash Function and HMAC Algorithm

The research methodology of the design process is depicted in Figure 3.4. It shows the concept and framework of the project to ensure that the study utilises rigorous methodology and strong theoretical foundations.

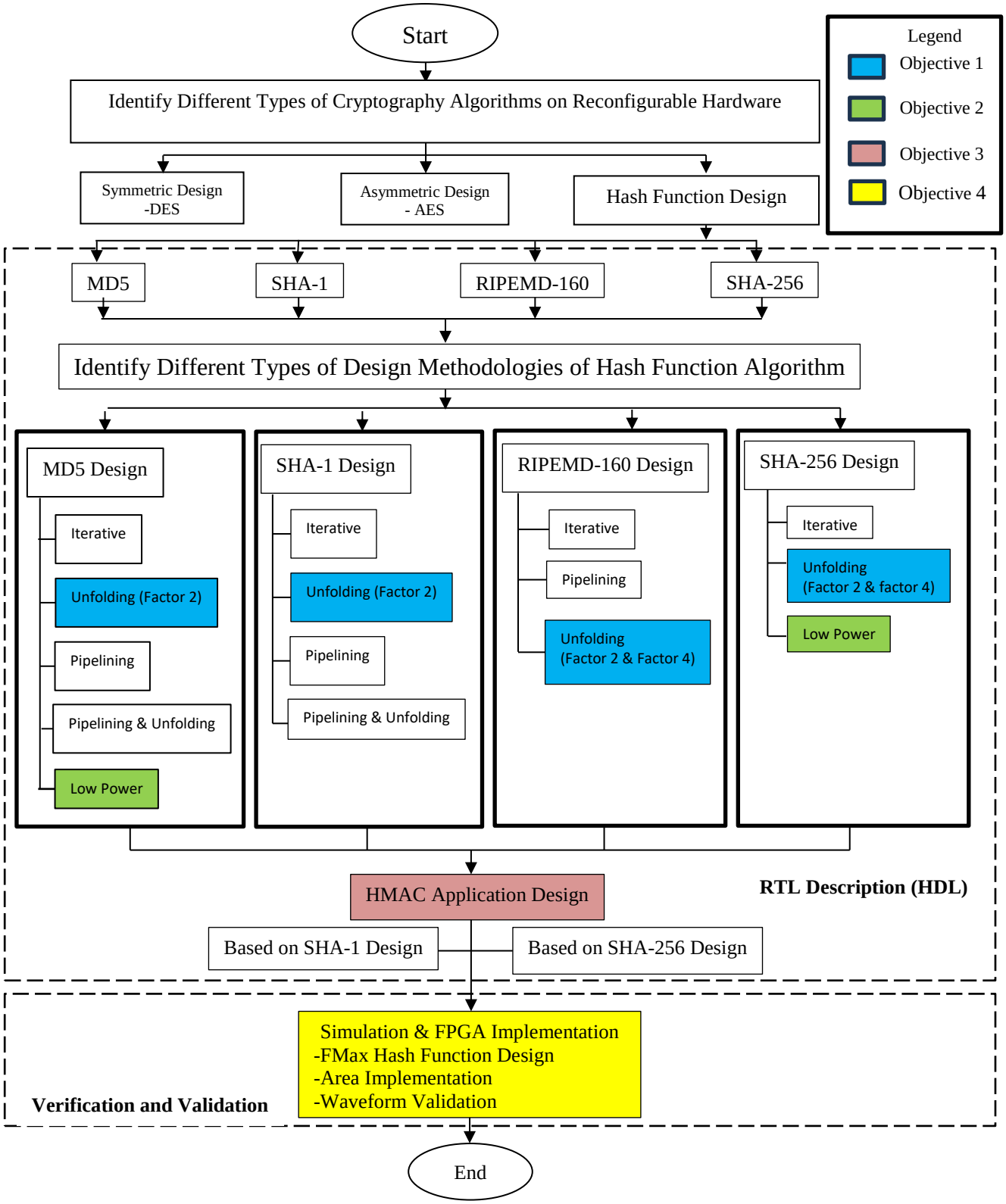


Figure 3.4 Design Overview of Hash Function and HMAC Algorithm

This research selected MD5, SHA-1, RIPEMD-160, and SHA-256 due to their extensive utilisation in security protocols and their differing degrees of cryptographic robustness and architectural intricacy. MD5 and SHA-1 are outdated hash algorithms that, although not advisable for high-assurance applications, continue to hold significance in legacy systems and serve as useful benchmarks for assessing design techniques. Conversely, RIPEMD-160 and SHA-256 are superior algorithms that provide enhanced security margins, crucial for contemporary applications demanding greater resilience against collision and preimage attacks. With respect to the selection of unfolding factors, MD5 and SHA-1 were implemented with an unfolding factor of 2 primarily because of their relatively straightforward internal architectures, which include fewer cycles and smaller state sizes. In these designs, the application of an unfolding factor of 2 resulted in a measurable increase in throughput without significantly increasing area utilisation or complicating routing on the FPGA fabric.

The performance metrics of RIPEMD-160 and SHA-256 were examined by exploring unfolding factors of both 2 and 4 to assess the effects of increased parallelisation. These algorithms have more computationally complex structures and a higher number of processing rounds, making them ideal for deeper unfolding. Unfolding factor 4 was utilised to evaluate the potential for increased throughput through higher degrees of parallelism, while also considering the trade-offs in logic resource utilisation and power consumption. This approach allowed the research to systematically compare the performance, area, and power characteristics of multiple unfolding configurations across hash functions with different levels of security strength and architectural complexity. The results provide insights into the scalability and feasibility of unfolding strategies when implementing cryptographic hash functions on FPGA platforms for secure embedded applications.

The research flow design starts with a design concept, as shown in Figure 3.5. The subsequent step involves initiating the project's initial design. This step is the most crucial part of the process because it requires much of the designer's effort, knowledge, skill, and intuition. A flowchart of the HMAC design and the overall hash functions is illustrated in Figure 3.5.

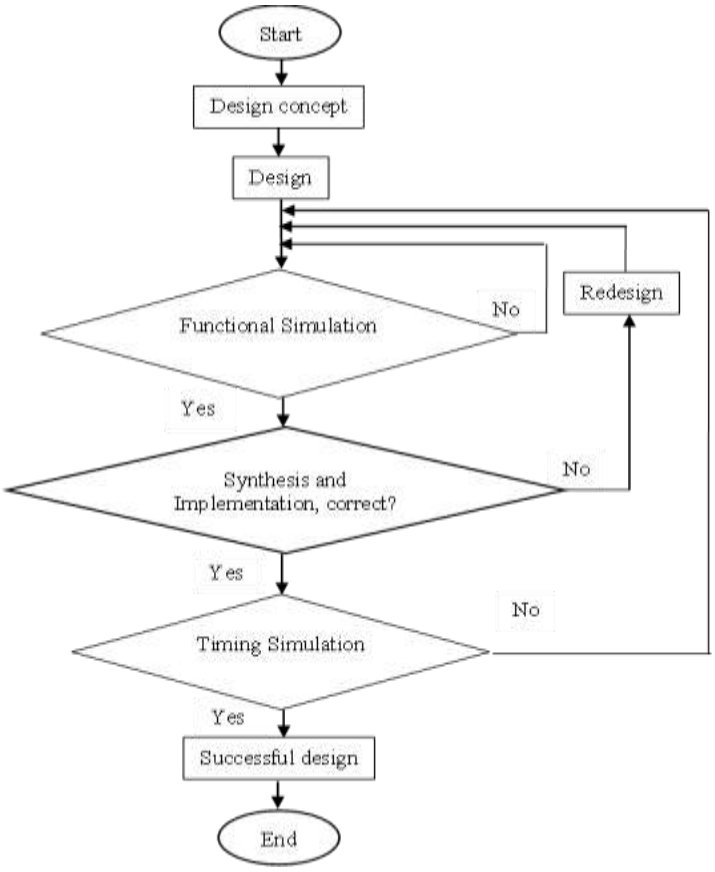


Figure 3.5 Flowchart for the Overall Hash Function and HMAC Design

Developing digital techniques, such as the hash function and HMAC, following the appropriate guidelines yields an efficient digital design. In this phase, two types of modelling are used in creating the hash functions and HMAC: behavioural modelling and dataflow modelling. In behavioural modelling, procedural statements such as if-else and case statements are employed, whereas continuous assignment is employed in dataflow modelling. This is achieved through the use of hash functions and HMAC. Testbench is a

file used during the simulation exercise to ensure the hash functions and HMAC design work correctly. Both the functional and timing aspects of these designs require simulation. Input data are provided as a test pattern to observe the output of results. It is necessary to incorporate the top-level design module with input data within the testbench file to produce output results. The functional simulation can do design validation. The gates are integrated at this point, and there is no wire delay. ModelSim is the design model tool for this research. Functional simulation is done at the clock level and does not consider glitches, setup and hold violations, and other timing issues (Navabi, 2006). The simulation requires data, top-level design, and a testbench file to produce a waveform of the architecture design. For functional simulation, results in terms of delay cannot be detected at this level.

Synthesis and implementation processes are executed to check the design for syntax errors after designing the circuit using Verilog codes based on Altera Quartus II CAD tools. As stated previously, this phase translates the RTL codes into the logic gate netlist. The target hardware device during compilation must be the FPGA device. The compilation procedure employs the timing and functional specification information. The design is converted to the intermediate phase (analysis phase), followed by combining all the parts, generating the corresponding logic (synthesis phase), placing and routing components of the FPGA device, and producing timing details (Navabi, 2006).

When the design is completed, timing simulation will take place. The same simulator tool, ModelSim, generates timing simulation in this study. The input data will be given to the testbench file to evaluate the design results. In this phase, wire and gate delays are included during the simulation. The results will be different from those of the functional simulation. Timing analysis can be evaluated; the worst-case delay of the design is generated

from Altera Quartus II tools after the synthesis and implementation. The design must be changed to correct the output if there are any errors. This loop is repeated until the operation of the invention is successful. Finally, ModelSim is used again to evaluate and verify the design's functional and timing simulation output.

3.3 Dataflow and Behavioural Modelling

In this research, two types of modelling are used to design the hash function and HMAC: dataflow modelling and behavioural modelling (Palnitkar, 2003). Structural or gate-level modelling is not included in these designs because they consist of large circuits. Structural modelling is suitable for small circuits where the number of gates is small.

3.3.1 Dataflow Modelling

Dataflow modelling provides powerful methods to implement the digital circuit. This type of method can optimise circuit performance. The hash functions and HMAC in dataflow modelling are designed using a continuous assignment. Figure 3.6 shows a simple example of dataflow modelling. It consists of an operand and an operator and describes the design in terms of expressions.

```
//continuous assignment
wire out;
assign out = in1 & in2;
```

Figure 3.6 Dataflow Modelling

3.3.2 Behavioural Modelling

Behavioural modelling is like C programming. It describes the behaviour of the circuit. The designer needs to think about how the circuit works and performs to implement the circuit. Besides, it describes the functionality of the circuit. These designs use two types

of procedural statements: *if-else* and *case* statements. The design elements comprise "always blocks" in conjunction with a sensitivity list for the combinational circuit, while the sequential circuit incorporates a clock. These circuits are designed using two categories of assignments: non-blocking assignments and blocking assignments. Blocking assignment (=) is used for combinational circuits, while non-blocking assignment (<=) is used for sequential circuits. Table 3.1 shows a simple example of behavioural modelling for sequential and combinational circuits.

Table 3.1 Behavioural Modelling

Verilog HDL Codes	**Types of Circuit**	
	Combinational Circuit	**Sequential Circuit**

Example module	module andor(a,b,c,y,z);
	
	input a,b,c;
	output y, z;
	
	//procedural statement
	always@(a or b or c)
	begin
	y = a & b;
	z = c | y;
	end
	
	endmodule
		module count1(clk,reset,c);
		
		input clk, reset;
		output [2:0] c;
		reg [2:0] c;
		
		//procedural statement
		always@(posedge clk or negedge reset)
		begin
		if(!reset)
		c <= 0;
		else
		c <= c + 1;
		end
		
		endmodule

3.4 Hash Function and HMAC Algorithm

This section describes hash functions and HMAC techniques in detail. The MD5, SHA-1, RIPEMD-160, and SHA-256 algorithms were investigated as hash function types in

this research. The hash function was then employed to elucidate the structure of the HMAC technique, emphasising the significance of the message authentication code in the advancement of cryptographic security.

3.4.1 MD5 Algorithm

Cryptographic protocols and internet communication are frequently implemented with hash function algorithms. Ronald Rivest developed the MD5 message-digest algorithm, which is one of the most renowned hash functions. (Henriquez et al., 2006). MD5 algorithm starts with pre-processing the message by calculating the bit length of arbitrary input It is a process that converts a variable-sized input *m* into a fixed-size string known as a hash value. The modern hash function aims to enhance the internal compression mechanism and the sequence of message processing (Rivest, 1992). The core of a hash algorithm is the compression function.

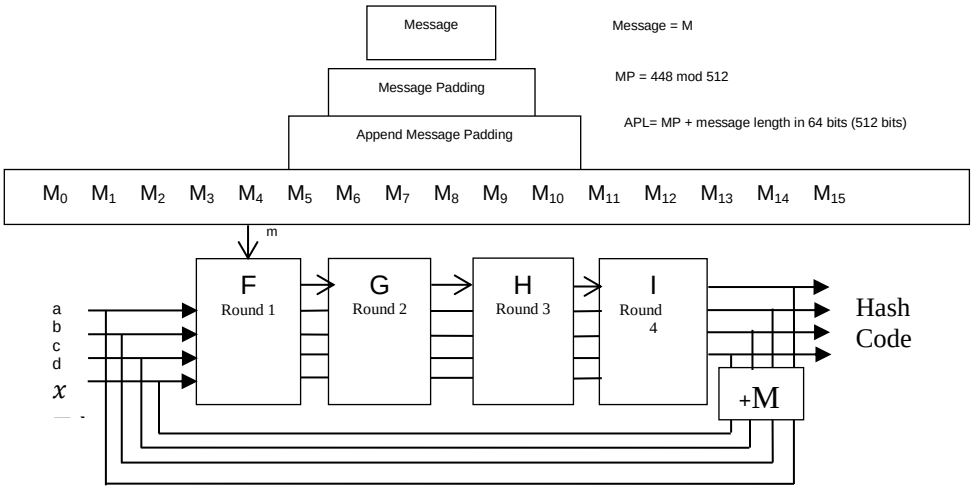


Figure 3.7 MD5 Architecture

Figure 3.7 depicts the mechanism of the MD5 compression function. The nth-bit message undergoes padding by adding a single 1 bit at the end, followed by 0 bit until the total length of the message is congruent to 448 mod 512. Following the padding bit of the message input, the subsequent 64 bits are allocated for appending the message length,

consisting entirely of zeroes except for the last bytes, which represent the length of the message counter. Consequently, the total message input, M, is 512 bits. The MD5 message algorithm starts processing with the four-word constant initial value input of 32-bit register (A, B, C, D) buffer initialisation, as shown in Table 3.2. The process will start executing in four rounds, each with 16 steps. The iterative process for each round will continue until 64 rounds to obtain the hash output of the MD5 algorithm.

Table 3.2 Buffer Initialisations

	Normal Value	**Little-Endian Format**
A	32'h01234567	32'h67452301
B	32'h89abcdef	32'hefcdab89
C	32'hfedcba98	32'h98badcfe
D	32'h76543210	32'h10325476

The security of the compression function is a significant factor in the cryptography of the hash function (Henriquez et al., 2006). The term compression is derived from the bit length of the original input message, which is used to reduce the bit length of the output hash code. Four non-linear functions are present in this design: F, G, H, and I. A compression function with four distinct inputs, including A, B, C, and D, is depicted in Figure 3.8. This function consists of a non-linear function, message input, constant value, and shift value. For every non-linear function, this function takes three parameters namely B, C, and D and returns the correct output. Therefore, there are 64 rounds of operation to produce the result of the MD5 hash code. Behavioural/RTL modelling is used as a coding style due to the procedural statement in the design. Design with a less sensitive list can reduce the compilation and simulation time. However, if some of the signals in the design are not listed in the sensitivity list, the design will give the wrong results. In this design, structural and dataflow modelling are not used in designing the compression function because the design

data keep changing every 10 ns. $M_i[k]$ represents the message input of the design with *k* times and i rounds. The 32-bit message input is executed every 10 ns and *k* times the message count. To ensure the efficient operation of the data, they are stored in a specific memory location.

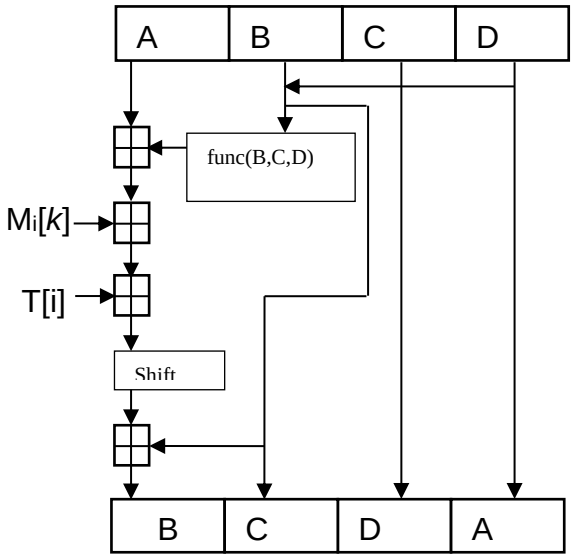


Figure 3.8 Compression Function of MD5

Consequently, the overall algorithm is executed in 64 stages. The auxiliary functions of the four cycles of the MD5 algorithm are illustrated in Equations 3.1, 3.2, 3.3, and 3.4. The logical operations AND, OR, NOT, and XOR are represented by ∧, ∨, ¬, and ⊕, respectively. The little-endian format is used to execute all operations due to its ability to facilitate rapid processing on multiple processors. From these equations, the non-linear function H only uses the XOR gate for 16 operation steps. This operation entails high-power consumption; it is well-established that the XOR gate produces a lot of glitches in the design during state transition.

$$F(B,C,D) = (B \wedge \mathrm{C}) \vee (\neg B \wedge \mathrm{D}) \qquad \text{Equation 3.1}$$

$$G(B,C,D) = (B \wedge \mathrm{D}) \vee (\mathrm{C} \wedge \neg D) \qquad \text{Equation 3.2}$$

$$H(B,C,D) = (B \oplus C \oplus D) \quad \text{Equation 3.3}$$

$$I(B,C,D) = (C \oplus (B \vee \neg D)) \quad \text{Equation 3.4}$$

Considering input i as a step-index of the MD5 algorithm for each step, the 64-element of the 32-bit constant table T[i] constructed from the sine function is used in this design, as shown in Table 3.3. Subsequently, let the symbol << S denotes a left circular shift by S bits, and $M_i[k]$ represents the kth 32-bit word of M_i. Thus, the value $Temp$ from Equation 3.5 can be found by using the values for k and S at different indexes, as shown in Table 3.4. The operations of the MD5 function are executed from $i = 0$ until $i = 63$. Equation 3.5 shows the operation for MD5 rounds with *Func*(*B*,*C*,*D*) representing function F(*B*,*C*,*D*) for the first round, G(*B*,*C*,*D*) for the second round, H(*B*,*C*,*D*) for the third round, and I(*B*,*C*,*D*) for the fourth round. The hash function's output is obtained in the final step of the MD5 algorithm. The final output of the compression function I(*B*,*C*,*D*) will be added to the initial value (IV) to generate the hash value. The normal value must be converted back to the output hash value.

$$(A, B, C, D, M_i[k], S, T[i])$$

$$Temp = B + ((A + Func(B,C,D) + M_i[k] + T[i]"S)$$

$$A \leftarrow D, B \leftarrow Temp, C \leftarrow B, D \leftarrow C \quad \text{Equation 3.5}$$

Table 3.3 Constant T[i] of MD5 Algorithm

i	T[i]	i	T[i]	i	T[i]	i	T[i]
0	d76aa478	1	e8c7b756	2	242070db	3	c1bdceee
4	f57c0faf	5	4787c62a	6	a8304613	7	fd469501
8	698098d8	9	8b44f7af	10	ffff5bb1	11	895cd7be
12	6b901122	13	fd987193	14	a679438e	15	49b40821
16	f61e2562	17	c040b340	18	265e5a51	19	e9b6c7aa
20	d62f105d	21	02441453	22	d8a1e681	23	e7d3fbc8
24	21e1cde6	25	c33707d6	26	f4d50d87	27	455a14ed

Table 3.3 Continued

28	a9e3e905	29	fcefa3f8	30	676f02d9	31	8d2a4c8a
32	fffa3942	33	8771f681	34	6d9d6122	35	fde5380c
36	a4beea44	37	4bdecfa9	38	f6bb4b60	39	bebfbc70
40	289b7ec6	41	eaa127fa	42	d4ef3085	43	04881d05
44	d9d4d039	45	e6db99e5	46	1fa27cf8	47	c4ac5665
48	f4292244	49	432aff97	50	ab9423a7	51	fc93a039
52	655b59c3	53	8f0ccc92	54	ffeff47d	55	85845dd1
56	6fa87e4f	57	fe2ce6e0	58	a3014314	59	4e0811a1
60	f7537e82	61	bd3af235	62	2ad7d2bb	63	eb86d391

Table 3.4 Shift, S, and message value, *k* of MD5 Algorithm

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

3.4.2 SHA-1 Algorithm

The hash function is a transformation that converts the variables in the input message M into a fixed-length value known as the hash value (Henriquez et al., 2006). The design of SHA-1 hash function using Verilog HDL based on FPGA is studied to optimise both the hardware resource and performance. The objective of this study is to examine the structure of the SHA-1 hash function on reconfigurable hardware and to achieve a high maximal

frequency and a small area implementation. In short, a balance between maximum frequency and area implementation of the design needs to be considered. The high performance of hash function design is important to improve the throughput of the design as nowadays, all systems need fast implementations. The motivation of this research is to investigate the structure of the SHA-1 hash function, which is crucial for certain applications, including the message authentication code (MAC) (Henriguez et al., 2006). The SHA-1 hash algorithm has been studied with careful design at every stage of its inner structure using Verilog. Many researchers have considered using SHA-1 FPGA-based implementation (Díez et al., 2002; Jarvinen, 2004; Kang et al., 2002; Miao et al., 2009; Dai & Zhou, 2003; Jiang et al., 2009; Sklavos et al., 2003; Lee et al., 2006; Kim et al., 2012; Lee et al., 2011). However, some of the studies require further improvement. This research chooses the Altera Quartus II Arria II GX: EP2AGX45DF29C4 as the target device for implementing SHA-1, SHA-1 unfolding, and combined SHA-1 unfolding with pipelining because to its capability to deliver superior performance for the design. The SHA-1 input must $< 2^{64}$ and the message is processed in sequential 512-bit blocks, yielding a 160-bit message digest output. The SHA-1 algorithm's processing is divided into two segments: pre-processing and hash computation. The non-linear function of SHA-1 processes three 32-bit words, namely B, C, and D, following a logical sequence from f_0 until f_{79}.

The pre-processing procedure comprises three steps: padding the message, parsing the padded message into message blocks, and setting the initial hash value (Nichols, 1999). The message padding is a multiple of either 512 or 1,024 bits, depending on the input message. Let the width of an input message M be l bits. Therefore, the value of the input l must be in the range of $0 \leq l < 2^{64}$. The length of the message, which is equivalent to 448 modulo 512, is then followed by a 0-bit and a single 1-bit at the end. M(1), M(2).... M(N) is

the result of parsing the padded message M into N 512-bit blocks. The buffer initialisation of SHA-1 is presented in Table 3.5. As seen in Table 3.5, it consists of five 32-bit starting values that must be executed prior to the hash computation.

Table 3.5 Buffer Initialisation of SHA-1

Register	Buffer Initialisation
H_0	32'h67452301
H_1	32'hefcdab89
H_2	32'h98badcfe
H_3	32'h10325476
H_4	32'hc3d2e1f0

The padded message is processed using a 32-bit message schedule consisting of 80 stages (W_0, W_1.....W_{79}) in the SHA-1 hash computation. The SHA-1 compression function for inputs A, B, C, D, and E is depicted in Figure 3.9 and is represented by Equation 3.6. The symbol << means that the register input shifts to the left with the value given. T is made up of W_t and K_t, where W_t and K_t are the expanded message word of round t and the round constant of round t, respectively.

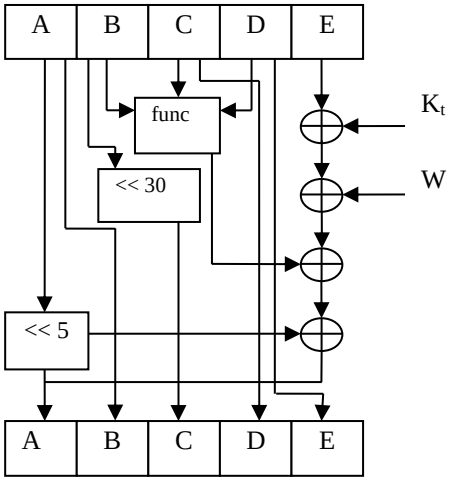


Figure 3.9 SHA-1 Compression Function

$$T = (A \text{ "5}) + F(B, C, D) + W_t + K_t + E$$

$$A = T, B = A, C = B \ll 30, D = C, E = D \qquad \text{Equation 3.6}$$

The formula for the derivation of 32-bit block message schedule W_t is simply from the message input for $t < 16$. The remaining values of W_t, where $t \geq 16$, are derived using the following Equation 3.7.

$$W_t = ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \qquad \text{Equation 3.7}$$

Following the initialisation of five operational variables A, B, C, D, and E with buffer values H_0, H_1, H_2, H_3, and H_4 during pre-processing, the hash computation employs the constant K_t and the round function for $0 < t \leq 79$, as delineated in Tables 3.6 and 3.7, to process the message. The symbols $\wedge$, $\neg$, and $\oplus$ in the non-linear function of the four-round SHA-1 algorithm, as presented in Table 3.6, denote the logical operations AND, NOT, and XOR, respectively. After rounding four rounds that consist of 80 steps, the final step is adding the initial value to the last output hash.

Table 3.6 Constant K_t

Step Number	***K*$_t$(Hex)**
$0 \leq t \leq 19$	32'h5a827999
$20 \leq t \leq 39$	32'h6ed9eba1
$40 \leq t \leq 59$	32'h8f1bbcdc
$60 \leq t \leq 79$	32'hca62c1d6

Table 3.7 Round Function

Step Number	Function F(*B,C,D*)
$0 \leq t \leq 19$	$(B \wedge C) \oplus (\neg B \wedge D)$
$20 \leq t \leq 39$	$B \oplus C \oplus D$
$40 \leq t \leq 59$	$(B \wedge C) \oplus (B \wedge D) \oplus (C \wedge D)$
$60 \leq t \leq 79$	$B \oplus C \oplus D$

3.4.3 RIPEMD-160 Algorithm

The Race Integrity Primitives Evaluation Message Digest (RIPEMD)-160 algorithm generates a 160-bit hash code composed of five 32-bit words. RIPEMD-160 produces output in the little-endian format. Algorithm 1 shows the RIPEMD-160 algorithm.

Algorithm 1: RIPEMD-160 Algorithm

for t=0 { $A = H_0, B = H_1, C = H_2, D = H_3 E = H_4;$

$A' = H_0, B' = H_1, C' = H_2, D' = H_3, E' = H_4;$

for t=0 to 79 {

$T = rol_{s(t)}\left(A + f_t(B,C,D) + X[m(t)] + K(t)\right) + E;$

$A = E;\ E = D;\ D = rol_{10}(C);\ C = B;\ B = T;$

$T' = rol_{s'(t)}\left(A' + f_{79-t}(B',C',D') + X[m'(t)] + K(t)\right) + E';$

$A' = E';\ E' = D';\ D' = rol_{10}(C');\ C' = B';\ B' = T';$

}

$H_0 = H_1 + C + D';\ H_1 = H_2 + D + E';\ H_2 = H_3 + E + A';$

$H_3 = H_4 + A + B';\quad H_4 = H_0 + B + C';$ }

There are 80 steps of processing for five 32-bit different initial inputs. From Algorithm 1, five initial inputs H_0, H_1, H_2, H_3, and H_4 will be given to five inputs from the left, namely A, B, C, D, and E, as well as five inputs from the right, namely A', B', C', D', and E'. Two parallel processes of RIPEMD-160 occur during the execution.

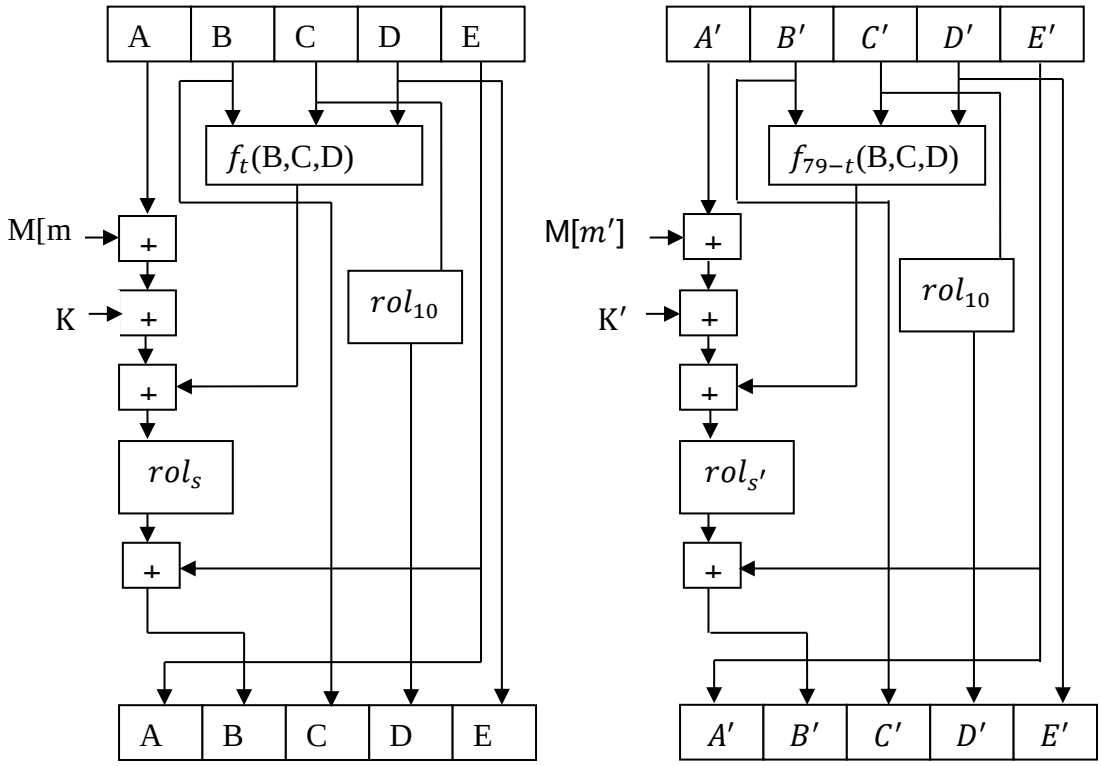


Figure 3.10 Compression Function of RIPEMD-160 Block Diagram

Figure 3.10 depicts the structural compression function of the RIPEMD-160 algorithm. The figure illustrates a non-linear function f(B,C,D) with inputs B, C, and D. The outputs D and D' are rotated to the left over 10 positions. Figure 3.11 illustrates the top level of the RIPEMD-160 algorithm; the initial input, input value for message selection, shift rotation, and constant are described in the following table. For the iterative design, only one block of compression function will be used in the process, while for the pipelined design, there are five blocks of compression functions, which consist of different non-linear functions *f*(*B,C,D*). From this figure, M and M' represent the 32-bit message input with a message selection for each step, ten constant inputs K and K' for five parallel operations, and a shift value for 80-step operation. Finally, five initial inputs, five variables A, B, C, D, and E from the left and five variables A', B', C', D', and E' from the right will provide the output of the hash function. The final result has to be reverted back into normal values to get the

correct output of RIPEMD-160 hash function as the output of RIPEMD-160 hash function is in the little-endian format.

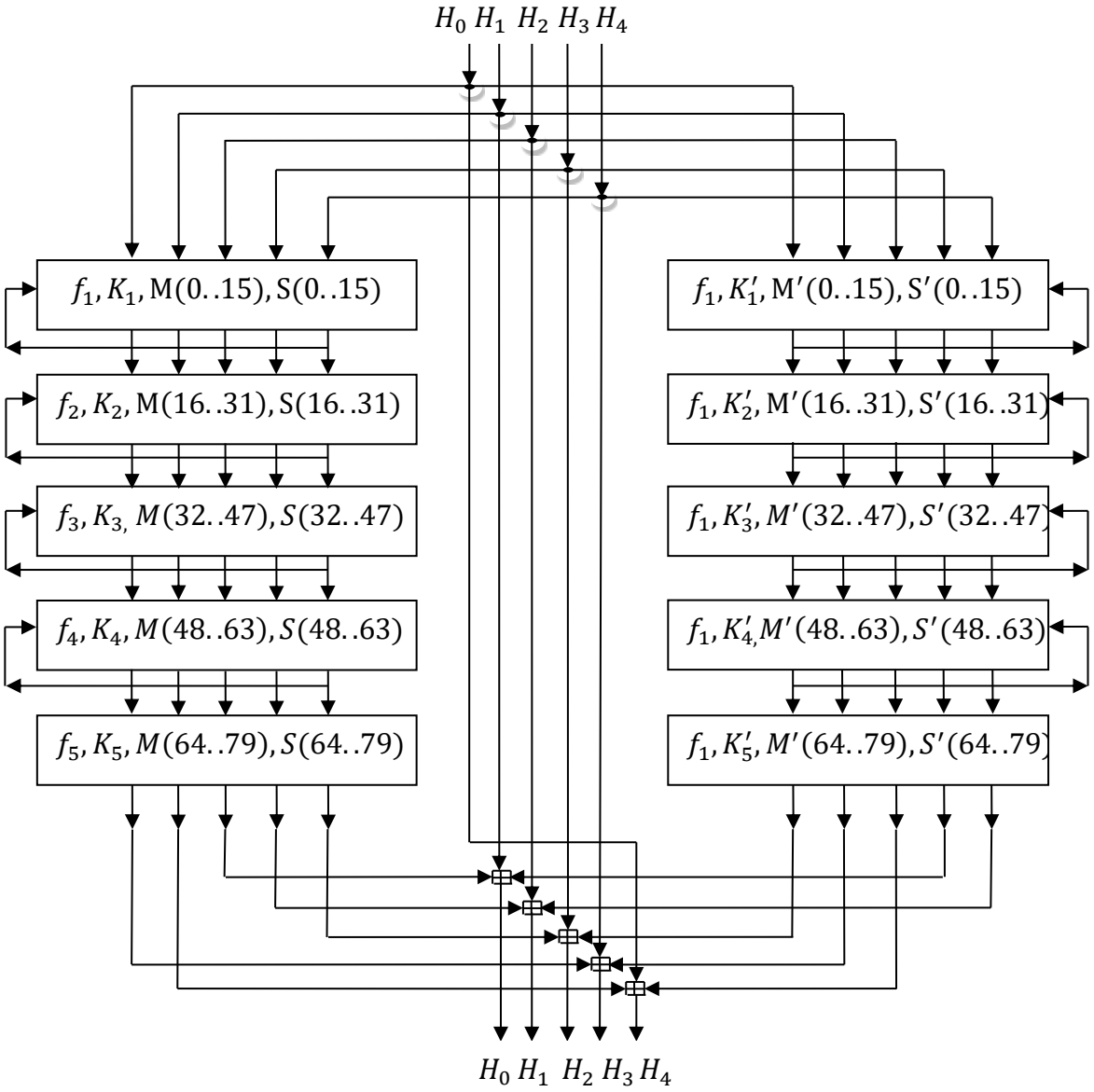


Figure 3.11 RIPEMD-160 Top-Level

The following tables show some parameter operations to describe RIPEMD-160. Table 3.8 presents the initial inputs for RIPEMD-160. There are five initial inputs: H_0, H_1, H_2, H_3, and H_4.

Table 3.8 Initial Value

Register	Buffer Initialisation
H_0	32'h67452301
H_1	32'hefcdab89
H_2	32'h98badcfe
H_3	32'h10325476
H_4	32'hc3d2e1f0

There are five different non-linear functions for *f*(B,C,D) of RIPEMD-160. Table 3.8 presents the non-linear functions for five different steps: f_1, f_2, f_3, f_4, and f_5. The operation $\oplus$, $\wedge$, $\vee$, and $\neg$ represent bitwise XOR, AND, OR, and complement, respectively. Table 3.9 shows the rounding of Boolean functions for both the left and right lines. The sequence of the functions will be based on round function, as shown in Table 3.10, where the right line will use the reverse order of non-linear function for the left line.

Table 3.9 Non-linear Function of RIPEMD-160

Step Number	Non-linear Function, *f*(B,C,D)	
$0 \leq t \leq 15$	f_1(B,C,D)	B $\oplus$ C $\oplus$ D
$16 \leq t \leq 31$	f_2(B,C,D)	(B $\wedge$ C) $\vee$ ($\neg$B $\wedge$ D)
$32 \leq t \leq 47$	f_3(B,C,D)	(B $\vee \neg$ C) $\oplus$ D
$48 \leq t \leq 63$	f_4(B,C,D)	(B $\wedge$ D) $\vee$ (C $\wedge \neg$ D)
$64 \leq t \leq 79$	f_5(B,C,D)	B $\oplus$ (C $\vee \neg$ D)

Table 3.10 Round Function

Lines	Round 1	Round 2	Round 3	Round 4	Round 5
Left	f_1	f_2	f_3	f_4	f_5
Right	f_5	f_4	f_3	f_2	f_1

To produce the output of RIPEMD-160 hash function, ten different constants, five from the left and five from the right lines, are used in this design. Only constant, K for $0 \leq t \leq 15$ and constant, K' for $64 \leq t \leq 79$ have a similar value, which is 32'h00000000. Table 3.11 shows five constants, K and K' for five rounds of RIPEMD-160 in hexadecimals.

Table 3.11 Constant of RIPEMD-160

Step Number	**Constant, K**	**Constant, K′**
$0 \leq t \leq 15$	32'h00000000	32'h50A28BE6
$16 \leq t \leq 31$	32'h5A827999	32'h5C4DD124
$32 \leq t \leq 47$	32'h6ED9EBA1	32'h6D703EF3
$48 \leq t \leq 63$	32'h8F1BBCDC	32'h7A6D76E9
$64 \leq t \leq 79$	32'hA953FD4E	32'h00000000

As RIPEMD-160 uses two parallel lines, which are from the left and right, there will be two different values for the message selection, m and m', as shown in Table 3.12. Accordingly, the data need to be kept in the memory first so that the data are already in the appropriate places. Once the data are already in proper places, the counter will be used to call the message based on the values from Table 3.12. There are sixteen sections to the messages, and each one comprises a 32-bit message word. In this design, there are five rounds for both lines; each round consists of 16 steps. There are 80 steps of processing to complete the output of this RIPEMD-160 design.

Table 3.12 Message Selection

Step Number	**Message Word, m**															
$0 \leq t \leq 15$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$16 \leq t \leq 31$	7	4	13	1	10	6	15	3	12	0	9	5	2	14	11	8
$32 \leq t \leq 47$	3	10	14	4	9	15	8	1	2	7	0	6	13	11	5	12
$48 \leq t \leq 63$	1	9	11	10	0	8	12	4	13	3	7	15	14	5	6	2
$64 \leq t \leq 79$	4	0	5	9	7	12	2	10	14	1	3	8	11	6	15	13
Step Number	**Message Word, m'**															
$0 \leq t \leq 15$	5	14	7	0	9	2	11	4	13	6	15	8	1	10	3	12
$16 \leq t \leq 31$	6	11	3	7	0	13	5	10	14	15	8	12	4	9	1	2
$32 \leq t \leq 47$	15	5	1	3	7	14	6	9	11	8	12	2	10	0	4	13
$48 \leq t \leq 63$	8	6	4	1	3	11	15	0	5	12	2	13	9	7	10	14
$64 \leq t \leq 79$	12	15	10	4	1	5	8	7	6	2	13	14	0	3	9	11

Similar to the message selection value, there are two Shift values: Shift from the left, s and Shift from the right, s'. Table 3.13 illustrates the 4-bit left rotation operation for RIPEMD-160, executed from both the left and right sides. By using the combination method, the message value and Shift value are integrated into one variable in the Verilog code, and the output of the design can be monitored easily.

Table 3.13 Shift Value

Step Number	Shift, s															
$0 \leq t \leq 15$	11	4	15	12	5	8	7	9	11	13	14	15	6	7	9	8
$16 \leq t \leq 31$	7	6	8	13	11	9	7	15	7	12	15	9	11	7	13	12
$32 \leq t \leq 47$	11	13	6	7	14	9	13	15	14	8	13	6	4	12	7	5
$48 \leq t \leq 63$	11	12	14	15	14	15	9	8	9	14	5	6	8	6	5	12
$64 \leq t \leq 79$	9	15	5	11	6	8	13	12	5	12	13	14	11	8	5	6

Step Number	Shift, s'															
$0 \leq t \leq 15$	8	9	9	11	13	15	15	5	7	7	8	11	14	14	12	6
$16 \leq t \leq 31$	9	13	15	7	12	8	9	11	7	7	12	7	6	15	13	11
$32 \leq t \leq 47$	9	7	15	11	8	6	6	14	12	13	5	14	13	13	7	5
$48 \leq t \leq 63$	15	5	8	11	14	14	6	14	6	9	12	9	12	5	15	8
$64 \leq t \leq 79$	8	5	12	9	12	5	14	6	8	13	6	5	15	13	11	11

3.4.4 SHA-256 Algorithm

The National Institute of Standards and Technology (NIST) standard specifies the use of secure hash algorithms including SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512 (Henriguez et al., 2006). Hash function techniques are employed in data transmission to generate the message digest. Consequently, it serves as a crucial instrument for embedding security in email, internet banking, and several other applications. A hash function converts an input of any length into a fixed-length result. A hash function is a non-reversible function; inverting a hash value to retrieve the original message input is challenging. Moreover, it is computationally impractical to identify a message that yields an identical hash value. These qualities are crucial for the effective functioning of a hash function. This study aims to present a high-speed hardware implementation of the SHA-256 algorithm. This algorithm is

developed and executed based on Arria II GX. The objective of this design is to enhance the performance of the SHA-256 algorithm.

SHA-2 comprises four distinct hash functions: SHA-224, SHA-256, SHA-384, and SHA-512. The output length of these hash algorithms is influenced by the SHA-2 specification, varying from 256 to 512 bits. This research presents the design of the SHA-256 hash function. The SHA-256 algorithm is elucidated in this section, along with its corresponding block diagram. The SHA-256 algorithm comprises two phases: pre-processing and hash computation. Pre-processing entails padding a message and parsing the padded message into m-blocks. Initialisation values are designated for usage in the hash computation. The hash computation generates a message schedule from the padded message. The hash value produced by hash computation is utilised to ascertain the message digest. Hash computation involves the repetitive generation of message schedules, functions, constants, and word operations to derive a hash value. Table 3.14 delineates the attributes of the SHA-256 hash function. The security of the SHA-256 hash function depends on the size of the hash value.

Table 3.14 Characteristics of SHA-256 Hash Function

Hash Function	**SHA-256**
Size of hash value (n)	256
Constants Kt number	64
Message block size (m)	512
Word size	32
Number of words	8
Digest rounds number	64

The first step of the SHA-256 hash algorithm involves pre-processing, during which the input message is padded. The message padding procedure commences upon receiving the message input, during which a single 1 bit is added at the end of the message. Subsequently, it is succeeded by n 0 bit until the message length is congruent to 448 modulo 512. The final 64 bits are allocated for determining the message length. Consequently, the total message input is 512 bits. Figure 3.12 depicts the message pre-processing of the SHA-256 hash function.

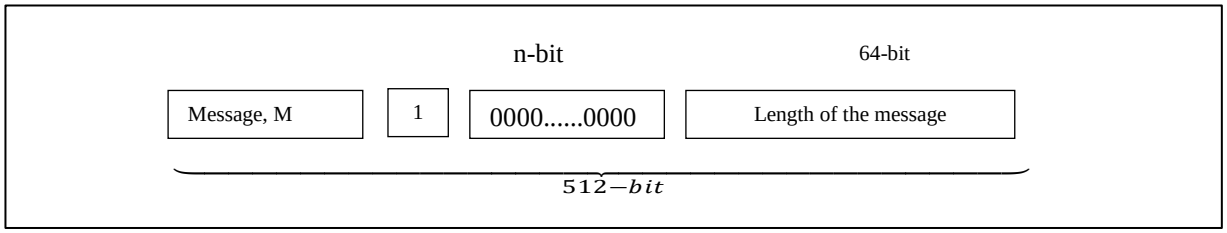


Figure 3.12 Message Preprocessing of SHA-256 Hash Function

Figure 3.13 illustrates the message scheduling of the SHA-256 algorithm. The SHA-256 message, W_t is calculated by the message scheduler, as illustrated in Figure 3.13. For $0 \le t \le 15$, a message is l-bit derived directly from the input message, but for $16 \le t \le 63$, a message W_t is computed using Equation 3.8. The variable t denotes the quantity of transformation rounds. $ROTR^n(x)$ denotes a right rotation of x by n bits, whereas $SHR^n(x)$ indicates a right shift of x by n bits, as seen in Equations 3.9 and 3.10.

Message schedule SHA-256, W_t:

$W_t =$ Message input $\quad 0 \le t \le 15$

$$W_t = \sigma_1^{256}(W_{t-2}) + W_{t-7} + \sigma_0^{256}(W_{t-15}) + W_{t-16} \qquad 16 \le t \le 63 \qquad \text{Equation 3.8}$$

Where:

$$\sigma_0^{256}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x) \qquad \text{Equation 3.9}$$

$$\sigma_1^{256}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x) \qquad \text{Equation 3.10}$$

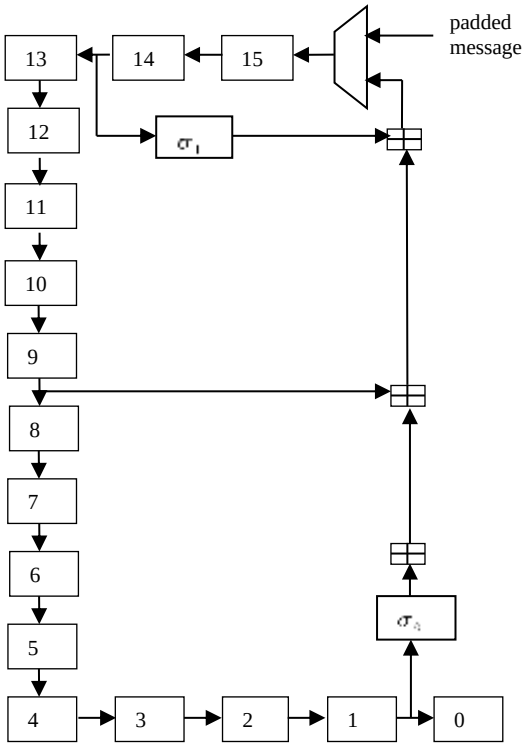


Figure 3.13: Message Schedule of SHA-256 Algorithm

Initial hash values, H_0 to H_7 as presented in Table 3.15, are allocated to the variables a, b, c, d, e, f, g, and h, correspondingly. The first hash values consist of 32-bit words.

Table 3.15 Buffer Initialisation of SHA-256 Algorithm

Register	Buffer Initialisation
H_0	32'h6a09e667
H_1	32'hbb67ae85
H_2	32'h3c6ef372
H_3	32'ha54ff53a
H_4	32'h510e527f
H_5	32'h9b05688c
H_6	32'h1f83d9ab
H_7	32'h5be0cd19

There are 64 values of the 32-bit constant K_t to process the SHA-256 hash function. Figure 3.14 shows the compression function of SHA-256, comprising four functions that start rounding from t = 0 until t = 63. The four functions are $Ch(x, y, z), Maj(x, y, z), \sum_0(x)$ and $\sum_1(x)$, as shown in the following Equations 3.11, 3.12, 3.13, and 3.14. Symbol $\wedge, \neg$, and $\oplus$ represent logical AND gate, NOT gate, and XOR gate, respectively.

$$Ch(e, f, g) = (e \wedge f) \oplus (\neg e \wedge g) \quad \text{Equation 3.11}$$

$$Maj(a, b, c) = (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c) \quad \text{Equation 3.12}$$

$$\sum_0(a) = ROTR^2(a) \oplus ROTR^{13}(a) \oplus ROTR^{22}(a) \quad \text{Equation 3.13}$$

$$\sum_1(e) = ROTR^6(e) \oplus ROTR^{11}(e) \oplus ROTR^{25}(e) \quad \text{Equation 3.14}$$

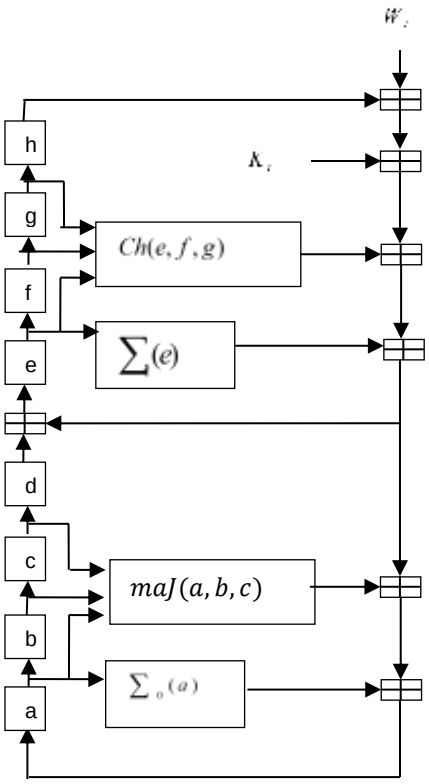


Figure 3.14 Compression Function of SHA-256 Algorithm

Hash computation produces eight variables a, b, c, d, e, f, g, and h with initial values to assess the four function equations 3.11, 3.12, 3.13, and 3.14, as shown in Figure 3.14. The 64 iterative operations comprise constant K_t and message input W_t. Equations 3.15, 3.16, and 3.17 yield hash values as output

$$Temp_1 = h + \sum 1(e) + Ch(e, f, g) + K_t + W_t \quad \text{Equation 3.15}$$

$$Temp_2 = \sum_0(a) + Maj(a, b, c) \quad \text{Equation 3.16}$$

$$\begin{aligned} h &= g \\ g &= f \\ f &= e \\ e &= d + Temp_1 \\ d &= c \\ c &= b \\ b &= a \\ a &= Temp_1 + Temp_2 \end{aligned} \quad \text{Equation 3.17}$$

The final hash value of SHA-256 is obtained in a big-endian format. Finally, hash values, H_0 to H_7, are calculated using the modulo-32-bit adders following 64 iterations.

$$H_0 = a + H_0, H_1 = b + H_1, H_2 = c + H_2, H_3 = d + H_3$$

$$H_4 = e + H_4, H_5 = f + H_5, H_6 = g + H_6, H_7 = h + H_7$$

Hence, the message digest of SHA256 algorithm $= H_0 \parallel H_1 \parallel H_2 \parallel H_3 \parallel H_4 \parallel H_5 \parallel H_6 \parallel H_7$

3.4.5 HMAC Algorithm

Message authentication is a procedure to ascertain the authenticity of messages. Two important aspects must be examined when verifying the message: the authenticity of the source and the integrity of the message. In conventional encryption, the sender and receiver

utilise a shared key for authentication during transmission. Consequently, only the genuine user possesses the capability to decrypt the message utilising the correct key. Figure 3.15 illustrates message authentication utilising Keyed-Hash Message Authentication Code (HMAC). It includes a secret key for generating data known as an MAC, which is attached to the message. The message and MAC are transmitted from the sender to the recipient. The receiver must compare the MAC to determine if it matches the one generated with the identical secret key. If the output MAC matches the transmitted MAC, it indicates that the message remains unaltered and is thus authentic.

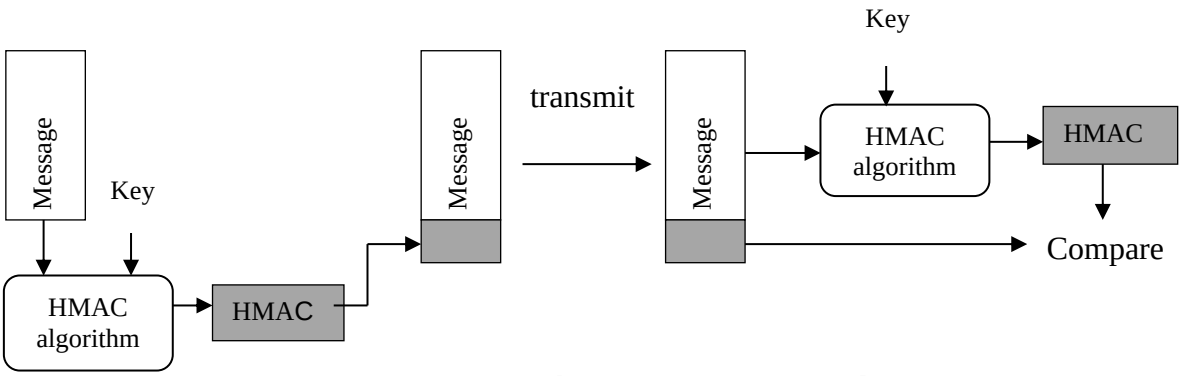


Figure 3.15 Message Authentication using Keyed- HMAC

Keyed-HMAC is an application that uses the hash function for verifying or authenticating the message. This research implements HMAC based on hash function. The process of HMAC implementation starts with hashing the input key already XOR with the inner pad (ipad) and concatenate with input text. The inner pad's value is 36 in hexadecimal format, utilising 64-byte times that are contingent upon the key's length. The output of this hash function serves as the input for the subsequent hash function to generate the output of the HMAC design. Prior to hashing the output of the first hash function, the input key must be XORed with the outer pad (opad), which is represented by the hexadecimal value 5C. The concatenation of the 64-byte key and opad output, along with the output of the initial hash function, will be hashed together to produce the HMAC output. Equation 3.18 demonstrates

the MAC applied to the data $text$ utilising the HMAC function, while Figure 3.16 depicts the architecture of HMAC employing a hash function.

$$\text{MAC}(text)_t = \text{HMAC}(K, text)_t$$

$$= \text{H}\Big((K_0 \oplus opad)||H\big((K_0 \oplus ipad)||text\big)\Big)_t \qquad \text{Equation 3.18}$$

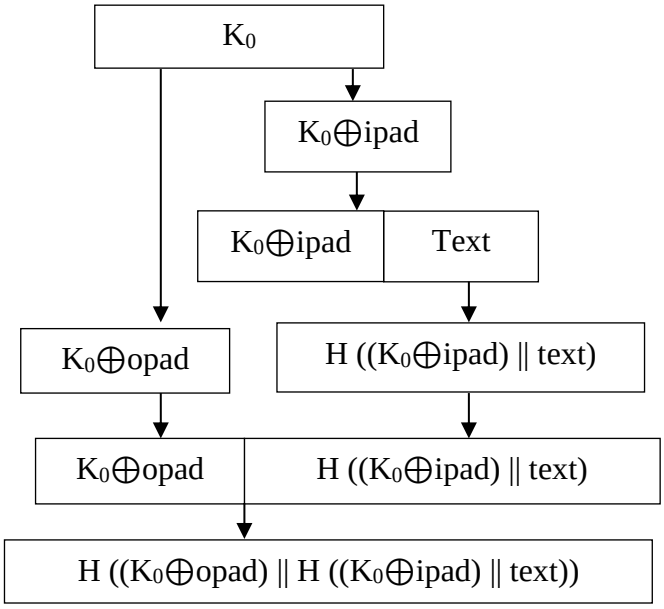


Figure 3.16 Illustration of HMAC construct

3.5 Proposed Design Methodology

This research involves the designing of hash functions utilising various approaches to improve performance. It employs a variety of techniques, including iterative processes, pipelining, unfolding, a combination of pipelining and unfolding, and low-power design. The main objective of developing various hash function designs through diverse methodology is to assess their performance regarding speed, area, and power consumption, employing hardware description language (HDL) coding techniques. The efficacy of these designs can be improved by utilising the Altera Quartus II advisor, the TimeQuest Timing Analyzer, the PowerPlay Power Analyzer, and modifying HDL coding styles. Typically,

iterative iteration is used in the standard design of hash functions. Figure 3.17 depicts the structure of iterative step function iteration, whereas Figure 3.18 shows the structure of inner-round pipelining for the step function hash function, incorporating additional registers at different stages of the same round. The construction of outer-round pipelining can be formed as illustrated in Figure 3.19.

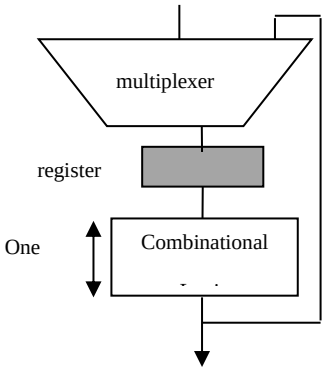


Figure 3.17 Iterative Looping (Henriguez et al., 2006)

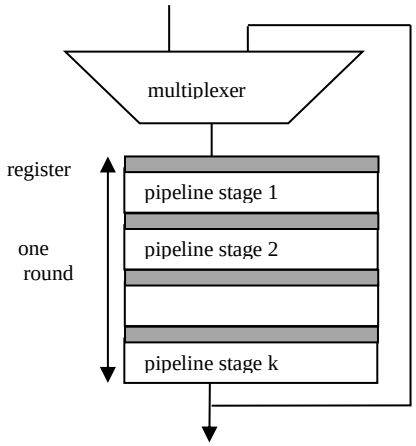


Figure 3.18 Inner-Round Pipelining (Henriguez et al., 2006)

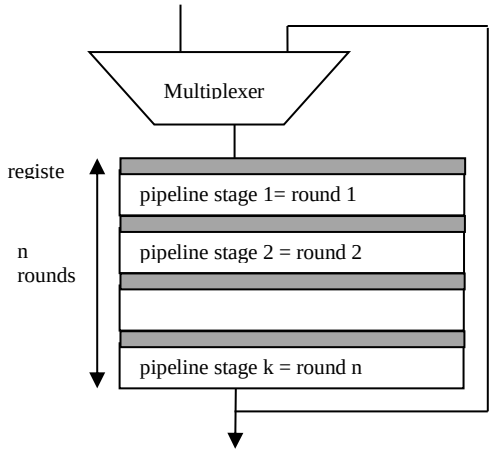


Figure 3.19 Outer-Round Pipelining (Henriguez et al., 2006)

Unfolding transformation technique has been proposed to increase the throughput of the hash function design. The unfolding algorithm is one of the techniques that digital signal processing (DSP) applications can use to generate a new programme that executes multiple iterations of the original programme. The unfolding factor, J, is introduced as the number of iterations. Figure 3.20 shows the structure of unfolding factor 2 of n rounds pipelining step function of hash function design. By applying these techniques to the hash function algorithm architecture, high-performance hash function can be implemented for system data transmission.

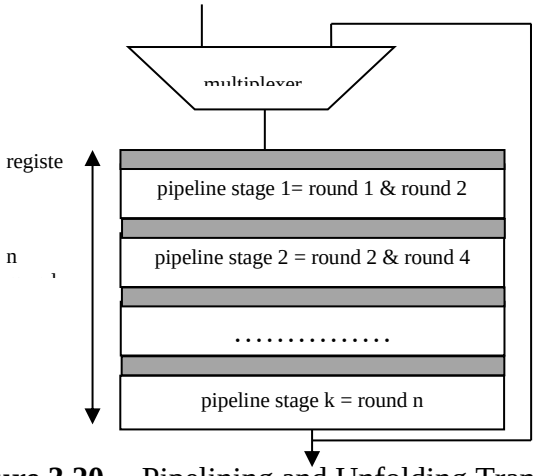


Figure 3.20 Pipelining and Unfolding Transformation

Several optimisation strategies can be used to improve the performance of cryptographic hash functions, especially with the increasing demand for high-speed and efficient digital systems. When developing robust and dependable hash function architectures, it is essential to achieve high throughput and fast execution time. The research has implemented two main methodologies such as unfolding transformation and low-power design techniques. These strategies were used in this research to optimise the implementation of hash functions on FPGAs. These methods are designed to enhance critical performance parameters, including power efficiency, maximal operating frequency, and logic area utilisation. It is important to note that reducing power consumption not only enhances energy efficiency but also enhances overall performance, particularly in portable and embedded systems. In CMOS-based FPGA designs, total power dissipation is usually divided into three components: short-circuit power, static (leakage) power, and dynamic power. Among these, dynamic power, which results from signal switching activity, is the most important during active operation. It is affected by signal transitions, clock frequency, and load capacitance.

3.5.1 Unfolding Transformation Design Technique

The unfolding algorithm is one of the methods that DSP applications can employ to create a new programme that executes more than one iteration of the original programme. Furthermore, J specifies the number of iterations from the initial programme. The rules of unfolding algorithm are explained as follows (Parhi, 1999):

1. For each node U in the original DFG, draw the J nodes U_0, U_1, U_2.
2. For each edge $U \rightarrow V$ with ω delay in the original, draw the J edges $U_i \rightarrow V_{(i+\omega)\%J}$ with $\left\lceil \frac{i+\omega}{J} \right\rceil$ delays for $i = 0,1 \ldots\ldots, J-1$.

To explain the structure of unfolding algorithm, an example of DSP programme is shown in Equation 3.19 and Figure 3.21.

$y(n) = ay(n-9) + x(n)$ Equation 3.19

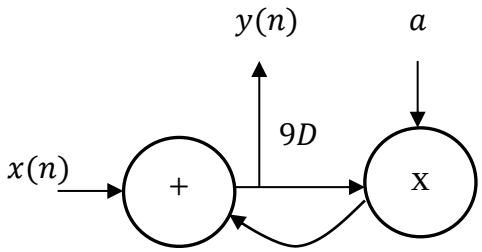


Figure 3.21 An Example of an Original DSP Programme (Parhi, 1999)

Data flow graph can be constructed based on Figure 3.22, which is an example of an original DSP programme, by replacing the input and output ports with nodes A and B, while the addition and multiplication processes are represented by nodes C and D, respectively, as shown in Figure 3.22.

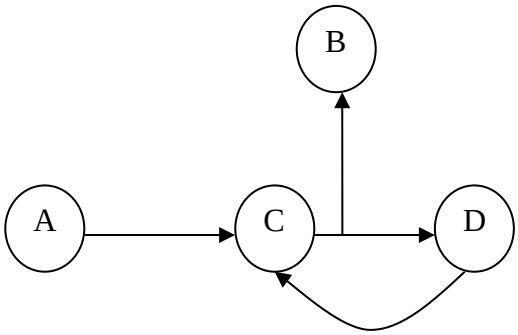


Figure 3.22 A DFG Corresponding to a DSP Programme (Parhi, 1999)

Based on the first rule of unfolding algorithm, there are eight nodes that represent $i = 0, 1$, namely $A_0, B_0, C_0, D_0, A_1, B_1, C_1$, and D_1. The second step of unfolding algorithm is to connect each edge $U \rightarrow V_{\text{in}}$ to the DSP programme. The edge $U \rightarrow V$ with no delay is divided into two parts, $U_0 \rightarrow V_0$ and $U_1 \rightarrow V_1$. Therefore, the edge $C \rightarrow D$ with $\omega = 9$ delays becomes $C_0 \rightarrow D_{(9+0)\%2}$ with $\left\lceil \frac{9+0}{2} \right\rceil$ delays and $C_1 \rightarrow D_{(9+1)\%2}$ with $\left\lceil \frac{9+1}{2} \right\rceil$ delays. Figure 3.23 shows the unfolded DFG corresponding to the 2-unfolded DSP programme. Finally, the 2-unfolded DFG is created with $C_0 \rightarrow D_1$ with 4 delays and $C_1 \rightarrow D_0$ with 5 delays.

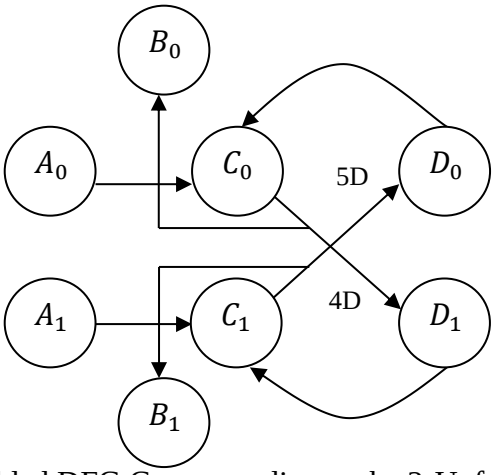


Figure 3.23 The Unfolded DFG Corresponding to the 2-Unfolded DSP Programme (Parhi, 1999)

3.5.2 Low Power Design Technique

Two main techniques are utilised to minimise dynamic power consumption in hash function hardware design are state encoding and clock gating. Individual states can be encoded using a variety of signal encoding techniques, including Gray encoding, one-hot encoding, and binary encoding. Encoding signals is a technique to reduce the power consumption of a system featuring a finite state machine. In this study, only two types of signal encoding are used: binary and Gray encoding. First, a comparison is conducted between Gray code and binary code in the context of state encoding. Gray code offers advantages by limiting signal transitions to a single bit per clock cycle, in contrast to binary encoding, which can result in multiple bit changes, such as potentially up to three in some instances. This reduction in bit transitions results in lesser switching activity, which effectively reduces dynamic power. Gray coding, in contrast to binary encoding, demonstrates a decrease in power consumption because state transitions necessitate only a single bit of bit toggling. Figure 3.24 illustrates the state encoding for both binary and Gray encoding. Consider A as an input and E as the last sequence of the system. In binary encoding, the FSM starts from states 0, 1, 2, 3, and 4, but in Gray encoding, the state

sequence commences from 0, 1, 3, 4, and 6. For each transition, the state of Gray signal encoding switches 1 bit.

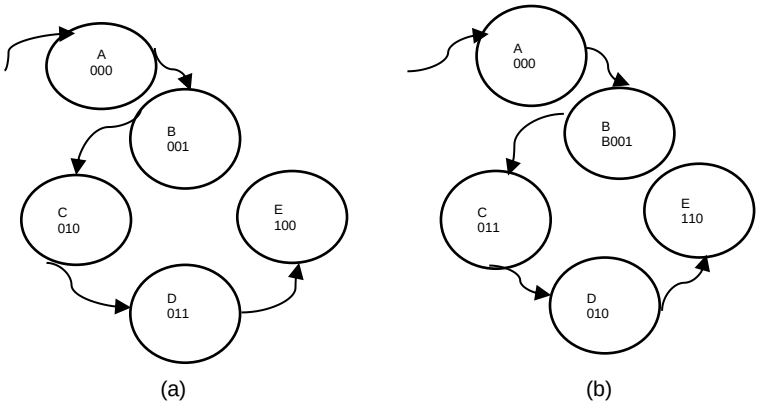


Figure 3.24 State Encoding: (a) Binary Encoding and (b) Gray Encoding

Secondly, clock gating functions by deactivating the clock signal to functional blocks that are not currently involved in processing. This method significantly reduces power usage by preventing redundant signal toggling within idle modules, as shown in the Figure 3.25. The power efficiency of the hardware design is considerably improved by the combination of these two techniques, which reduce the overall switching frequency and minimise unnecessary logic activity throughout the circuit. Furthermore, gating the clock at the input of the Step Function module is one of the techniques to save dynamic power consumed within the system on chip (SoC). In this research, two types of clock gating are designed for both Binary encoding with clock gating and Gray encoding with clock gating. Input enable is inserted together with the clock to convert them into the clock-gated implementation, as shown in Figure 3.25.

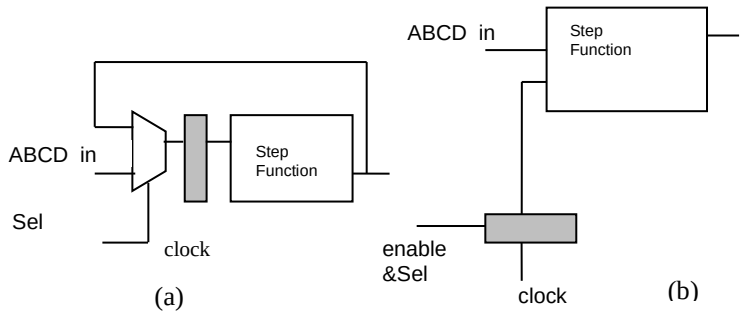


Figure 3.25 Gating the Clock: (a) Normal and (b) Clock-Gated Implementation

3.6 Modification to Hash Function and HMAC Design

This research particularly investigates the development of four hash functions: MD5, SHA-1, SHA-256, and RIPEMD-160. The performance of the design is improved through the implementation of a variety of techniques. Various approach designs are employed for each of the hash functions, as illustrated in Figure 3.26. These include iterative computation, pipelining, unfolding, a combination of pipelining and unfolding, loop unrolling, and a low-power technique.

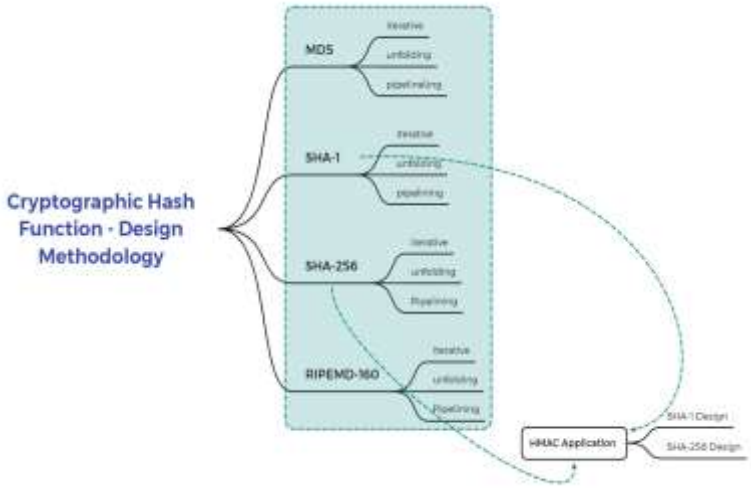


Figure 3.26 Design Methodology of Hash Function

3.6.1 MD5 Design

This research involves the development of iterative MD5 architectures, which encompass four distinct types of MD5 designs: MD5 Binary, MD5 Gray, MD5 Binary with Gating, and MD5 Gray with Gating designs. The main goal of developing a variety of MD5 designs is to assess their efficiency in terms of speed, area, and power consumption by employing the HDL coding language. The performance of these designs can be improved by utilising the Altera Quartus II advisor, the TimeQuest Timing Analyzer, the PowerPlay Power Analyzer, and modifying HDL coding styles.

3.6.1.1 MD5 Iterative and Low Power Design

Figure 3.27 illustrates the MD5 Binary design's top-level architecture. It comprises 12 modules. StepFunction_MD5 module comprises the core of these modules.

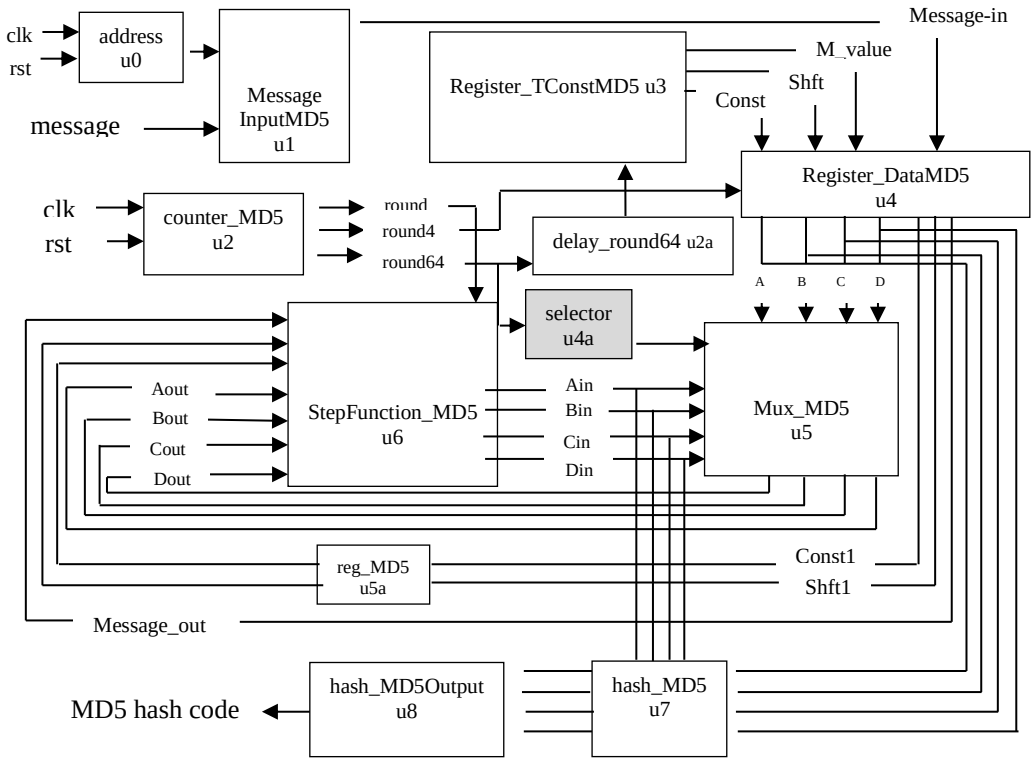


Figure 3.27 Top-Level Architecture of MD5 Binary Design

This module will continually execute four non-linear functions to generate the hash code. The input of the MD5 Binary module is also significantly influenced by the Selector module. This implies that the data will be assigned the initial values of A, B, C, and D. Data from Register_Data MD5 must be sequentially executed with Message_in.

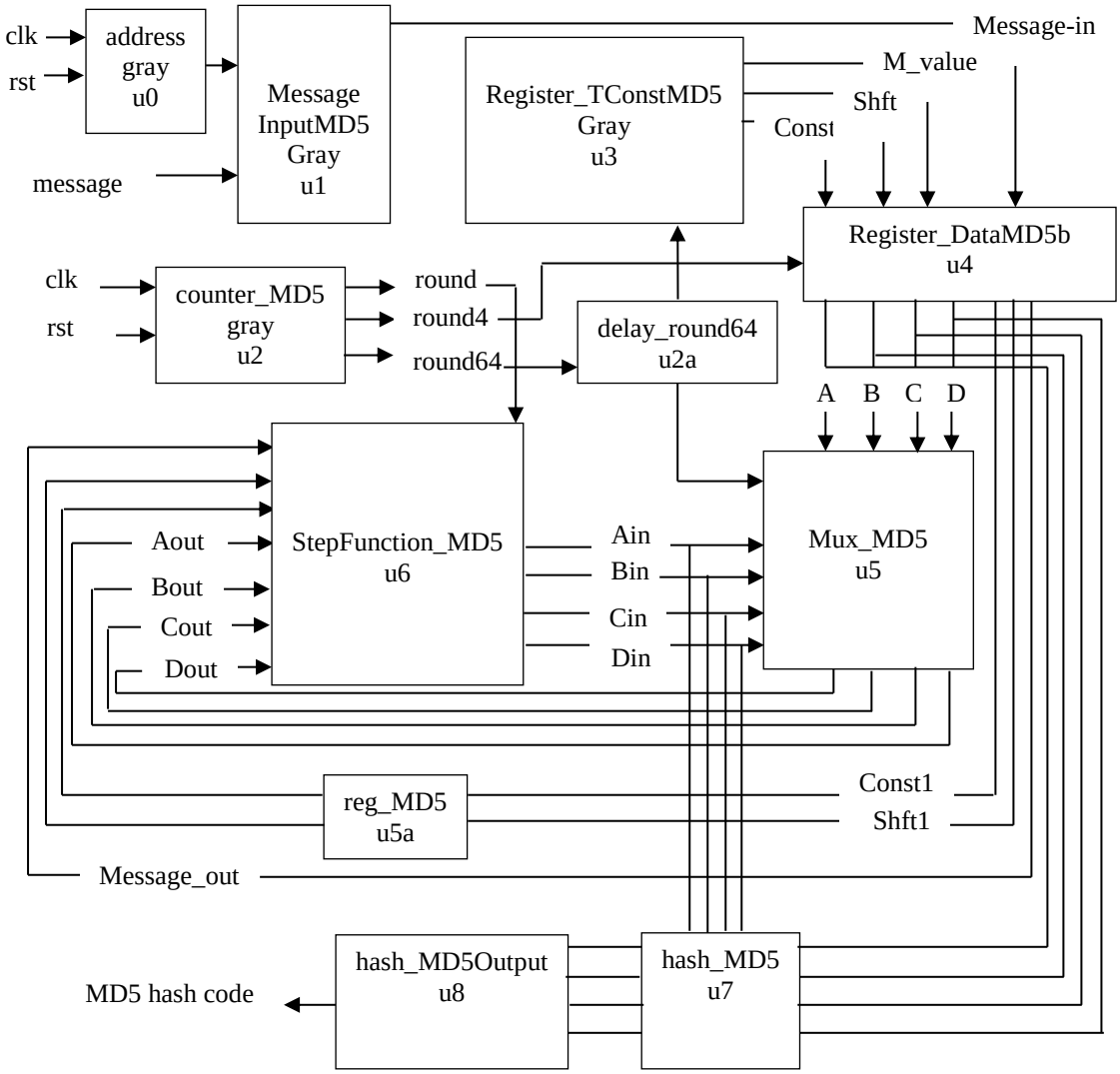


Figure 3.28 Top-Level Architecture of MD5 Gray Design

Figure 3.28 illustrates the top-level architecture of the MD5 Gray design, which regulates the multiplexer selector. The architecture of MD5 Gray design is nearly identical to the MD5 Binary design. Two differences distinguish MD5 Gray from MD5 Binary design: the finite state machine (FSM) for MD5 Gray is written in Gray encoding, and the selector

input for the multiplexer is derived directly from the delay_round64 module. Given that the FSM sequence is sequential, the selector module can be utilised to control input selection in the MD5 Binary design. For example, the seven states for binary encoding are 1, 2, 3, 4, 5, 6, and 7, whereas the sequence for Gray encoding is 0, 1, 3, 2, 6, 7, 5, and 4. Furthermore, delay_round64 module must contain the round values.

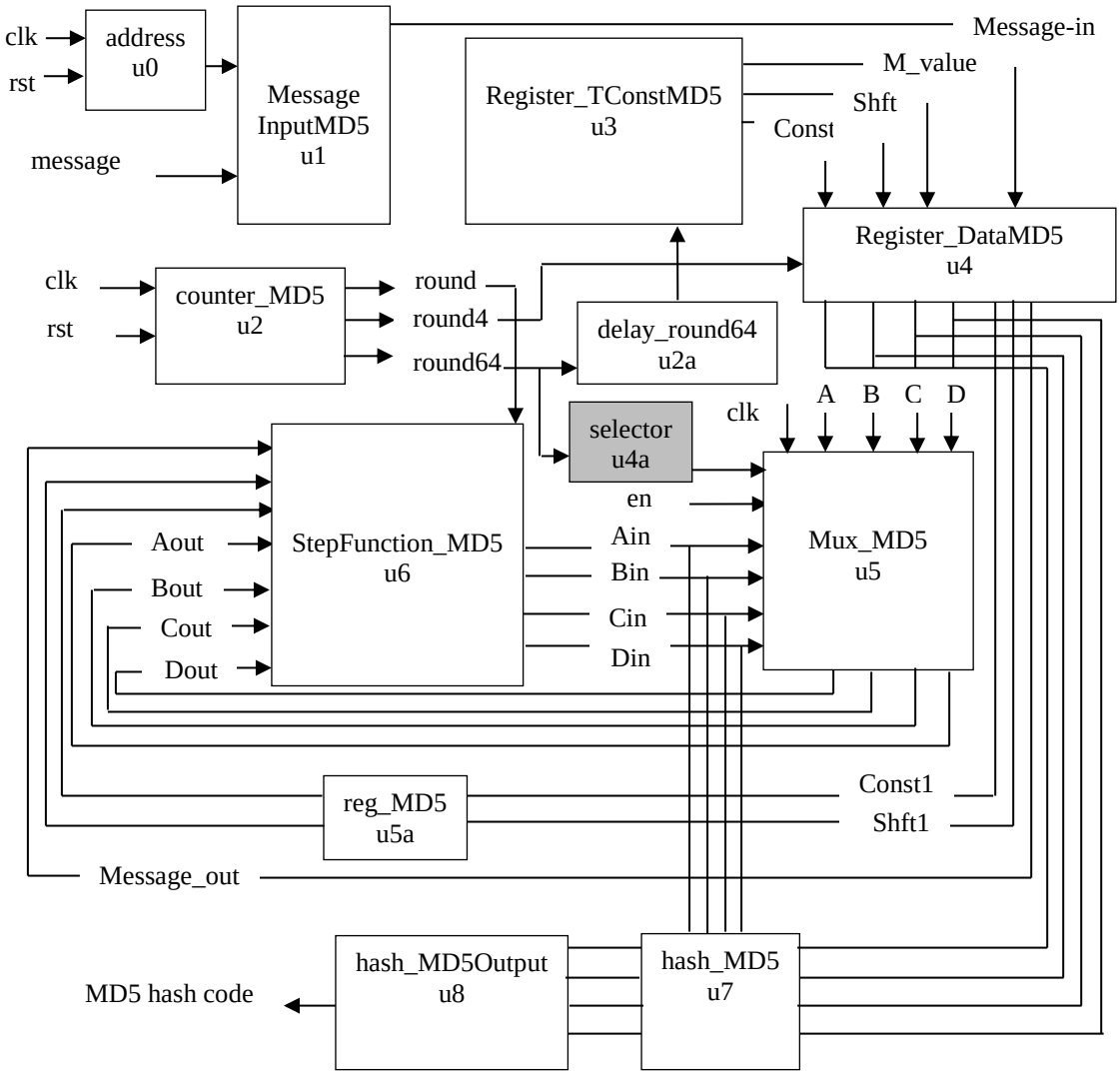


Figure 3.29 Top-Level Architecture of MD5 Binary with Gating Design

Figure 3.29 shows the top-level architecture of MD5 binary with gating design. This architecture is similar to MD5 binary design; all the FSM structures are designed in binary encoding. Additional modification in this architecture is clock gating. One of the low-power

design techniques is switching activity. It happens during signal toggling. Two types of switching activities are Gray and clock gating. From this figure, there is an “en” input, which is ANDed together with clk input. By gating the clock to the MD5 binary with gating design, the power consumption can be reduced.

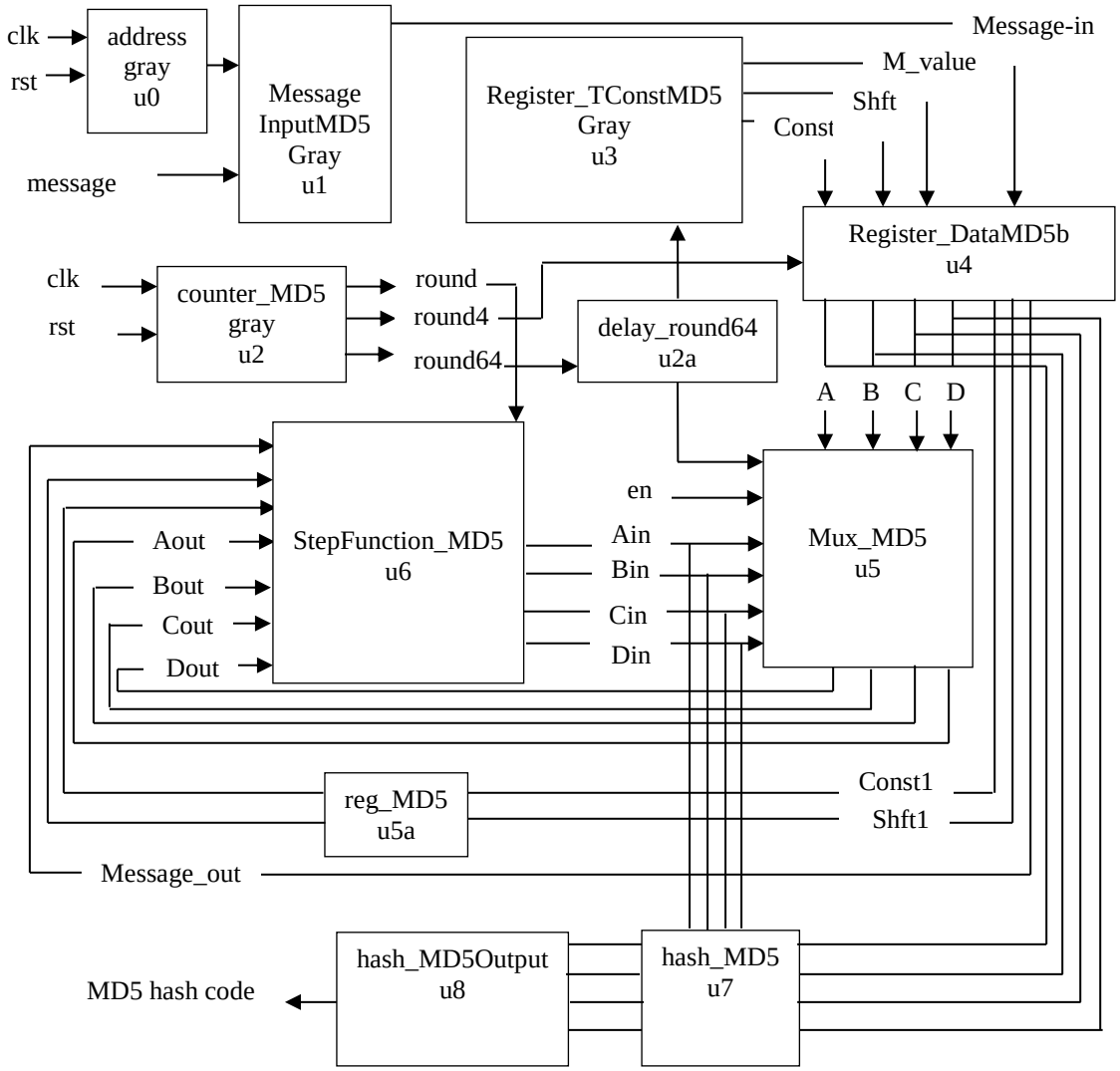


Figure 3.30 Top-Level Architecture of MD5 Gray with Gating Design

Figure 3.30 shows the top-level architecture of MD5a Gray gating design. This MD5 architecture of FSM is designed based on Gray encoding. This architecture design is similar to MD5 Gray design. The difference between this design and the MD5 Gray design lies in the clock gating methodology. As previously noted, this method serves to reduce power

consumption. Consequently, by implementing clock gating within this architecture during signal toggling, power consumption can be reduced. The “en” input is ANDed together with clock input to reduce power consumption.

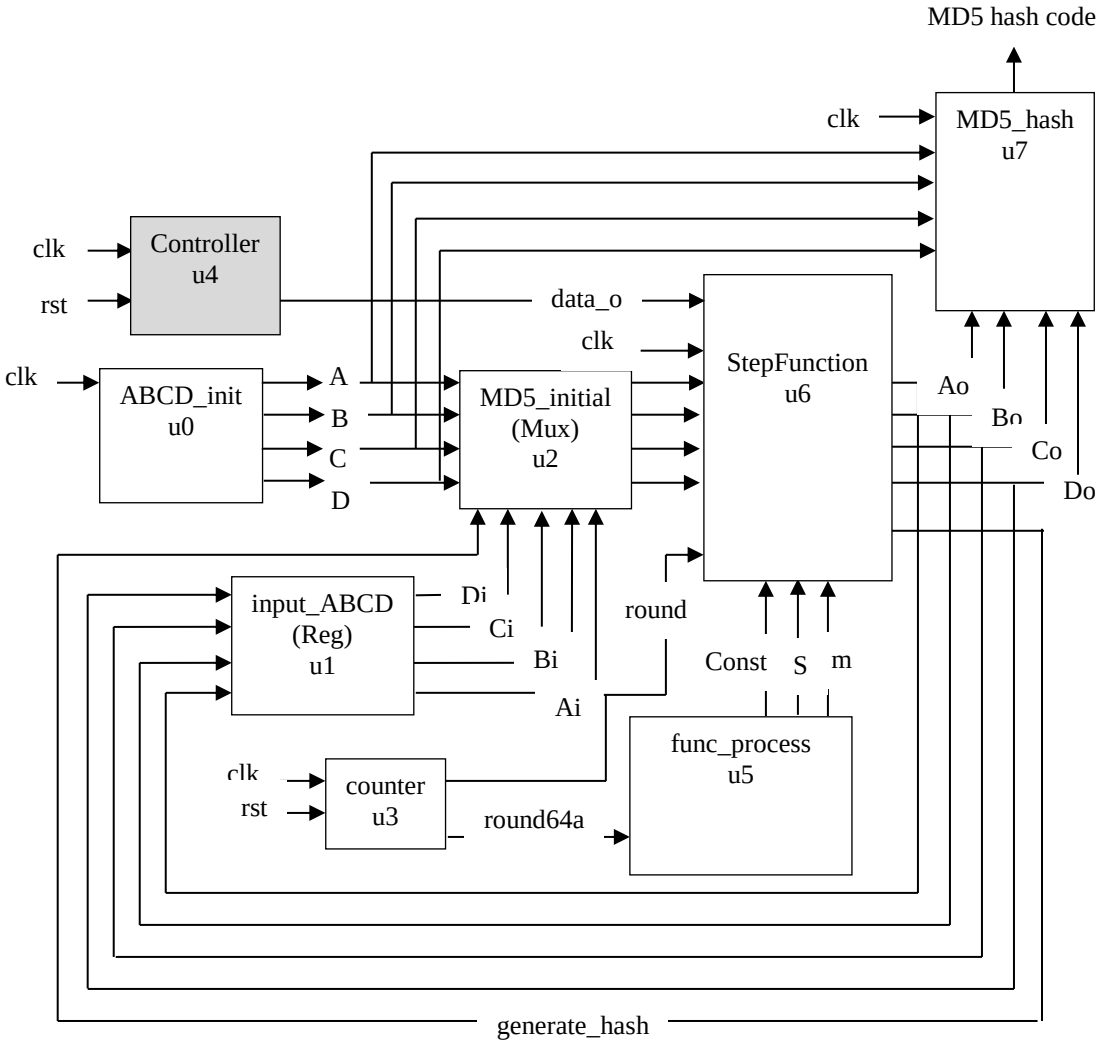


Figure 3.31 Top-Level Architecture of MD5 Binary with Controller Design

Figure 3.29 shows the hardware implementation of the top-level architecture of MD5 Binary with controller design. This design is slightly similar to the previous MD5 binary design with minor modification. The main differences between this MD5 binary with controller design and MD5 binary design are the controller module, counter module, and modification in the StepFunction module. In this architecture, the input message is already

kept in the memory to generate the output hash code. The main objective of implementing this architecture on the FPGA board is to ensure that the controller can provide data and verify the output of hash code.

MD5 architecture design consists of several block modules. One of the important block modules is the architecture of MD5 compression function. It is designed based on the basic architecture, which is known as iterative looping. Iterative looping refers to the looping of a designated block of compression function modules for a predetermined number of iterations. In this research, the output of MD5 design will be fed back to the input until 64 rounds, depending on the number of MD5 architecture. This type of design utilises a small logic area but consumes more clock cycle. The architecture of the MD5 compression function significantly influences the throughput of the resulting hash. Verilog code is employed to develop the MD5 logic design. Figure 3.32 illustrates the architecture of the MD5 compression function.

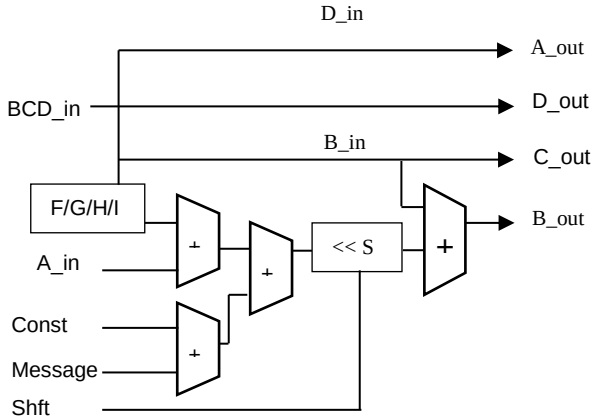


Figure 3.32 MD5 Compression Function Architecture

MD5 compression function is the main operation for MD5 algorithm implementation, and 64 steps are running iteratively. During the procedure, four non-linear functions, F, G, H, and I, are executed sequentially based on the MD5 compression function.

This compression function is the most important module in designing the MD5 hash function; the procedures of the four functions F, G, H, and I must be executed iteratively to complete the overall hash processing. The architecture of MD5 design has been successfully synthesised using Quartus II Altera. Modification to the structure of compression function can improve the maximum frequency of the design; the registers are added at the end of the design. Besides, Altera Quartus II CAD tools play an important role to produce better results, such as the TimeQuest Timing Analyzer and advisors. There are three types of advisors: resource, timing, and power optimisation advisors. MD5 design uses two types of signal encoding: binary and Gray encoding to evaluate the effect of MD5 hash function during the synthesis and implementation processes based on FPGA. There are three main inputs to the step function module: message, constant, and shift values. M_value, Const, and Shft represent these inputs, respectively, as shown in Equation 3.20.

TConst = {Const [43:12], Shft [11:4], M_value [3:0]} Equation 3.20

After designing the MD5 compression function, the next stage is to construct the architecture of MD5 design. The MD5 architecture commences with four initial inputs and four distinct non-linear functions: F, G, H, and I. The 32-bit input of MD5 algorithm is used in_mem [511:0] memory to keep the 448-bit data; the data for the last 64 bits are calculated by checking each byte of the message input. In this research, the input is padded with a padding unit until 448-bit because the message length can only be calculated after 448-bit. The counter message calculates the length of message in the last 64-bit, and the overall resulting message input is 512 bits.

In this research, the top level of MD5 design consists of various types of modules. All these modules are connected from the input module to the output module of the design.

The MD5 design process is mostly executed by the counter MD5 module, as depicted in Figure 3.33. This counter module controls the input message of MD5. The 32-bit message is segmented into four 8-bit messages to ascertain the message length. The final 64-bit represents the message's bit length. Consequently, the counter counts the message during round 14 of round 4. The entire bit message is calculated byte by byte.

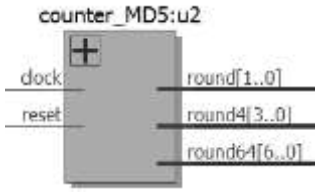


Figure 3.33 Counter MD5 module

Altera Quartus II has obtained the maximum frequency of the design during synthesis and implementation, and the design must meet the requirement of timing analysis. Thus, the analysis of the clock constraint needs to be considered. Based on Altera Quartus II as the CAD tool of the design, the TimeQuest Timing Analyzer is considered as the best solution in giving the clock constraint to the design. The timing must be met to avoid violation in the design to obtain positive slack. The setup and hold slack value are automatically calculated by CAD tools in determining the differences between data arrival and data required.

The overview of the hash function concept is based on the Altera Quartus II CAD tool, which generates the top level of MD5 design when no syntax errors occur throughout the synthesis and implementation processes. The inputs are as follows: clk, rst, load, and message [31:0]. If a message is present, the load input is set to logic “1” to receive the 32-bit input message into the MD5 design module. The message procedure loops indefinitely until 64 rounds are completed. Finally, the MD5 architecture design generates a hash output of 128 bits.

3.6.1.2 MD5 Unfolding Factor 2 Design

Figure 3.34 shows the top-level architecture of unfolding MD5 design with factor 2. There are six modules in this MD5 architecture: ABCD_init, input_ABCD, MD5_initial, func_process, getdata, and MD5_hash module. The top-level unfolding MD5 architecture has 32 rounds of non-linear functions. Each non-linear function consists of eight rounds, where two hash operations will be processed in one cycle. All of these operations will be executed in the getdata module. Some modification has been done in the getdata module. Two non-linear functions, two messages, and two constants will be executed in parallel form in order to apply the unfolding method to this MD5 algorithm. The func_process module will generate the data for message, shift, and constant values. The MD5 initial module is used to select the input ABCD for MD5 algorithm. First, the MD5 algorithm starts with the data from the ABCD_init module; subsequently, the hash operation will use the data from the input_ABCD module.

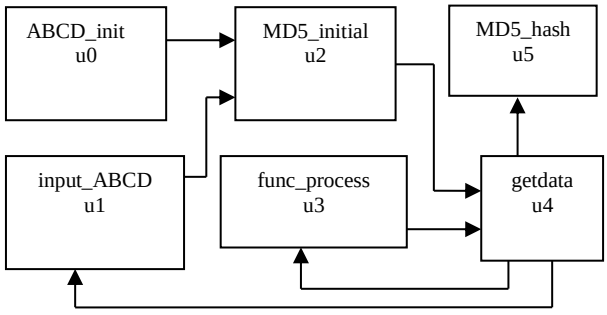


Figure 3.34 Top Level of Unfolding MD5 Design

This section describes the unfolding transformation applied to the MD5 hash function with an unfolding factor of two, which aims to improve throughput and reduce delay in hardware implementation. Figure 3.35 shows how the unfolding technique allows two non-linear function computations to be performed simultaneously within a single clock cycle, reorganising the original iterative MD5 design. The parallelisation achieves a twofold

increase in functional throughput relative to the baseline sequential design, thereby enhancing performance significantly.

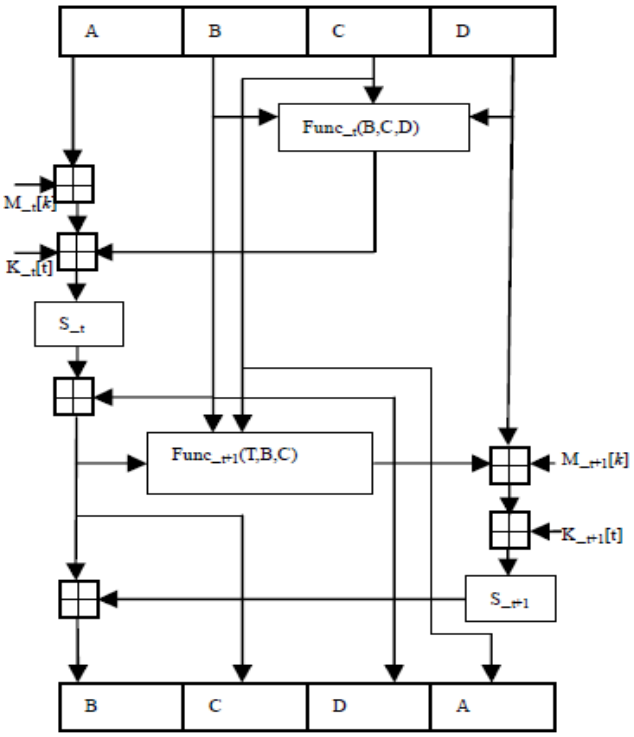


Figure 3.35 Compression Function of Unfolding MD5 Design

The unfolded MD5 hash function design with an unfolding factor of 2, where two non-linear functions are executed in parallel per clock cycle to improve throughput are shown in the Equation 3.21. This equation computes the intermediate value T based on the MD5 function applied to inputs B, C, and D, followed by message word M[K], constant T[i], and a left rotation by S. Then, followed by Equation 3.22 continues the processing by computing Temp using a subsequent function and message word M[i+1]. The final output (A_t, B_t, C_t, D_t) update the state registers with the next values after two operations are completed in parallel as shown in Equation 3.23. Finally, the state variables A, B, C, and D are rotated to prepare for the next cycle. This transformation reduces the number of required

clock cycles and enhances pipeline efficiency, contributing to higher data throughput in the MD5 hardware implementation.

$$T = B + ((A + Func(B,C,D) + M_i[K] + T[i]) << S) \quad \text{Equation 3.21}$$

$$Temp = T + ((D + Func(T,B,C) + M_{i+1}[K] + T[i]) << S) \quad \text{Equation 3.22}$$

```
A_t <= C_next;

B_t <= func_out1;

C_t <= func_out;

D_t <=B_next;
```

Equation 3.23

$$A = C, B = Temp , C = T , D = B$$

3.6.1.3 MD5 Pipelining and Unfolding Design

A combination of pipelining and unfolding method can improve the performance of MD5 algorithm. Figure 3.36 illustrates the architecture of MD5 compression function with pipelining and unfolding transformation techniques. This research proposes a novel combination of pipelining and unfolding transformation to enhance the throughput of the MD5 design. The proposed method achieves better performance because it can provide high throughput of MD5 algorithm based on FPGA. In this design, four registers are required to implement four-stage pipelining, and 32 registers are required to implement 32-stage pipelining. By applying this technique to the MD5 algorithm, the performance of MD5 design increases significantly. Modification on the getdata module has been done to obtain the pipelining and unfolding design for four-stage pipelining with unfolding transformation and 32-stage pipelining with unfolding transformation. The rest of the modules remain unchanged with some modifications in terms of input and output data.

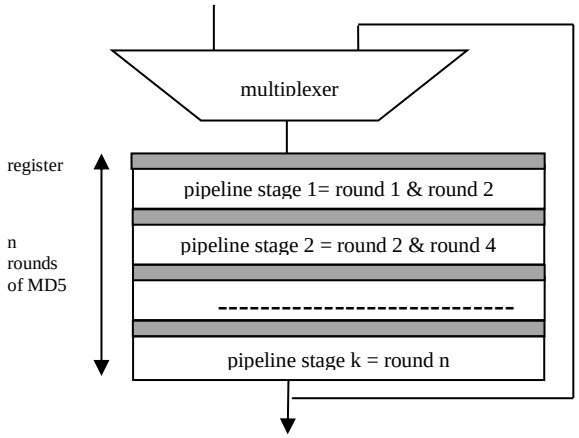


Figure 3.36 MD5Pipelining and Unfolding Compression Function

3.6.2 SHA-1 Design

The most significant module is StepFunction1, where all the operations that result in the hash code output take place in this module. Four distinct categories of non-linear functions are also described within this module. The StepFunction1 module must be rounded by 80 rounds in order to obtain the hash code for the output. Consequently, the counter for this module will accumulate to 80 rounds.

3.6.2.1 SHA-1 Iterative Design

The top-level architecture of the SHA-1 design is depicted in Figure 3.37. The diagram illustrates eleven modules for the SHA-1 design. One coder module is present within this architecture. The primary purpose of the coder module is to partition the output counter into four distinct counts, with 20 cycles comprising each count. Thus, it is possible to generate a constant value for each tally obtained from the coder output. The SHA-1 design ultimately generates a 160-bit hash code.

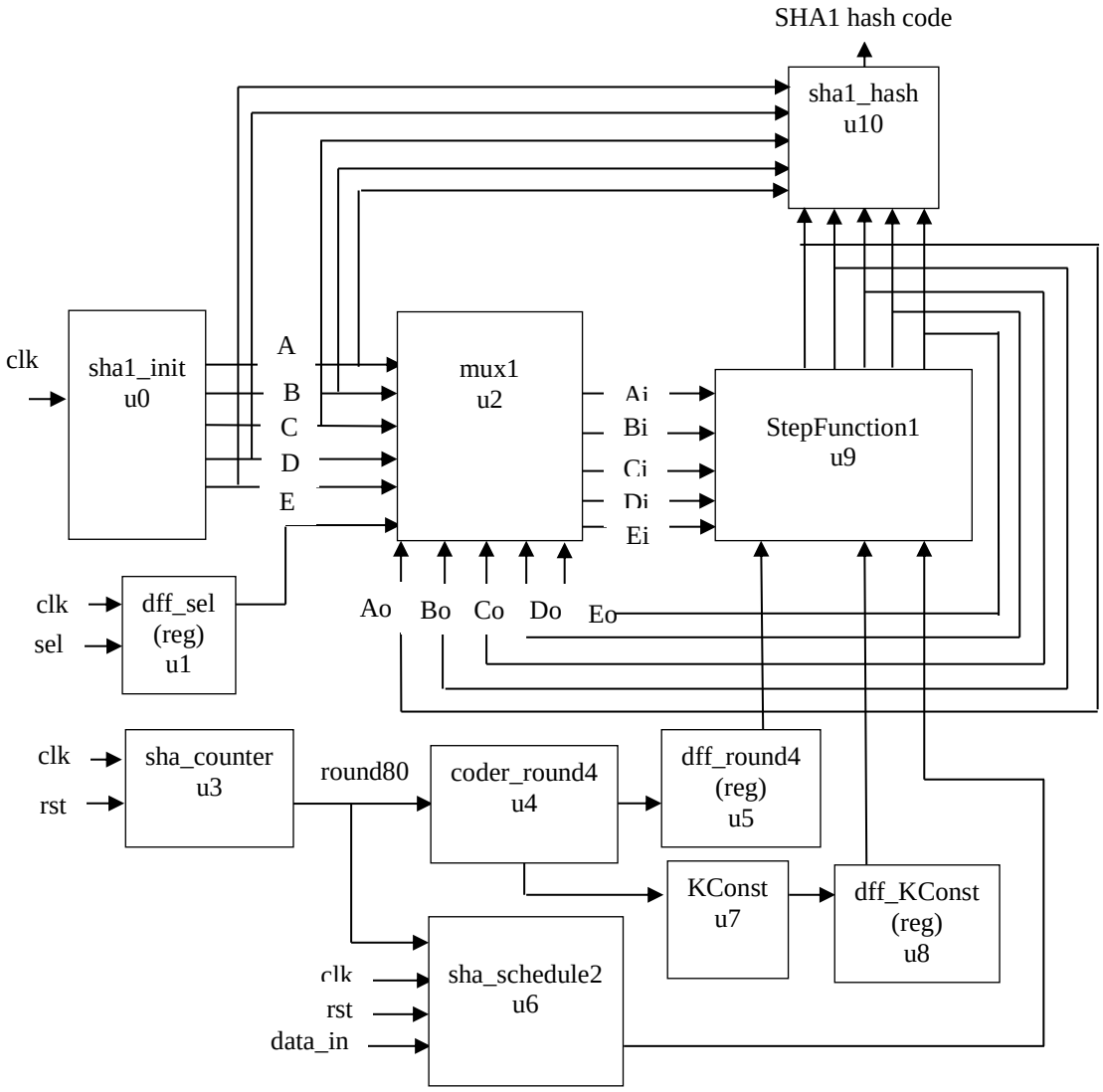


Figure 3.37 Top-Level Architecture of SHA-1 Design

3.6.2.2 SHA-1 Unfolding Factor 2 Design

The efficient implementation of cryptographic hash functions on reconfigurable hardware is a problem that requires solution. Consequently, the high performance of hash function design regarding speed and area is crucial for enhancing the throughput of the hash function, as current security systems require rapid implementation. Maximum frequency needs to be considered in designing hash function to increase the speed of the design. Besides, an efficient hash function also has small area implementation. Thus, to improve the

speed and area, this study proposes the architecture of SHA-1 with unfolding factor 2 transformation. Furthermore, this method can be applied to various hash functions to enhance the design's efficiency.

The proposed SHA-1 unfolding algorithm with factor 2 is shown in Figure 3.38. It consists of two non-linear functions with three different inputs: two circular left shifts of both A and B by 30, two circular left shifts of A by 5, and Temp by 5. From this figure, there are eight addition operations, which perform in parallel form during execution. Thus, the critical path of the design has only four addition processes. In other words, two hash operations are executed per cycle. This process reduces the number of cycles from 80 cycles to 40 cycles to obtain the final output hash.

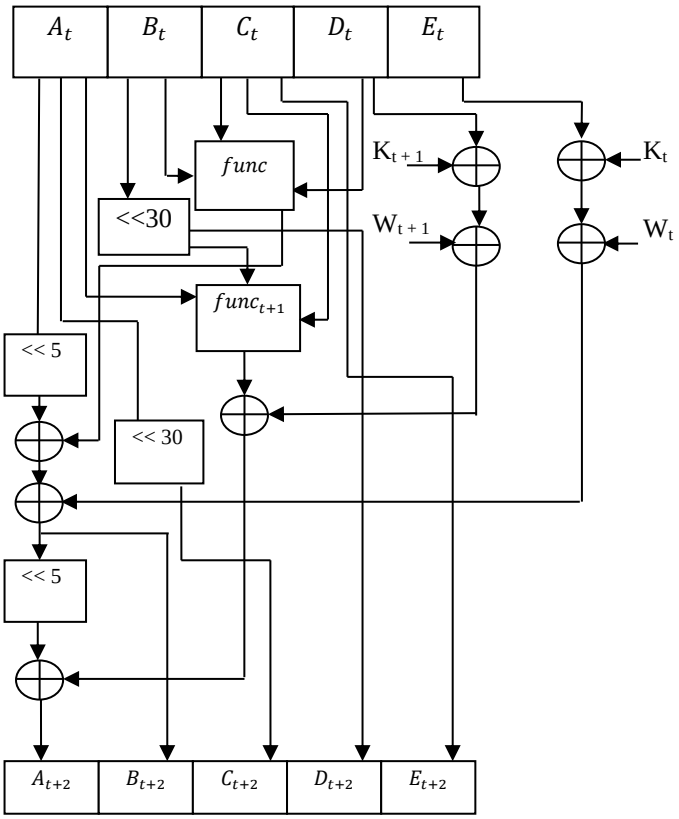


Figure 3.38 SHA-1 Unfolding Compression Function

Therefore, the SHA-1 hash function's throughput can be enhanced by the unfolding transformation. The following equations show the outputs of the SHA-1 unfolding algorithms. $\boldsymbol{ROTL^{a}(b)}$ represents the circular left shift or left rotation operation of $\boldsymbol{b}$ by $\boldsymbol{a}$ position to the left, and$\boldsymbol{func_t(p,q,r)}$ means a non-linear function at time $\boldsymbol{t}$ for three different inputs $\boldsymbol{p}, \boldsymbol{q}$, and $\boldsymbol{r}$. Equation 3.24 illustrates the output $\boldsymbol{A_{t+2}}, \boldsymbol{B_{t+2}}, \boldsymbol{C_{t+2}}, \boldsymbol{D_{t+2}}$, and $\boldsymbol{E_{t+2}}$ of SHA-1 unfolding compression function.

$$Temp = ROTL^{5}(A_t) + func(B_t, C_t, D_t) + E_t + W_t + K_t$$

$$A_{t+2} = ROTL^{5}\{Temp\} + func_{t+1}(A_t, ROTL^{30}(B), C_t) + D_t + W_{t+1} + K_{t+1}$$

$$B_{t+2} = Temp$$

$$C_{t+2} = ROTL^{30}(A_t) \qquad \text{Equation 3.24}$$

$$D_{t+2} = ROTL^{30}(B_t)$$

$$E_{t+2} = C_t$$

Figure 3.39 shows the top-level architecture of SHA-1 unfolding design. From this figure, there are ten modules of SHA-1 unfolding design. In this design, the main module is func module. All the processes of the SHA-1 structure are stated in this module. Additionally, this module contains four non-linear functions of SHA-1. The SHA-1 unfolding design is distinguished from the SHA-1 design in that the structure of the former is folded by factor 2. In other words, the clock cycle of the design is reduced from 80 cycles to 40 cycles. The operation executes two hash operations per cycle. The most important part is by adding the constant value and message value to two units, as shown in Figure 3.38. Message_unfolding module and Kt module produce two parallel outputs for message and constant, respectively. Thus, the operation executes in a parallel form to obtain the hash code.

Other modules are similar to those of the SHA-1 design. The maximum frequency of the design can be enhanced by placing the register at the end of the sha1 module, the final module of SHA-1 unfolding. The SHA-1 design is written in the big-endian format. Therefore, the output of design remains in its normal format.

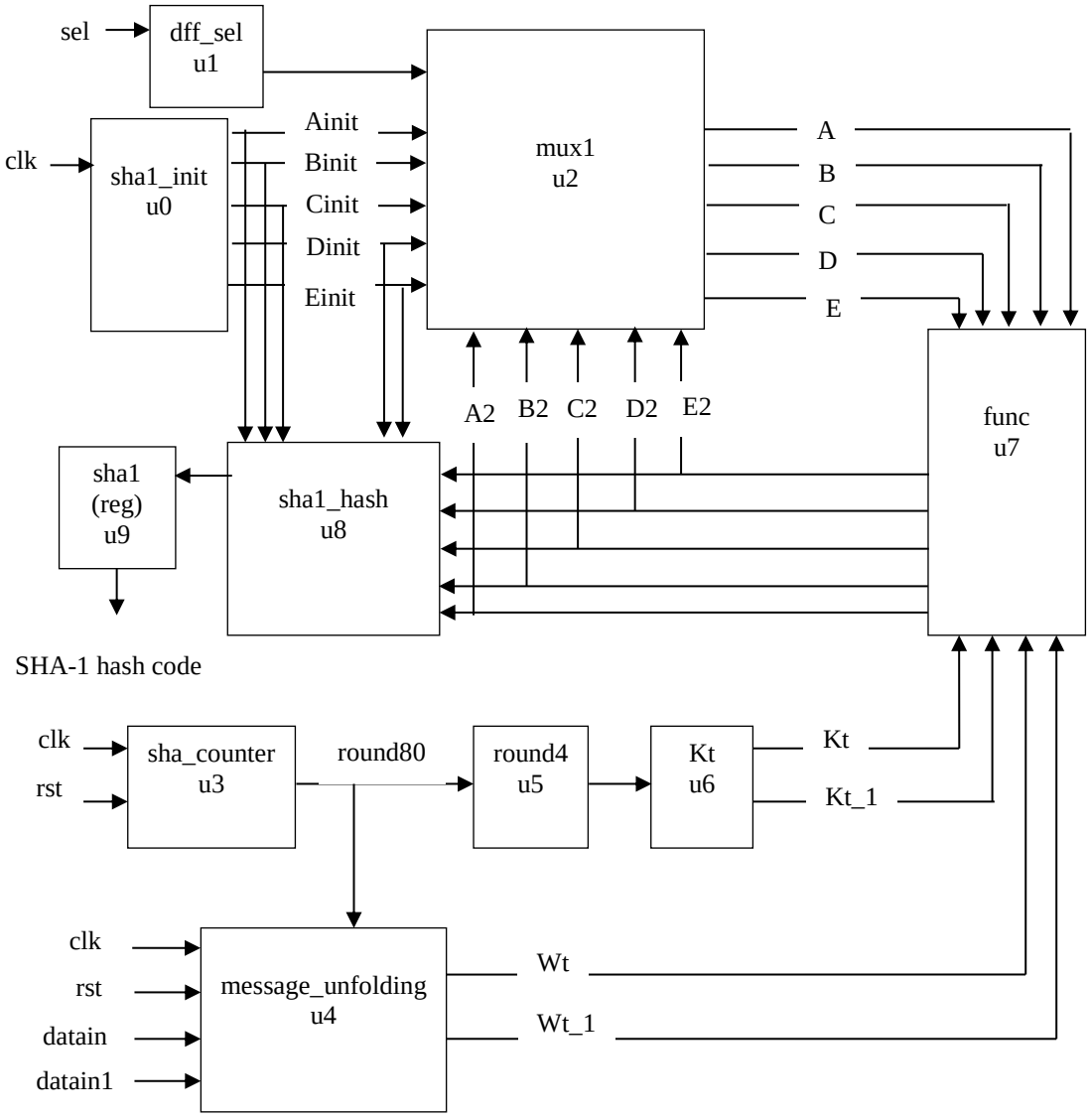


Figure 3.39 Top Level Architecture of SHA-1 Unfolding Design

3.6.2.3 SHA-1 Inner Pipelining Design

Figure 3.40 shows the top-level design of SHA-1 inner pipelining. There are eight modules in this architecture. The structure of the inner pipelining design focuses on the

StepFunction module, where the module consists of clock in one round. The difference between iterative design and inner pipelining design is that the combinational logic of the StepFunction module is separated with the register in the iterative design, while in inner pipelining, the register is used in designing the StepFunction module with only one round. In other words, the iteration process generates based on the number of SHA-1 rounds. Similar to other SHA-1 designs, the design starts with the sha1_init module to generate buffer initialisation. The sha_counter module will count the number to 80 rounds to produce the sequence of sha_schedule module, which is the message of SHA-1 design. From this figure, the coder_round4 module is used to divide the 80 rounds output counter into four parts. The output from the coder_round4 module can be used to generate the KConst module as there are four 32-bit constants in the SHA-1 algorithm. Additionally, this output has the potential to produce a non-linear function of the SHA-1 algorithm. The SHA-1 algorithm comprises four non-linear functions. Thus, it is easy to produce the output of StepFunction with different non-linear functions. Finally, SHA-1 hash code is obtained by adding the five-buffer initialization to the last output round of StepFunction.

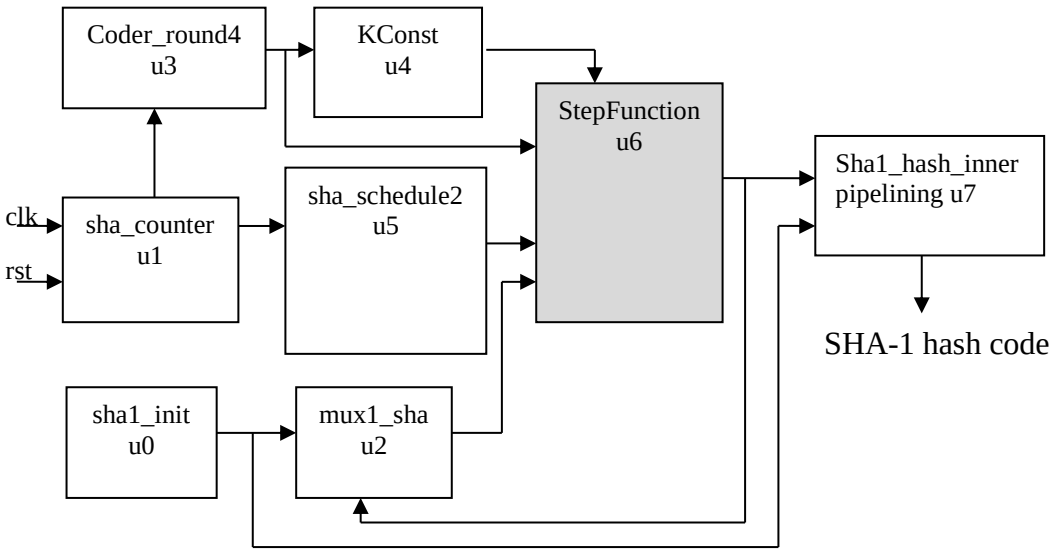


Figure 3.40 Top Level of SHA-1 Inner Pipelining Design

3.6.2.4 SHA-1 Four-stage Pipelining Design

Figure 3.41 shows the top level of SHA-1 four-stage pipelining design.

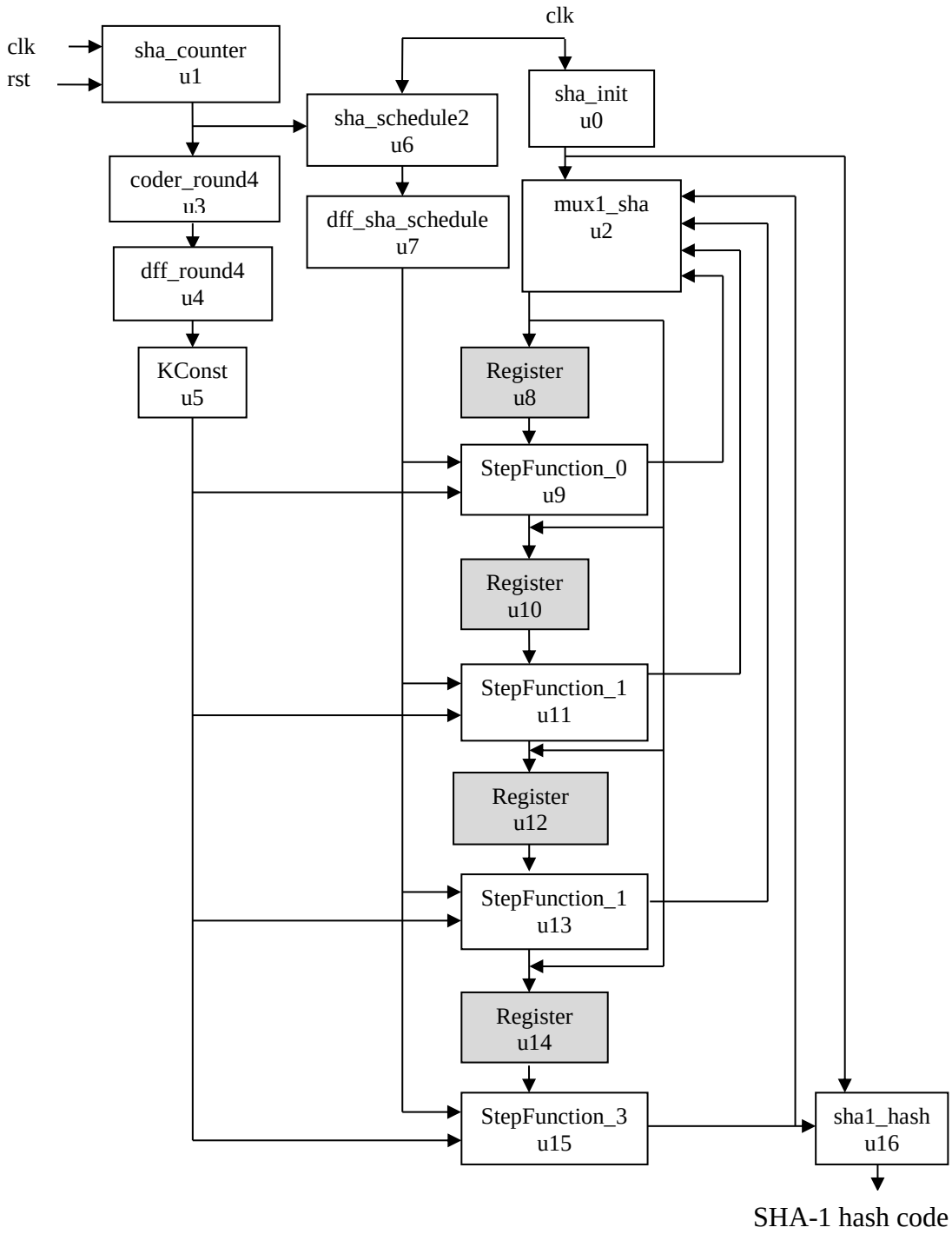


Figure 3.41 Top Level of SHA-1 Four-stage Pipelining Design

The StepFunction module is divided into four different modules, where each non-linear function consists of different non-linear functions. This top-level SHA-1 four-stage pipelining contains four non-linear functions: StepFunction_0 module, StepFunction_1 module, StepFunction_2 module, and StepFunction_3 module. The rest of the modules are similar with the previous design of SHA-1 architecture. The sha_counter module will give the output until 80 rounds. Therefore, the output of this counter can be divided into four parts to generate constant SHA-1 algorithm from the KConst module. In this architecture, two D flip-flops are used to obtain the correct output at the appropriate time, such as the dff_round4 module and the dff_sha_schedule module. By adding this module to this SHA-1 design, the result can be synchronised. The mux1_sha module is employed to determine the outcome of various non-linear functions within the SHA-1 algorithm. This design undergoes 80 cycles of iteration. Next, the sha1_hash module will generate a SHA-1 hash code. The result is obtained by combining the initial inputs A, B, C, D, and E with the final round of the SHA-1 algorithm in this module.

3.6.2.5 SHA-1 40-stage Pipelining Design

Figure 3.42 presents the top level of SHA-1 40-stage pipelining design. There are 40 modules of StepFunction, where each module consists of two rounds of SHA-1 algorithm non-linear function. The coder_round40 will control the process of StepFunction from StepFunction_1 until StepFunction40. Similar to the previous SHA-1 design, D flip-flops are used in this design to synchronise the SHA-1 design. Finally, the sha1 hash code is obtained after adding the initial value to the last output round. The process of SHA-1 algorithm rounds up to 80 rounds.

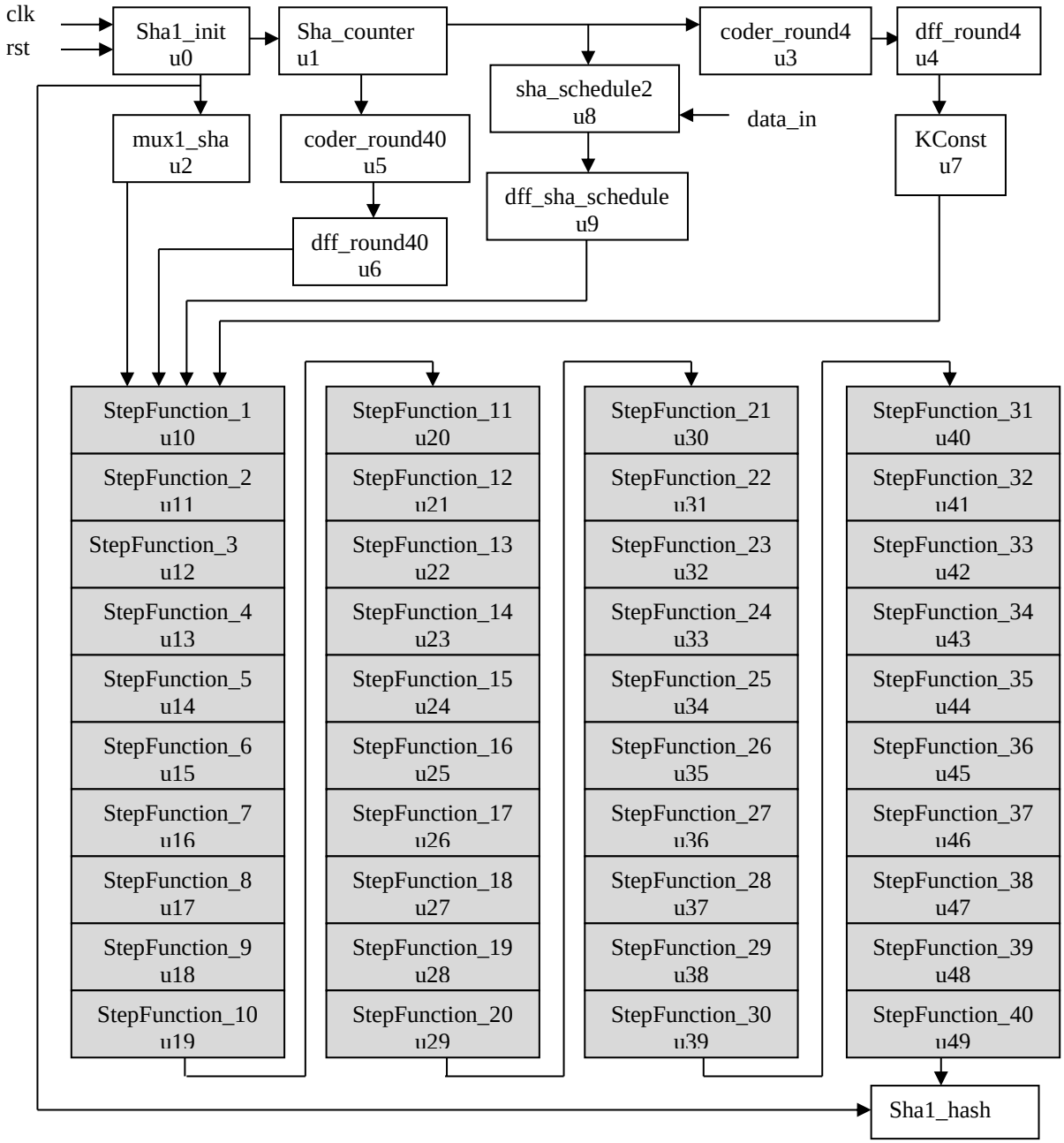


Figure 3.42 Top Level of SHA-1 40-stage Pipelining Design

3.6.2.6 SHA-1 Unfolding 40-stage Pipelining Design

Figure 3.43 shows the top level of SHA-1 unfolding 40-stage pipelining design. Similar to SHA-1 40-stage pipelining, there are 40 modules of SHA-1 unfolding 40-stage pipelining.

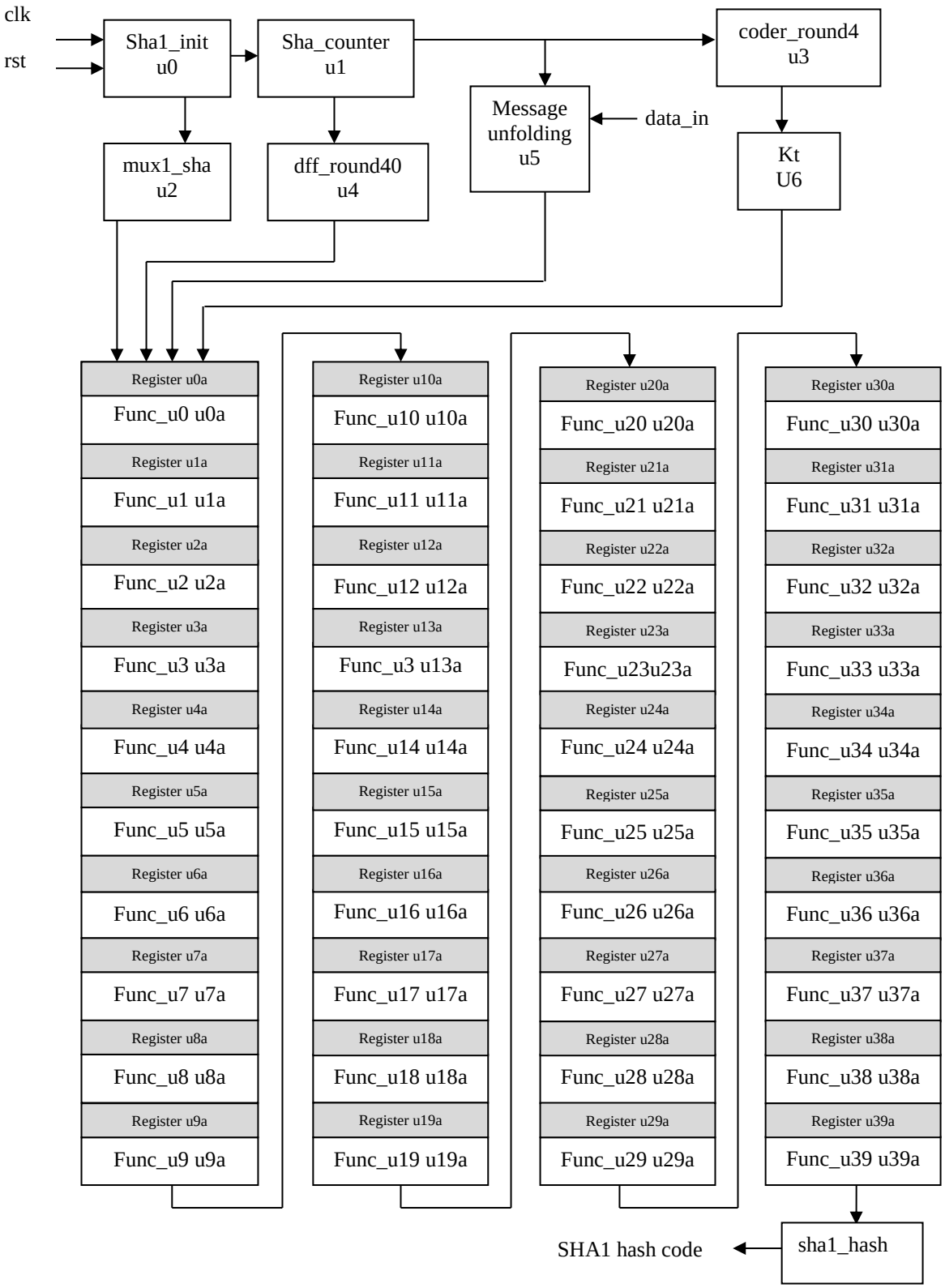


Figure 3.43 Top Level of SHA-1 Unfolding 40-stage Pipelining Design

This approach is further enhanced by combining unfolding with pipelining, as illustrated in Figure 3.43. Pipeline registers are systematically placed between non-linear function blocks based on the number of necessary pipeline stages. This facilitates the concurrent processing of multiple message blocks, thereby reducing idle cycles and enhancing data flow within the architecture. The combination of unfolding and pipelining decreases the total clock cycles needed for each message digest while enhancing the maximum operating frequency. High-performance cryptography applications on FPGA platforms might benefit from the overall design's increased throughput and decreased latency. However, the architecture of the Func module, which is a compression function of non-linear function SHA-1 unfolding 40-stage pipelining, is different from SHA-1 40-stage pipelining. Each of this func module consists of two operations of hash function. These two operations are processed in one cycle. Thus, the number of iterations of SHA-1 unfolding 40-stage pipelining reduces from 80 rounds to 40 rounds. Some modifications need to be done to the Message Unfolding module and Kt module. In the former module, two messages are operated simultaneously to reduce the message until 40 rounds. Therefore, the latter module, which is a constant of SHA-1 algorithm, also provides two constant data simultaneously. In this architecture, the sha_counter module only counts until 40 rounds.

3.6.3 RIPEMD-160 Design

RIPEMD-160 hash functions are extensively employed in a variety of cryptographic applications, including digital signatures, HMAC, and other data security applications. In this research, there are four proposed RIPEMD-160 designs: RIPEMD-160 iterative design, RIPEMD-160 pipelined design, RIPEMD-160 unfolding with factor 2, and RIPEMD-160 unfolding design with factor 4. An iterative design employs a single compression function, but a pipelined approach uses five distinct compression functions. The proposed pipelined

design is based on the modifications by Michail et al. (2008). Some modification has been done to produce the optimised RIPEMD-160. These methodologies were utilised to analyse the internal architecture of RIPEMD-160 concerning area, maximum frequency, and throughput of the design. The RIPEMD-160 hash function employs an unfolding transformation approach with factor 4, resulting in a high-throughput implementation. The throughput of the RIPEMD-160 unfolding architecture improved markedly. The aim of this research is to improve the throughput performance of RIPEMD-160. By using unfolding transformation factor 4, the throughput of RIPEMD-160 can be improved.

3.6.3.1 RIPEMD-160 Iterative Design

The iterative design is a basic design of hash function. Figure 3.44 shows the iterative design for a step function of RIPEMD-160. It consists of a non-linear function, shift rotation over ten positions, and left circular shift with a shift value. M and K represent the message and constant of RIPEMD-160, respectively. In this design, only one block of step function will be used during execution. All the parameters stated in the previous section will be used in this RIPEMD-160 design to obtain the correct output. The design's maximum frequency can be attained by utilising advisors regarding resources, timing, and power. Furthermore, positioning the register towards the end of the design might enhance the maximum frequency, as the register-to-register delay constitutes one of the most significant delays in modern circuit design.

There is only one non-linear function block for the iterative design that consists of five different non-linear functions. First, the message will be kept in the memory with input load, which means that if there is input load with logic “1”, the data will be transferred to the specific length of register. Then, the message will be kept in the memory starting from 0 to 15 locations. By using this method, it is easier to call the message based on message

selection ordering, as shown in the previous table. Counter can be used to call the message,m,m' and Shift, s, s' values. The iteration of this architecture will be processed until 80 rounds. Finally, the output $A, B, C, D,$ and E from the left and A', B', C', D' and $E'$from the right will be added with initial valuesH_0, H_1, H_2, H_3 and H_4 at the end of this design.

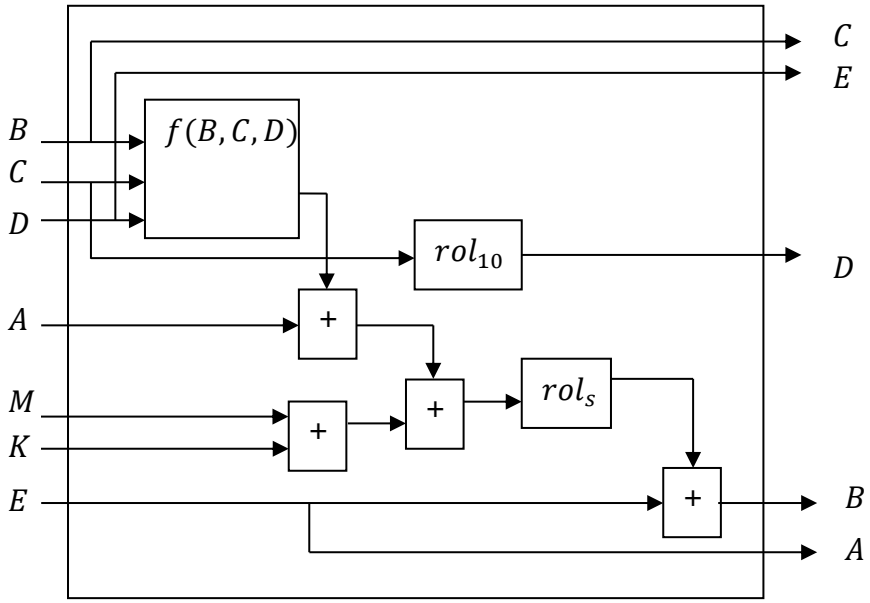


Figure 3.44 RIPEMD-160 Iterative Architecture Design

Figure 3.45 shows the top-level architecture of RIPEMD-160 design. From this figure, there are eight modules of RIPEMD-160 design. All the non-linear functions operate in this module. The output for the message, shift, and constant value comes from individual separate modules. As RIPEMD-160 executes in parallel form from left and right, there will be ten variables, which are five variables from the left (A, B, C, D, and E) and five variables from the right (A', B', C', D', and E').

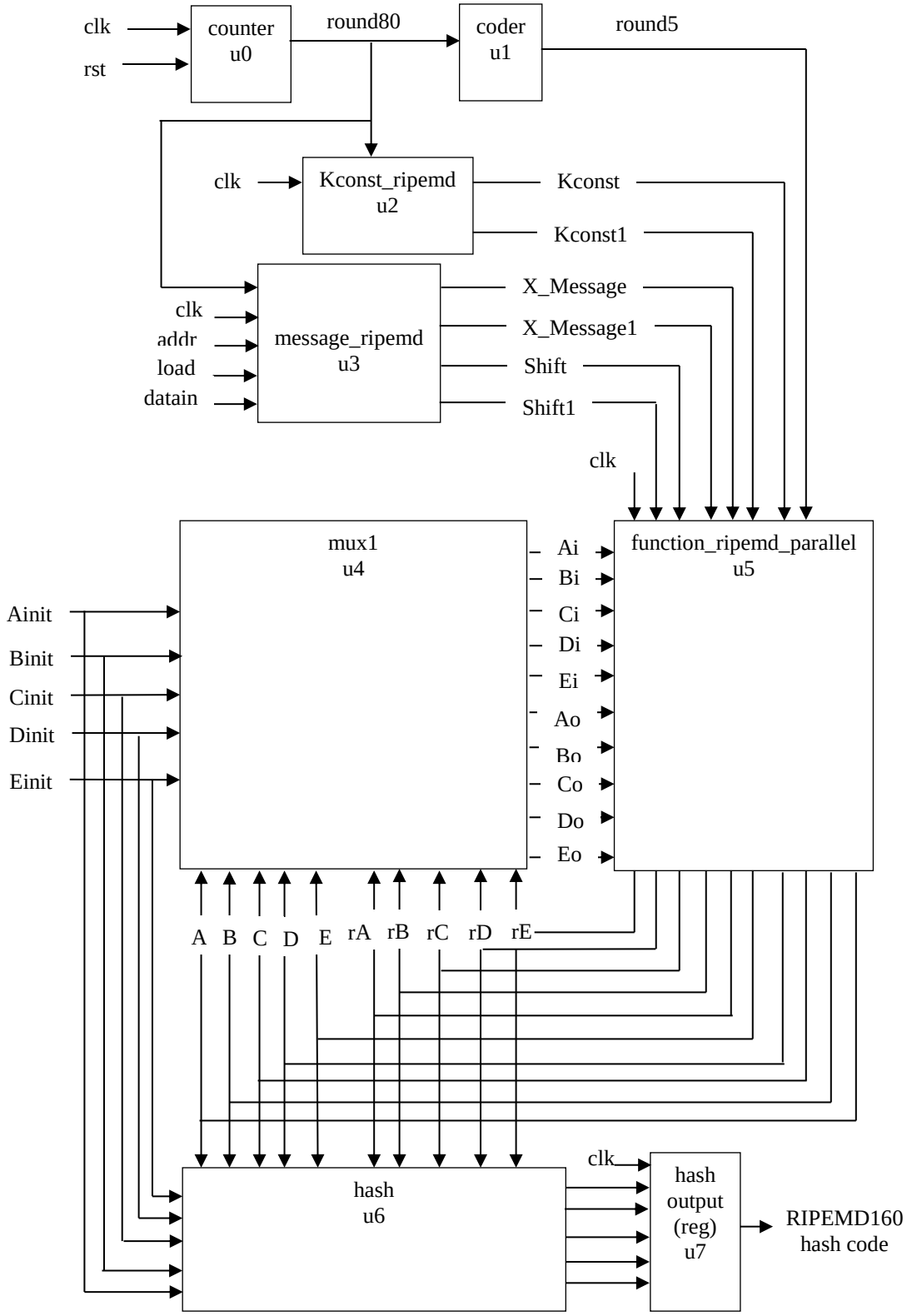


Figure 3.45 Top-Level Architecture of RIPEMD-160 Design

3.6.3.2 RIPEMD-160 Pipelined Design

Figure 3.46 shows the concept of a pipelined design. There are five stages of RIPEMD-160 design, starting from round 1 to round 5. Each round consists of 16 steps, and the overall rounds for this design are 80 rounds. By using the pipelined design in the RIPEMD-160 architecture, the performance of the design can be improved significantly even though there will be slight increment in terms of area implementation. Thus, the Quartus II advisor can help to reduce the logic elements used for this design. Generally, timing is vital to achieve the high maximum frequency of the design. Therefore, timing requirements for setup and hold time in the design need to be met by using the TimeQuest Timing Analyzer. In addition, it can increase the speed of the design by giving the appropriate clock constraint to the design.

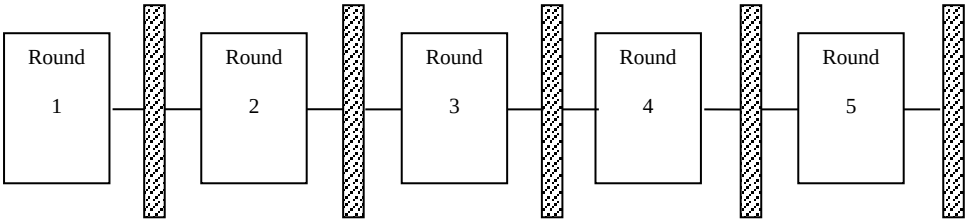


Figure 3.46: Pipelined Design

The implementation of pipelined design, as seen in Figure 3.46, enhances the performance of the RIPEMD-160 architecture. Besides, an idea of modification by Michail et al. (2008) has been considered. Figure 3.46 illustrates the pipelined architecture design for a single block of RIPEMD-160. The difference between the design by Michail et al. (2008) and this design is that the message, M_3 and constant, K_3 will be added with input A directly. Meanwhile, in Michail et al. (2008), message, M_{t+2} and constant, K_{t+2} will be added with D_{t-1}.

By using the same sequence as normal RIPEMD-160 algorithm for input A, M_3 and K_3, the design can be improved significantly in comparison to the previous design. From Figure 3.47, there are two initialisation values for variables h and W. First, M_1 and K_1will be added with A value to give the output W_1.Then, the output Z will shift to the left with 4-bit amount and will be added with input E to obtain the output B, B'. This concept is the same as the original idea, which is based on algorithm 1 of RIPEMD-160 algorithm. For the second cycle, let us consider the input M_2 and K_2. These two inputs will be added to input *E* to give output h_1. Then, the output will proceed to Win and will be added to non-linear *f*(B, C,D). In this case, the input for A_{t+1} is equal to E_t. Subsequently, the output will be directly added to non-linear *f*(B, C,D). The sequence to obtain output B,B' is similar to the previous method but with a different input. By using algorithm 1 of RIPEMD-160 algorithm, the output A, C, D and E from the left and $A', C', D'$and E' from the right can be obtained. Table 3.16 shows the proposed pipelined design of RIPEMD-160.

Table 3.16 RIPEMD-160 Input for Pipelined Design

Pre_Computation	**Final_Calculation**
$A_t = A_t$	$A_{t+1} = E_t$
$B_t = B_t$	$B_{t+1} = E_t + Rol_s(Z_t)$
$C_t = C_t$	$C_{t+1} = B_t$
$D_t = D_t$	$D_{t+1} = Rol_{10}(C_t)$
$E_t = E_t$	$E_{t+1} = D_t$
$Z_t = f_t$(B,C,D) + W_t	$W_{t+1} = h_t$
$h_t = h_t$	$h_{t+1} = M_t + K_t + A_t$

Some modifications, as shown in Figure 3.47, have been done to increase the performance of the design. In this design, five pre-computation blocks are used because of the five different types of non-linear functions, while for Final_Calculation, only one block unit will be used. The calculation for M_3 and K_3 starts at this stage. The output of hin will

be used for the rest of the calculation of RIPEMD-160 hash function. This output h will go to output Win. This output Win will be selected as W input for each Pre_Computation. The process executes in pipelined design, and selection needs to be done when it comes to different rounding of RIPEMD-160.

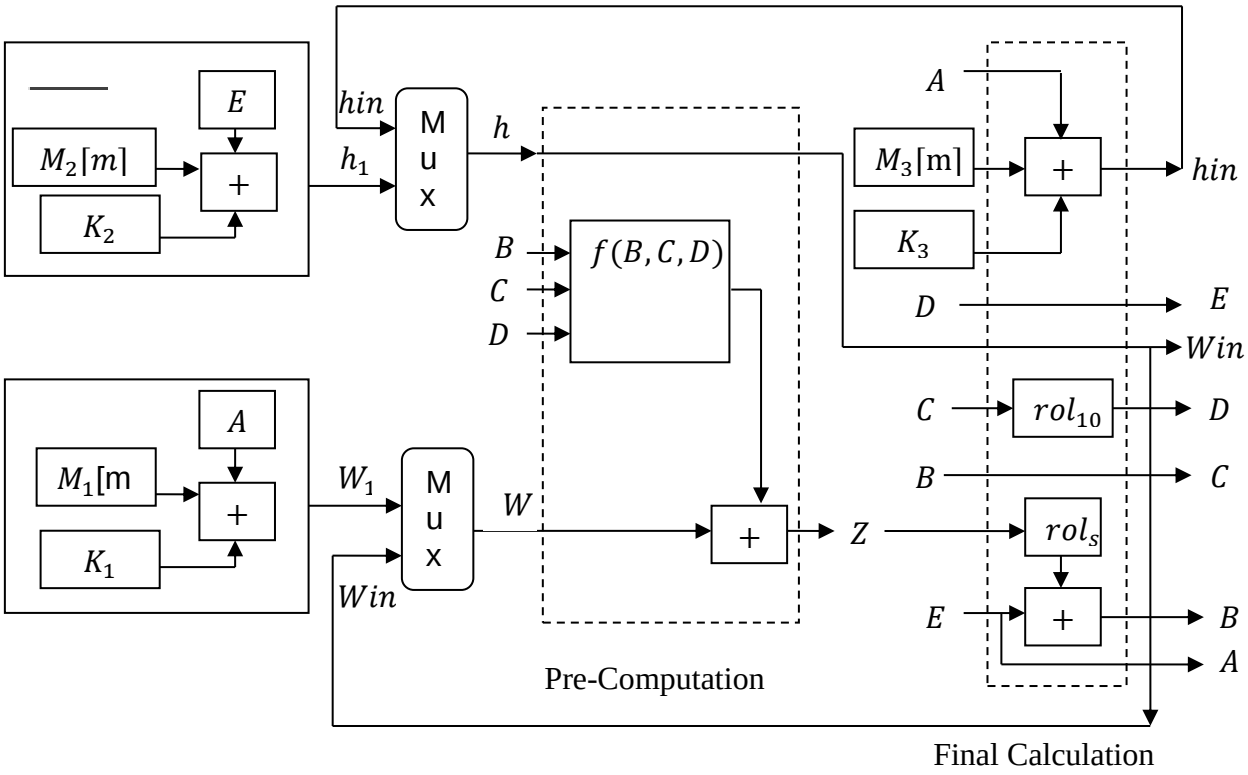


Figure 3.47 RIPEMD-160 Pipelined Architecture Design for a Single Block

Pipelined design can contribute to fast design, but it suffers disadvantages with large area implementation. Utilising the Quartus II advisors can optimise area implementation, enhance timing, and decrease power consumption. The HDL coding style needs to be considered as well in designing the RIPEMD-160 design because this design will be implemented in the gate level implementation based on the FPGA family device. The design can be improved by using a proper and efficient coding style. Moreover, by using an appropriate clock constraint in the design, the setup and hold time requirements will be met.

In this design, synopsys design constraint (SDC) input clock 4 ns is used in the design to obtain a high maximum frequency of the design. Besides, by using the similar method in the proposed iterative design, the speed of the design can be enhanced by putting the register at the end of the output port. This delay contributes to register-to-register delay. In this design, the performance of the RIPEMD-160 pipelined design increases significantly in comparison to the iterative design. Figure 3.48 shows the top-level architecture of the pipelined RIPEMD-160 design. From this figure, there are 23 modules in the pipelined RIPEMD-160 design. Each pre-computation module consists of non-linear functions for different rounds. The final output of RIPEMD-160 will be calculated at the Final_Calculation module. Two initial conditions originate from XK_1 and XK_2 modules to produce the h and W values. In this case, there will be two outputs: the data from the left and the data from the right.

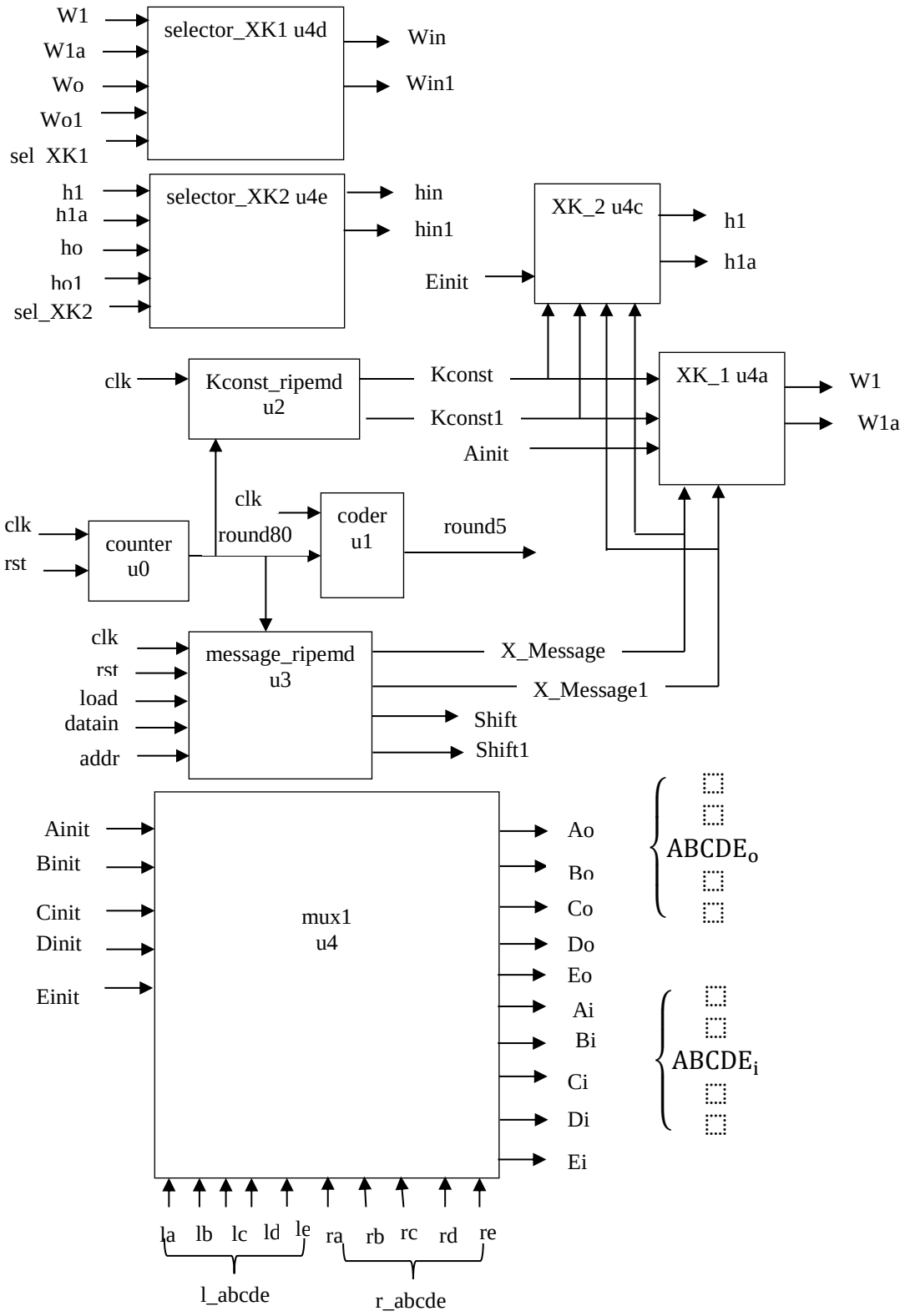


Figure 3.48 Top-Level Architecture of Pipelined RIPEMD160 Design

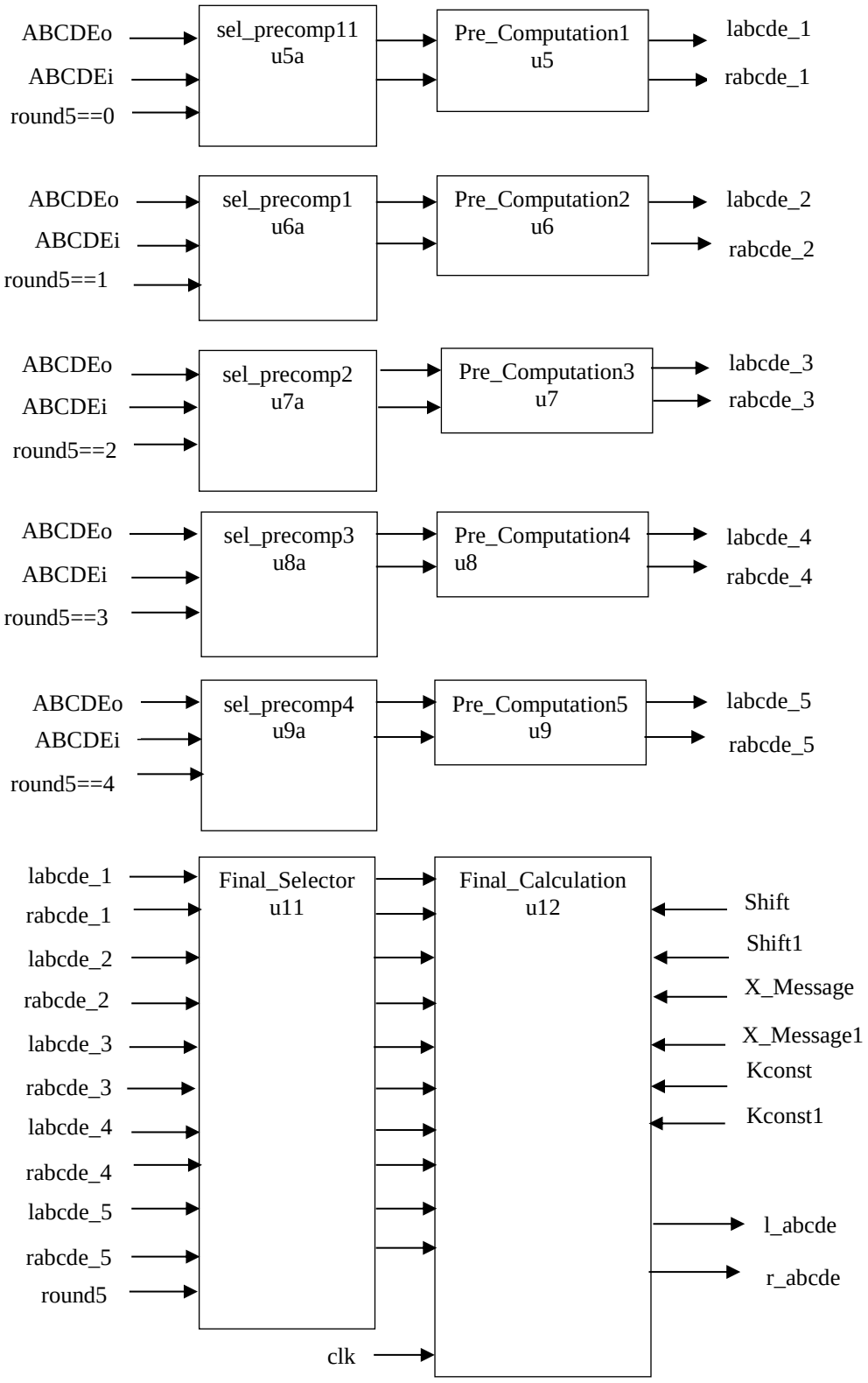


Figure 3.48 Continued

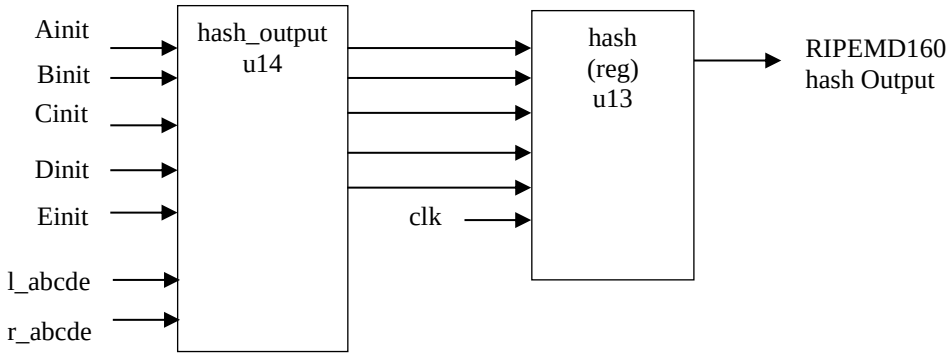


Figure 3.48 Continued

3.6.3.3 RIPEMD-160 Unfolding Factor 2 Design

Figure 3.49 shows the top level of RIPEMD-160 unfolding design with factor 2. This architecture is almost similar to the iterative RIPEMD-160 design with a slightly different, where this architecture has the Func module. Some of these modules need to be modified by applying the unfolding transformation technique to the design. The modules are Coder, KConst, Message, and Function Parallel modules. The rounding numbers of this design need to be reduced by 2. Thus, each round only consists of eight rounds instead of 16 rounds in the iterative design. Overall, the number of looping is 40 rounds.

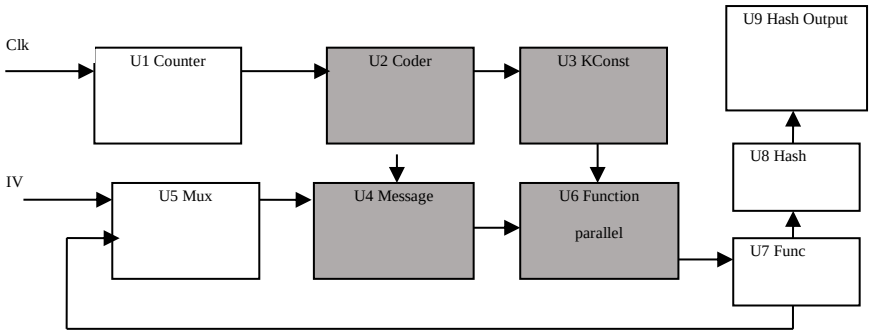


Figure 3.49 Proposed RIPEMD-160 Unfolding Factor 2 Design

This architecture accounts for both the left and right sides of the design, necessitating parallel data translation for all modifications. To generate this design in unfolding transformation factor 2, the Coder module is designed based on the Counter module. Figure

3.50 illustrates the block diagram of the Coder module for RIPEMD-160 unfolding factor 2. The number of steps in each cycle must be reduced from 16 to 8. There are five rounds in module: Round0, Round1, Round2, Round3, and Round4. Each round consists of eight steps. Overall, there are 40 rounds, which are divided into five parts in the Coder module.

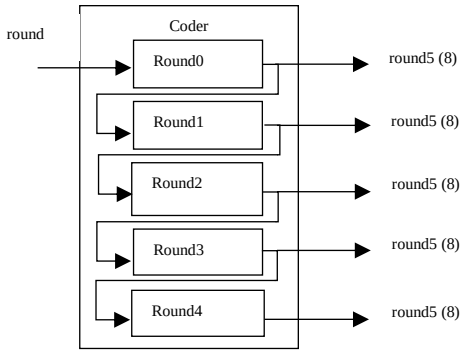


Figure 3.50 Coder Module of RIPEMD-160 Unfolding Factor 2

Then, the KConst module is a constant module for both left and right paths of RIPEMD-160 design. It operates in parallel mode of two data from both parts. In other words, four data are generated for each cycle: two from the left path and two from the right path. Figure 3.51 illustrates the block diagram of KConst for RIPEMD-160 with unfolding factor 2. In each module consists of KConsta and KConstb from the left line and KConst1a and KConst1b from the right line. Therefore, it is divided into five parts because the RIPEMD-160 hash function contains five non-linear functions. Each non-linear function will operate eight steps for each round.

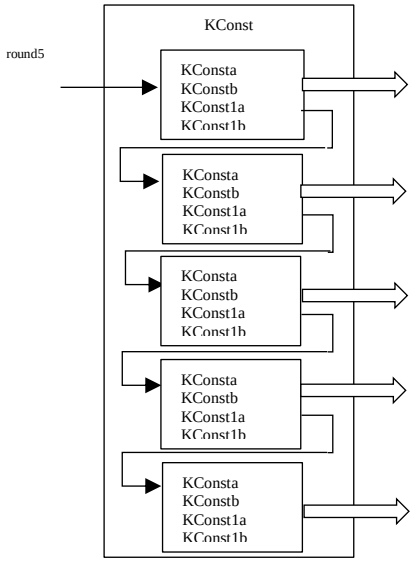


Figure 3.51 KConst Module of RIPEMD-160 Unfolding Factor 2

Similarly, the Message module also generates the output for both left and right paths of this design. It consists of message and shift values, as shown in the previous table. First, all the data input of RIPEMD-160 will be divided into 16 parts. Each part comprises 32-bit data. These data are stored in the 16-memory location. The counter will generate the input to execute the correct sequence of the data. Figure 3.52 shows the block diagram of the Message module of RIPEMD-160 unfolding factor 2. In this Message module design, message and shift values are combined into one 8-bit block, where the first 4 bits is for message input and the shift value will be for the second 4 bits. Overall, eight data will be generated from this module: four data are for the message value (left and right) and four data are for the shift value (left and right).

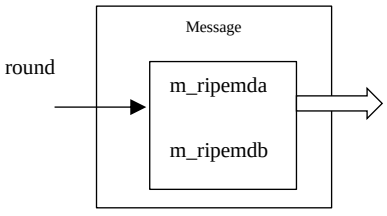


Figure 3.52 Message Module of RIPEMD-160 Unfolding Factor 2

The last module that needs to be modified in this design is the Function Parallel module. Figure 3.53 shows the Function Parallel module of RIPEMD-160 unfolding factor 2. Two non-linear functions will be executed in parallel mode. The first non-linear function consists of input B, C, and D for both left and right. Let say the output for the first non-linear function is func_rmda and func_rmd1a. Both outputs represent data from both left and right data. Then, simultaneously, the second non-linear function needs to generate the output from three different inputs.

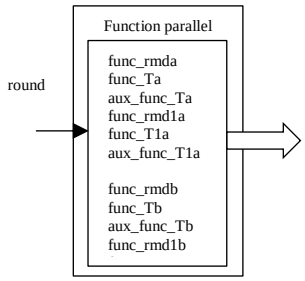


Figure 3.53 Function Parallel of RIPEMD-160 Unfolding Factor 2

The following Algorithm 2 shows the derivation input for the second non-linear function. Messagea, KConsta, and Shifta are the message, constant, and shift values for the left path. Message1a, KConst1a, and Shift1a represent the data for the right path. The output for both second non-linear functions are func_rmdb and func_rmd1b. Three different inputs for this function are obtained from Algorithm 2. The algorithm illustrates the compression function of RIPEMD-160 algorithm for both parts.

There are two non-linear functions that will operate in one cycle. From Algorithm 2, func_rmda and func_rmd1a refer to the first non-linear function for both left and right paths. Then, func_rmdb and func_rmd1b represent the second non-linear function for both left and right paths. In RIPEMD-160 unfolding factor 2, two non-linear functions will operate at the same time, which are func_rmda and func_rmdb with three different inputs. In conventional compression function, there is only one non-linear function. The structure of compression

function of a conventional design is not really complicated compared with unfolding factor 2. The input for the second non-linear function, func_rmdb must be correct to obtain the result. The final output of this function generates output in the Func module. The output will be different from the compression function, as shown in Figure 3.10 because two parallel data are executed at the same time in one cycle to fulfil the unfolding transformation technique. Finally, the output of hash code is obtained after 40 rounds of looping.

Algorithm 2: Compression Function of RIPEMD-160 Unfolding Factor 2

```
func_rmda(B,C,D)
func_Ta = A + func_rmda + Messagea + KConsta
rot1_la = func_Ta << shifta
rot2_la = func_Ta  >> (32 – shifta)
func__rmd1a(B1,C1,D1)
func_T1a = A + func_rmd1a + Message1a + KConst1a
rot1_ra = func_T1a << shifta
rot2_ra = func_T1a  >> (32 – shifta)
aux_func_Ta = (rot1_la | rot2_la) + E
aux_func_T1a = (rot1_ra | rot2_ra) + E1
func_rmdb (aux_func_Ta, B, {C[21:0],C[31:22]})
func_rmd1b (aux_func_T1a,B1,{C1[21:0],C1[31:22]})
```

3.6.3.4 RIPEMD-160 Unfolding Factor 4 Design

Modification on the top level of RIPEMD-160 design needs to be designed to apply the unfolding transformation with factor 4. In this method, the number of cycles in the RIPEMD-160 architecture is reduced from 40 cycles to 20 cycles. Reducing the design's number of cycles will decrease the latencies. Thus, this leads to high throughput of the RIPEMD-160 design. Similar to the previous section, which is unfolding with factor 2, the

modification needs to be designed for some modules, such as Coder, KConst, Message, and Function Parallel. Figure 3.54 shows the Coder module of RIPEMD-160 unfolding factor 4. In this design, the coder module consists of five rounds, where each round comprises four steps.

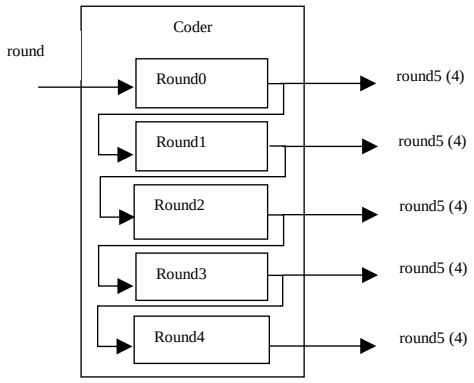


Figure 3.54 Coder Module of RIPEMD-160 Unfolding Factor 4

Figure 3.55 shows the KConst module of RIPEMD-160 unfolding factor 4. The KConst module provides a constant value for both left and right paths. In each round, there will be eight constants executed in parallel form. The Message module needs a slight modification, where the round is reduced into 20 rounds, and the data for both message and shift value ares compressed into eight data of 8-bit input. From this figure, there are four data from the left and four data from the right (KConst1a, KConst1b, KConst1c, and KConst1d).

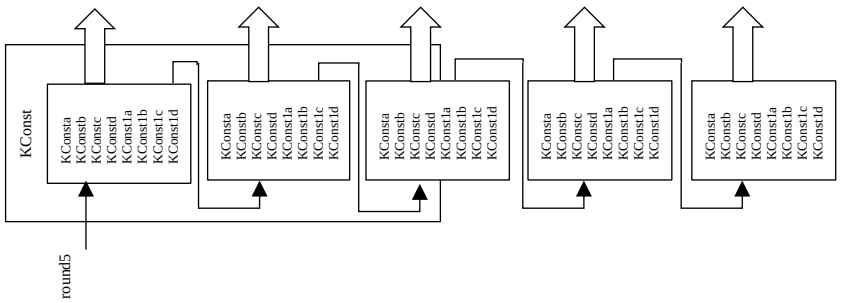


Figure 3.55 KConst Module of RIPEMD-160 Unfolding Factor 4

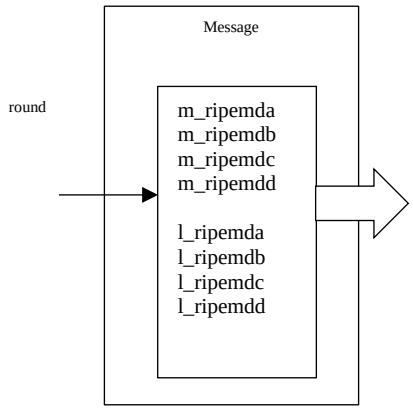


Figure 3.56 Message Module of RIPEMD-160 Unfolding Factor Four

Figure 3.56 shows the Message module of RIPEMD-160 unfolding factor 4. The Message module consists of message and shift values. There are eight outputs of messages: four from the left path (m_ripemda, m_ripemdb, m_ripemdc, and m_ripemdd) and four from the right path (l_ripemda, l_ripemdb, l_ripemdc, and l_ripemdd). This module will operate the operation in 20 rounds.

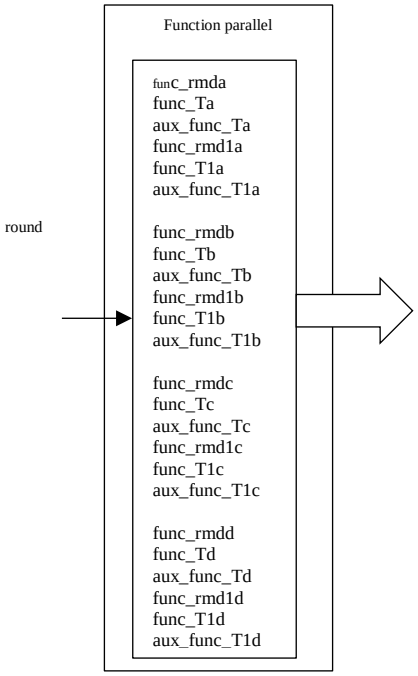


Figure 3.57 Function Parallel Module of RIPEMD-160 Unfolding Factor 4

Figure 3.57 shows the Function Parallel module of RIPEMD-160 unfolding factor 4. There are non-linear functions in this module, which are based on the RIPEMD-160 algorithm. Therefore, there will be four parallel non-linear functions generated simultaneously to produce the output in one cycle. Three different inputs of non-linear function need to be identified. The output func_rmda, func_rmd1a, func_rmdb, and func_rmd1b represent the first and second non-linear functions for both left and right, respectively. Meanwhile, for the third and fourth non-linear functions in this module, they are represented by func_rmdc, func_rmd1c, func_rmdd, and func_rmd1d, respectively.

Algorithm 3 shows the three different inputs for both left and right paths of four non-linear functions in this Unfolding factor 4 design. The new idea is proposed in this study, especially the unfolding transformation with factor 4, where all non-linear functions need to be executed in four parallel transformations.

Algorithm 3: Compression Function of RIPEMD-160 Unfolding Factor 4

```
func_rmda(B,C,D)
func__rmd1a(B1,C1,D1)
func_rmdb (aux_func_Ta, B, {C[21:0],C[31:22]})
func_rmd1b (aux_func_T1a,B1,{C1[21:0],C1[31:22]})
func_rmdc (aux_func_Tb, aux_func_Ta, {B[21:0],B[31:22]})
func_rmd1c (aux_func_T1b,aux_funcT1a,B1[21:0],B1[31:22])
func_rmdd(aux_func_Tc,aux_func_Tb,{aux_func_Ta[21:0], aux_func_Ta[31:22])
func_rmd1d(aux_func_T1c,aux_func_T1b,{aux_func_T1a[21:0], aux_func_T1a[31:22])
```

The compression function of RIPEMD-160 unfolding factor 4 is almost similar to RIPEMD-160 unfolding factor 2. The other two non-linear functions are added from both left and right paths: func_rmdc, func_rmd1c, func_rmdd, and func_rmd1d. The three different inputs for each non-linear function for unfolding factor 4 are shown in Algorithm

2, and they must be evaluated correctly. This is the most important part in designing an unfolding design. The five outputs of RIPEMD-160 hash function depend on the correct result because it will be looping until the last rounds to obtain the hash value.

After obtaining the results from four non-linear functions, the outputs for A, B, C, D, and E for both left and right paths of this design are computed. The following equation shows the final output from the left, which are l_A, l_B, l_C, l_D, and l_E. Meanwhile, the output from the right path are r_A, r_B, r_C, r_D, and r_E. These outputs are looping until 20 rounds due to the implementation of unfolding transformation with factor 4. Finally, all these outputs are combined using the formula to obtain the hash codes. All the outputs need to be converted to the little-endian mode because the results for RIPEMD-160 are in the little-endian format. The unfolding technique will reduce the number of latencies of the RIPEMD-160 design. High throughput will be obtained if the unfolding factor increases. The number of unfolding factors will determine the reduction in the number of cycles of RIPEMD-160 unfolding. The number of cycles decreases as the number of unfolding factors increases. This leads to high throughput of the RIPEMD-160 unfolding transformation technique.

3.6.4 SHA-256 Design

In recent years, security has become an increasingly significant subject of study. The performance of these information-protecting methods has been enhanced by the development of numerous cryptographic algorithms. One of the cryptography categories is a hash function. The SHA-256 hash function is proposed for implementation in this study. In order to optimise the throughput of the SHA-256 design, this investigation implemented the unfolding transformation methodology. The hash function utilises the unfolding method to generate the hash value output by modifying the SHA-256 structure. The delay in this

unfolding method enables SHA-256 to reduce the number of clock cycles required for traditional architecture by a factor of two, from 64 to 34. To put it another way, one cycle of the SHA-256 design can generate up to four parallel inputs for the output.

3.6.4.1 SHA-256 Iterative Design

SHA-256 has been implemented using Verilog code and comprises six modules within its top-level architecture: counter SHA-256, message schedule, constant SHA-256, multiplexer, compression function, and output SHA-256. Figure 3.58 depicts the proposed architecture of the SHA-256 hash function. The input data consists of 15 blocks of 32-bit data, which is padded by adding a single 1 bit at the end of the message, followed by n 0 bit, with the final 64 bits representing the length of the message. The SHA-256 hash function produces a 512-bit output. The input message W_t for $16 \leq t \leq 63$ can be derived using Equation 3.8. The message sequence is produced with a counter module. The SHA-256 hash function employs 64 rounds of iteration of the compression function to derive the final hash code. Prior to the commencement of SHA-256 message processing, eight buffer initialisations are produced with a multiplexer module. The ROM blocks are utilised to define the constant, K_t, comprising 64 32-bit ROM blocks. Finally, the output module generates the SHA-256 message digest, incorporating buffer initialisations with the final output of the SHA-256 compression function.

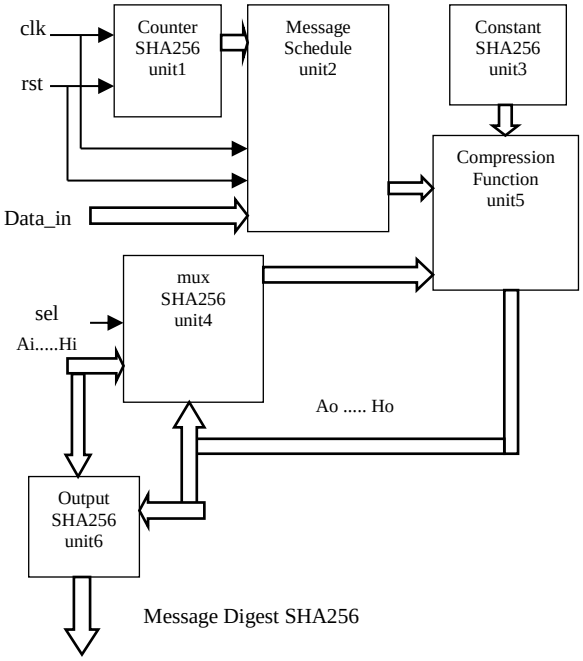


Figure 3.58 Top Level of SHA-256 Design

3.6.4.2 SHA-256 Unfolding Factor 2 Design

The message schedule and compression function of the SHA-256 algorithm require modification to achieve the unfolding architecture. An unfolding design is a method that decreases latency according to the number of J factors (Parhi, 1999). Furthermore, this technique can enhance the throughput of the SHA-256 algorithm. In this study, the unfolding technique with factor 2 has been employed. Each block in the message schedule and compression function requires modifications. Figures 3.59 and 3.60 illustrate the block diagrams of $Temp_{1o}$ and $Temp_{2o}$. The subsequent block diagrams of $Temp_{1o}$ and $Temp_{2o}$ represent the modifications of Equations 3.15 and 3.16.

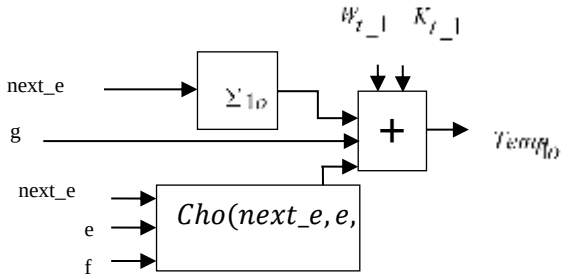


Figure 3.59 $Temp_{1o}$ Block Diagram Architecture

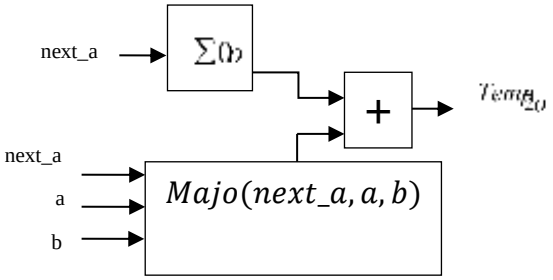


Figure 3.60 $Temp_{2o}$ Block Diagram Architecture

These equations are incorporated into the compression function of the SHA-256 algorithm. $Temp_{1o}$ comprises $\sum 1o$, $Cho(next\_e,e,f)$, Message, w_{t_1}, and Constant, K_{t_1}, while $Temp_{2o}$ encompasses $\sum 0\, o$ and $Majo(next\_a,a,b)$. A 32-bit adder is employed to obtain these outcomes. All data inputs are different for each of the blocks in $Temp_{1o}$ and $Temp_{2o}$ block diagram architectures. Figures 3.61 and 3.62 illustrate the two architectures for Cho(next_e,e,f) and Majo(next_a,a,b). Both architectures comprise AND, NOT, and XOR gates with varying implementation structures. The data inputs next_e and next_a can be derived from the compression function of the SHA-256 algorithm, as shown in the compression function of SHA-256.

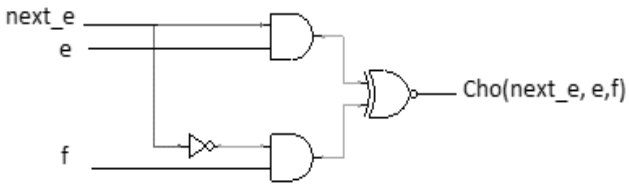


Figure 3.61 Cho (next_e,e,f) Function Architecture

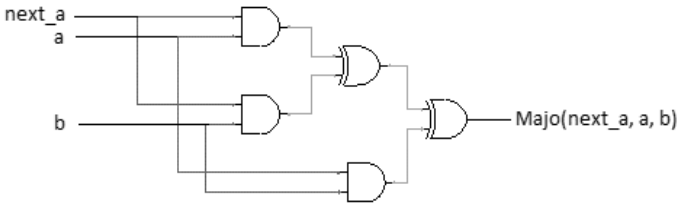


Figure 3.62 Majo(next_a,a,b) Function Architecture

The proposed architectures for $\sum 0\,o$ and $\sum 1o$ are shown in Figures 3.63 and 3.64, respectively. The input next_a is for $\sum 0\,o$, while the input next_e is for $\sum 1\,o$. All rotations for both architectures proceed in the correct direction with a fixed number of values. Finally, an XOR gate is employed to combine all the inputs to obtain the final outputs of $\sum 0\,o$ and $\sum 1o$.

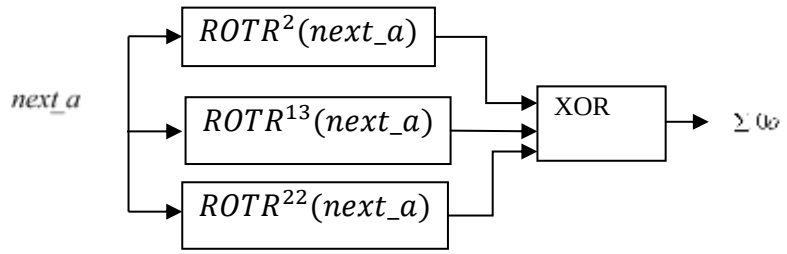


Figure 3.63 $\sum_{0o}(next\_a)$ Architecture

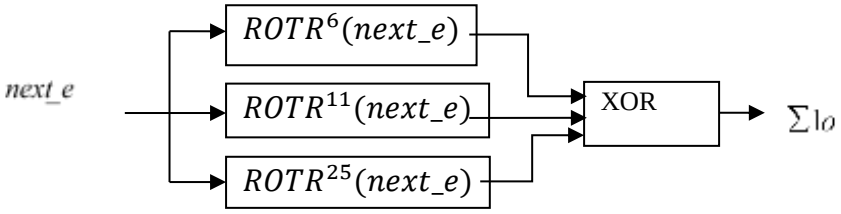


Figure 3.64 $\sum_{1O}(next\_e)$ Architecture

The message schedule is modified from prior results similar to that of the compression algorithm. The modification of the previous equation for sigma0 and sigma1 has been performed after receiving the signal. The start of this sequence is at wt2 and concludes at wt15. Figures 3.65 and 3.66 illustrate the architectures for σ_{0o} and σ_{1o} functions, respectively. These architectures are vital for generating the SHA-256 message schedule. A constant quantity of value is used to rotate w_2 in the appropriate direction for σ_{0o}, whereas for σ_{1o}, the new data input is used. Subsequently, w_2 is right shifted with a specific value in the σ_{0o} function, and similarly with the W_{15}. It is right shifted in the σ_{1o} function with a certain value.

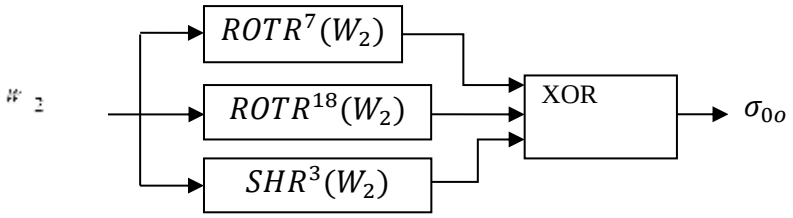


Figure 3.65 σ_{0o} Architecture

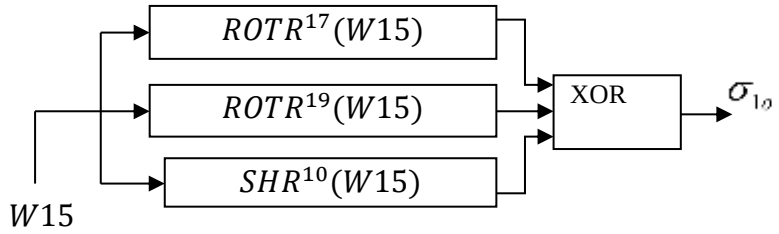


Figure 3.66 σ_{1o} Architecture

The message schedule of SHA-256 unfolding design must be modified as follows:

$W_t =$ message input $\qquad 0 \le t \le 15$

$$W_t = \sigma_{1o}^{256}(W_t) + W_{t-6} + \sigma_{0o}^{256}(W_{t-14}) + W_{t-15} \qquad 16 \le t \le 63 \qquad \text{Equation 3.25}$$

The following equations yield the output of SHA-256 unfolding hash values.

$$next\_e = D + Temp_1 \qquad \text{Equation 3.26}$$

$$next\_a = Temp_1 + Temp_2 \quad \text{Equation 3.27}$$

$$next\_eo = C + Temp_{1o} \quad \text{Equation 3.28}$$

$$next_{ao} = Temp_{1o} + Temp_{2o} \quad \text{Equation 3.29}$$

$$\begin{aligned} h &= f \\ g &= e \\ f &= next\_e \\ e &= next\_eo \\ d &= b \\ c &= a \\ b &= next\_a \\ a &= next\_ao \end{aligned} \quad \text{Equation 3.30}$$

Figures 3.67 and 3.68 illustrate the revised inner architecture for Temp11 and Temp21, respectively. These two inputs of figures are presented with distinct signals. This is due to the factor 4 of the unfolding technique. Similarly, function $\sum 11$ and $\sum 01$ also use a new input to be applied in the new architecture of $Temp_1$ and $Temp_2$. The new equations can be derived, as shown in Equations 3.31 and 3.32, by applying the new input signals to both equations.

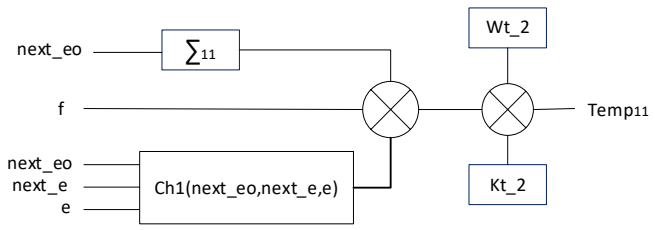


Figure 3.67 Architecture of $Temp_{11}$ Block Diagram

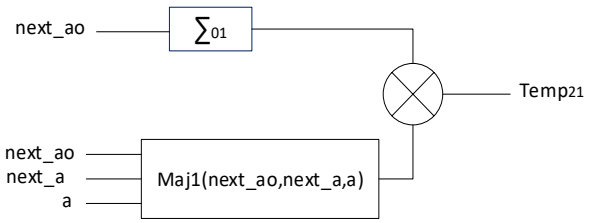


Figure 3.68 Architecture of $Temp_{21}$Block Diagram

next_e1 = b + $Temp_{11}$ Equation 3.31

next_a1 = $Temp_{11}$ + $Temp_{21}$ Equation 3.32

3.6.4.3 SHA-256 Unfolding Factor 4 and Low Power Design

This section outlines the development and execution of unfolding factor 4 for the SHA-256 hash function. The alteration starts at the top level of the SHA-256 architecture design, with a specific focus on two fundamental components: the message schedule and the compression function module. These components need to be restructured to enable the parallel execution of four rounds within each clock cycle, which effectively enhances throughput and reduces processing latency.

The message schedule that consists of σ_0 and σ_1 functions need to be rebuilt with updated input values that correspond to four-round advances in order to account for the unfolding factor 4. The updated equations for and σ_0 and σ1 are shown in Equation 3.33, 3.34. 3.3 and 3.36 which represents σ_{01}, σ_{02}, σ_{11}, and σ_{12} . These changes guarantee that the message words are appropriately pre-processed and positioned for processing in parallel over several rounds.

$$\sigma_{01}^{256}(W_3) = ROTR^{7}(W_3) \oplus ROTR^{18}(W_3) \oplus SHR^{3}(W_3)$$ Equation 3.33

$$\sigma_{02}^{256}(W_3) = ROTR^{7}(W_4) \oplus ROTR^{18}(W_4) \oplus SHR^{3}(W_4)$$ Equation 3.34

$$\sigma_{11}^{256}(next_{wt}) = ROTR^{17}(next\_wt) \oplus ROTR^{19}(next\_wt) \oplus SHR^{10}(next\_wt)$$ Equation 3.35

$$\sigma_{12}^{256}(next_{wt}) = ROTR^{17}(next\_wto) \oplus ROTR^{19}(next\_wto) \oplus SHR^{10}(next\_wto)$$ Equation 3.36

The unfolding of the compression function begins with the derivation of the Ch and Maj logical functions, which are used to calculate the Temp1 and Temp2 intermediate values. The Ch function and the Σ_1 operation, which are included in the Temp1 block. Equation 3.37 and Equation 3.38 shows the equation for Ch1 and Ch2 for SHA-256 Unfolding factor 4 design. The mathematical expressions for Σ_{11} and Σ_{12} for unfolding factor 4 are shown in Equation 3.39 and Equation 3.40.

$$Ch1\ (next\_Eo, next\_E, E)=(next\_Eo\ \wedge next\_E\ )\bigoplus(\neg\ next\_Eo\wedge E)$$ Equation 3.37

$$Ch2\ (next\_E1, next\_Eo, next\_E\ )=(next\_E1\ \wedge next\_Eo\ )\bigoplus(\neg\ next\_E1\wedge next\_E\ )$$ Equation 3.38

$$\sum_{11}(next_{Eo}) = ROTR^{6}(next_{Eo}) \oplus ROTR^{11}(next_{Eo}) \oplus ROTR^{25}(next_{Eo})$$ Equation 3.39

$$\sum_{12}(next_{E1}) = ROTR^{6}(next_{E1}) \oplus ROTR^{11}(next_{E1}) \oplus ROTR^{25}(next_{E1})$$ Equation 3.40

This is followed by the construction of the Temp2 block using the output of the Maj function and the Σ_0 transformation. The Maj1 and maj2 function for unfolding factor 4 is illustrated in Equations 3.41 and 3.42, while Equations 3.43 and 3.44 represent Σ_{01} and Σ_{02}. Consistent data propagation across the architecture is guaranteed by recalculating Σ_0 and Maj in accordance with the unfolding schedule. Once the formulations of Temp1 and Temp2 have been determined for the four-round unfolding scheme, they are employed to iteratively update the internal state variables of the hash engine. This leads to the optimisation of the

high-throughput design by allowing for the effective computation of the intermediate and final hash values in a reduced number of cycles.

$$Maj1(next_{Ao},\ next_A,\ A) = (next_{Ao} \wedge next_A) \oplus (next_A \wedge A) \oplus (next_{Ao} \wedge A) \quad \text{Equation 3.41}$$

$$Maj2(next_{A1} \wedge next_{Ao}) \oplus (next_{Ao} \wedge next_A) \oplus (next_{A1} \wedge next_A) \quad \text{Equation 3.42}$$

$$\sum_{01}(next_{Ao}) = ROTR^2(next_{Ao}) \oplus ROTR^{13}(next_{Ao}) \oplus ROTR^{22}(next_{Ao}) \quad \text{Equation 3.43}$$

$$\sum_{02}(next_{A1}) = ROTR^2(next_{A1}) \oplus ROTR^{13}(next_{A1}) \oplus ROTR^{22}(next_{A1}) \quad \text{Equation 3.44}$$

The SHA-256 unfolding factor 4 design is calculated until $Temp_{12}$ and $Temp_{22}$ since it requires four parallel executions.

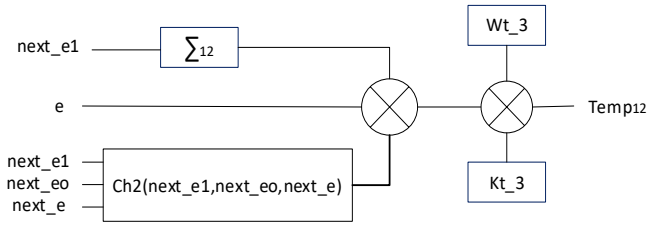


Figure 3.69 Architecture of $Temp_{12}$ Block Diagram

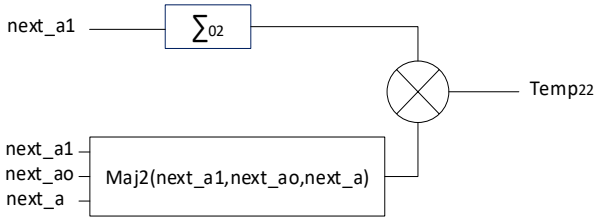


Figure 3.70 Architecture of $Temp_{22}$ Block Diagram

Figures 3.69 and 3.70 illustrate the new outputs for both Temp functions. In both function $\sum 12$ and $\sum 02$, two new signal inputs are employed. In addition, the data are relocated to the same place in the input sequence as the one previously. Both figures demonstrate these data inputs. The new signals for both Equations 3.45 and 3.46 are derived from the $Temp_{12}$ and $Temp_{22}$ datasets. The output of next_e2 and next_a2 are shown in the following equations.

next_e2 = a + $Temp_{12}$ Equation 3.45

next_a2 = $Temp_{12}$ + $Temp_{22}$ Equation 3.46

The message schedule is modified from prior results similar to that of the compression algorithm. The modification of the previous equation for sigma0 and sigma1 has been performed after receiving the signal. The start of this sequence is at wt2 and concludes at wt15. Figures 3.71 and 3.72 illustrate the architectures for σ_{0o} and σ_{1o} functions, respectively. These architectures are vital for generating the SHA-256 message schedule. A constant quantity of value is used to rotate w_2 in the appropriate direction for σ_{0o}, whereas for σ_{1o}, the new data input is used. Subsequently, w_2 is right shifted with a specific value in the σ_{0o} function, and similarly with the W_{15}. It is right shifted in the σ_{1o} function with a certain value.

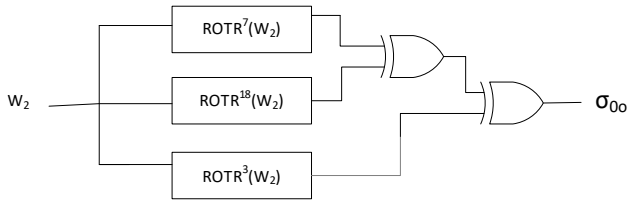


Figure 3.71 Architecture of σ_{0o}

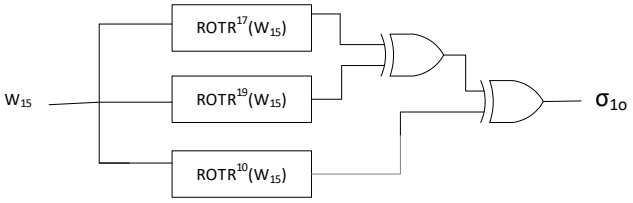


Figure 3.72 Architecture of σ_{1o}

With input W_3 and next_wt, σ_{01} and σ_{11} are computed for unfolding factor 4. The following Equation 3.47 is used to calculate next_wt1.

next_wt1 = W_2 + σ_{01} + W_{11} + σ_{11} Equation 3.47

Finally, input W_4 and next_wto are used to generate the input for σ_{02} and σ_{12} . Figure 3.73 depicts the structure of next_ wt2 and the overall design of unfolding factor 4 for the message schedule. The message input of Wo data begins with W_4 and ends with W_{15}. The output sequence of next_wt used in unfolding factor 4 uses the similar method used in unfolding factor 2.

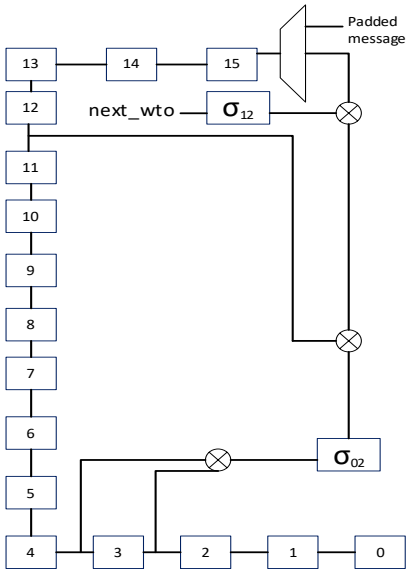


Figure 3.73 Message Schedule of SHA-256 Unfolding Factor 4 Design

The proposed concepts in this SHA-256 design concentrate on low power design implementation. Switching activity is a method to reduce dynamic power. State encoding was introduced in this research as an activity technique for low power devices to reduce the power consumption of SHA-256. For state encoding, the SHA256 unfolding transformation designs were divided into two unique designs: SHA-256 unfolding with Binary encoding and SHA-256 unfolding with Gray encoding. State of encoding transitions between binary and Gray encoding are shown in Table 3.17. The maximum number of possible transitions in a single clock cycle is shown to be three for Binary encoding and one for grey encoding in the following table. This is because each transition between states in the encoding only

affects one bit. Using this strategy in the creation of Finite State Machines can reduce power usage (FSM).

Table 3.17 Comparison of Binary and Gray Encoding

State Encoding	Binary Encoding	Gray Encoding
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

3.6.5 HMAC Design

Message authentication is the procedure of validating the authenticity of messages, where two important factors must be considered in the verification of the authenticity of a message, such as a source, and that the message has not been altered. In conventional encryption authentication, both the sender and receiver possess an identical key during the transmission process (Stallings, 1996). In other words, only the genuine user has the key to decrypt the message. Figure 3.74 illustrates the block design for the comprehensive message authentication procedure, which incorporates a secret key utilised for data generation within the algorithm. The sender receives this message and its MAC; once the MAC has been compared to the MAC at the receiver, the receiver must determine whether it has received the same MAC using the same secret key. The message has not been changed and is deemed authentic if the output MAC matches the one transmitted.

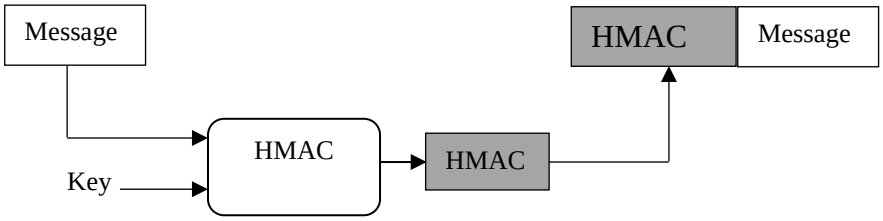


Figure 3.74 Message Authentication using HMAC

3.6.5.1 HMAC-SHA-1 Design

HMAC is an application of hash functions. Figure 3.75 illustrates the top-level implementation of the HMAC architecture. It consists of input pad (Ipad) with the fixed value of 36 in hexadecimal format and output pad (Opad) with the fixed value of 5C in hexadecimal format. The Ipad values are used as an input to the first SHA-1 design. Subsequently, the counter activates the HMAC count to produce the output of the initial SHA-1, which serves as the input for the subsequent SHA-1, resulting in the comprehensive HMAC output. Figure 3.74 shows that the input message from Key ⊕Ipad will be combined with the text input. These values will be the input message for SHA-1 hash function. By using the initialisation value and message input, there will be two times of block input of 512 bits. This design comprises 64 states to produce the sequence input for both the initial and subsequent SHA-1 designs. For the first SHA-1 design, 32 states are executed in the processing of the data:16 states for the first 512 bits input, and another 16 states for the second 512 bits input of SHA-1 design.

In the second 512-bit loop, the message will utilise the output from the initial 512 bits of the SHA-1 hash function as inputs to generate the output for the first SHA-1 design. HMAC_count is utilised to produce the accurate output of the SHA-1 hash function by employing a counter to activate the output. The input concept for the second SHA-1 design

is identical to the input ipad. First, the output Key ⊕ opad will be combined with the output of the first SHA-1.

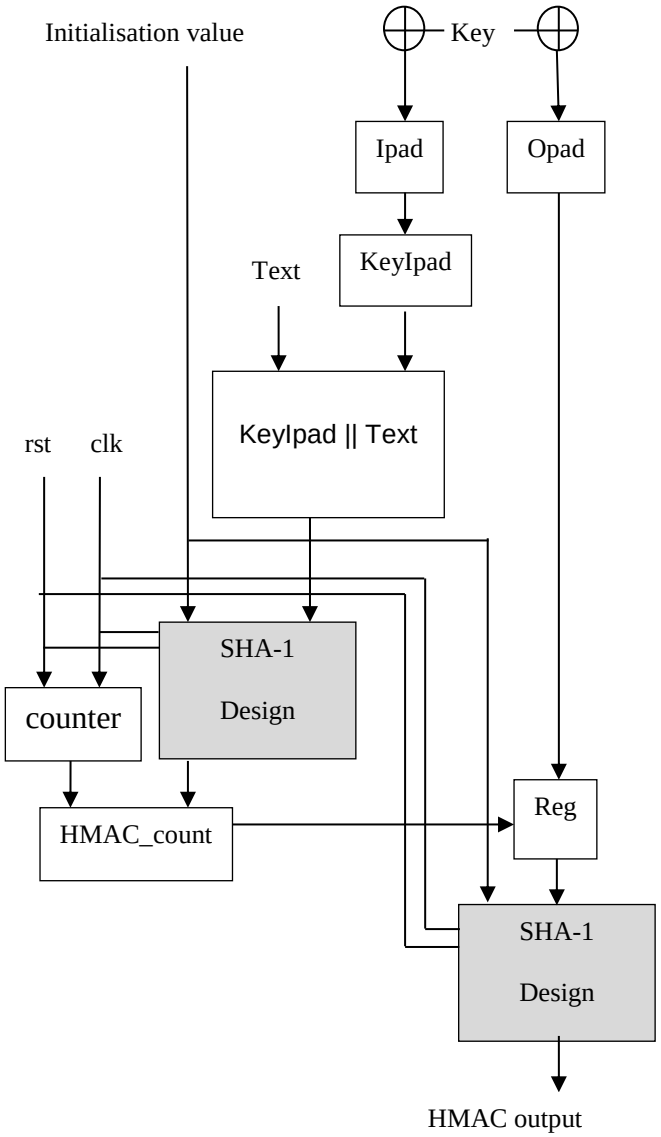


Figure 3.75 Implementation of Proposed HMAC-SHA-1

Subsequently, there will be two blocks of 512 bits; the output from Opad will constitute the first block of 512 bits, while the second block of 512 bits is followed with the output from the initial SHA-1 design. The states start from 33 states to 48 states. Then, for the second, the states will be executed until 64 states. Similar to the prior SHA-1 design process, the output of the second block in the second SHA-1 design will utilise the output from the initial 512 bits of the second SHA-1 hash function. The output of HMAC-SHA-1

has been generated. Figure 3.76 illustrates the top-level architecture of the HMAC-SHA-1 design. From this figure, there are five modules of the HMAC-SHA-1 design.

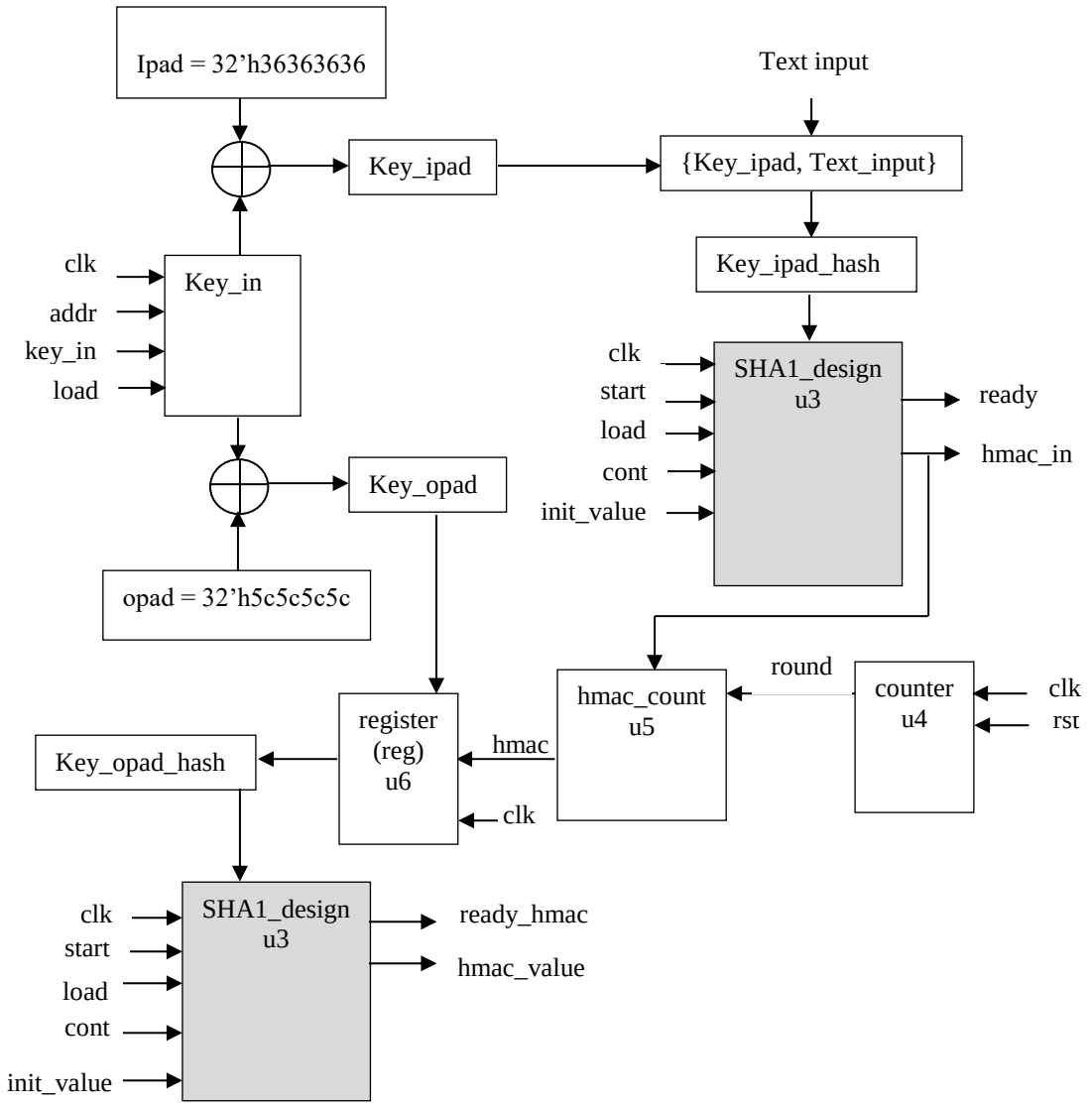


Figure 3.76 Top-Level Architecture of HMAC-SHA1 Design

The remaining designs are written at the top level of the HMAC-SHA-1 module, including the output of the key XORed with Ipad and Opad. Furthermore, the execution procedure of states for both SHA-1 implementations is stated in the top-level architecture of the HMAC-SHA-1 design. This is because the data are chosen based on the correct count of this module. The register module that combines the output opad data with the output of first SHA-1 design is to be hashed one more time to obtain the results.

3.6.5.2 HMAC-SHA-256 Design

One of the applications that use the hash function to verify message authentication is HMAC. An HMAC based on SHA-256 was implemented in this study. Before any input text is hashed using HMAC, the input key is first hashed. The input key is generated by XORing the inner pad (Ipad) with the input text. An inner pad value is 36 hexadecimal values with a 64-byte timeout, depending on the length of the key. The result of this SHA-256 hash function will be input into the subsequent SHA-256 to generate the HMAC design output. Before hashing the first SHA-256 output, the key inputs must be XORed with the outer pad (Opad), which has 5C values in hexadecimal (NIST FIPS, 2002). The concatenation of the 64-byte key, K_0, and Opad outputs, as well as the output of the first SHA-256, will then be hashed together to produce the HMAC output. Equation 3.48 shows an MAC calculated with the HMAC function over a textual representation of the data, while Figure 3.77 depicts the HMAC structure using SHA-256. The symbols $\oplus$ and || stand for XOR and concatenation, respectively.

$$MAC(\text{Message input})_t = HMAC(K, \text{Message input})_t$$

$$H((K_0 \oplus \text{Opad}) \,||\, H((K_0 \oplus \text{Ipad}) \,||\, \text{Message input}))_t \qquad \text{Equation 3.48}$$

MAC is an application of the hash function. Figure 3.77 illustrates the overarching implementation of the HMAC architecture, which includes an input pad (Ipad) featuring a constant 36 hexadecimal value and an output pad (Opad) with a fixed 5C hexadecimal value. The values from the Ipad are sent into the initial SHA-256 design. The counter then causes the HMAC count to generate the output of the first SHA-256 to the second SHA-256 in order to obtain the overall HMAC output. HMAC is a cryptographic authentication technique that uses a hash function and a secret key. To obtain the HMAC results, two identical hash

function architectures are used in the HMAC design. Meanwhile, two hash functions must be used in the HMAC architecture to complete the HMAC structure. In this study, SHA-256 algorithms were employed in the HMAC design.

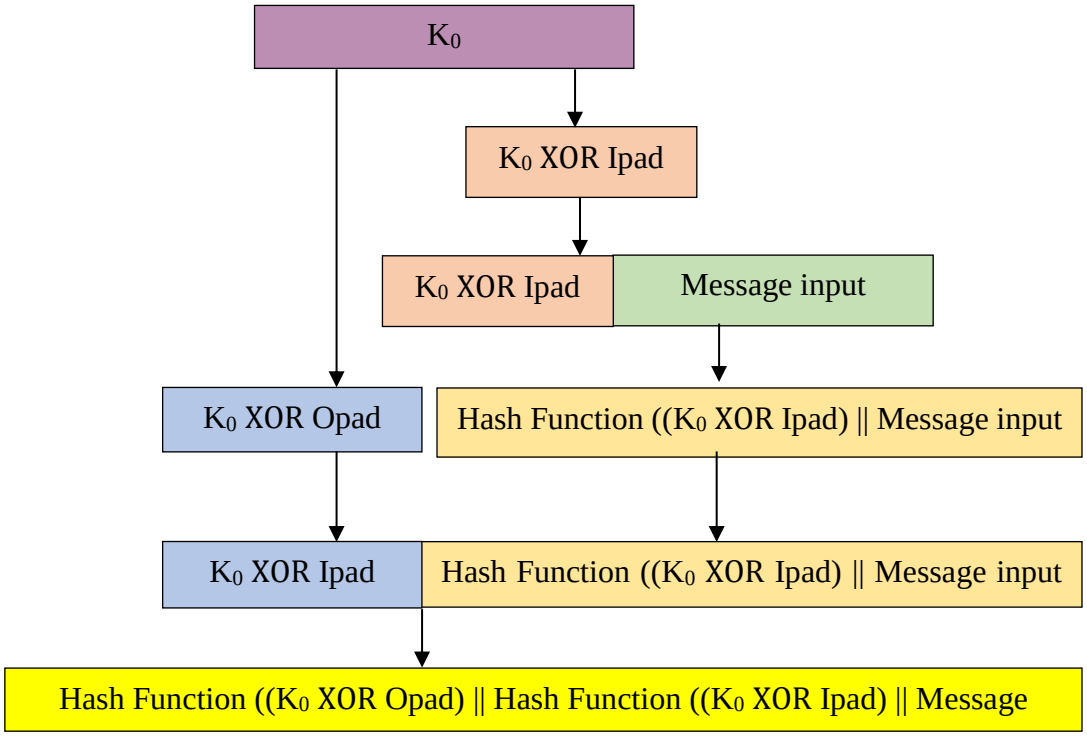


Figure 3.77 Illustration of HMAC Construction

Figure 3.78 depicts the architecture of an HMAC design with two SHA-256 hash algorithms. Moreover, it illustrates how the key Ipad input message will be integrated with the text input. These values will be used as the input message of SHA-256 hash function. There will be two instances of 512-bit blocks if the initialisation value and message input are used. The state machine controls the input for the first and the second SHA-256 in the proposed design. As a result, the first SHA-256 must be executed first, and the second SHA-256 must wait for the message input from the output of the first SHA-256 design, as shown in Figure 3.78.

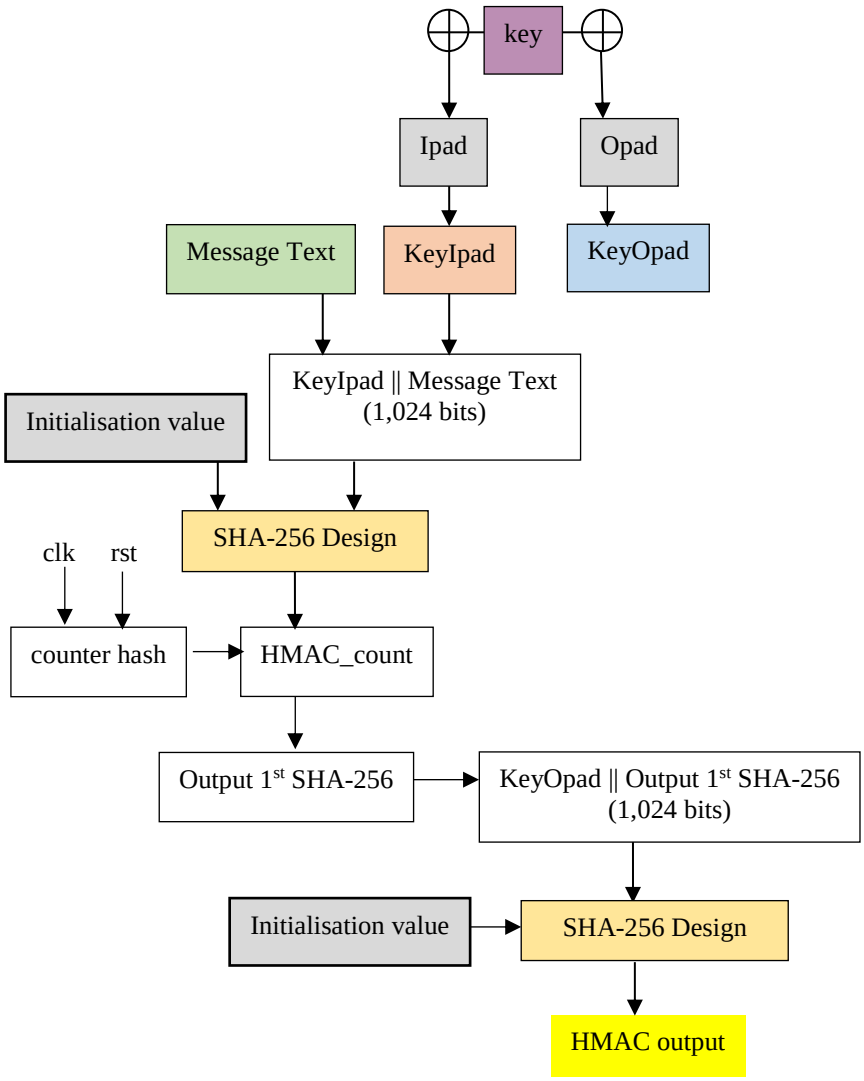


Figure 3.78 Implementation of Proposed HMAC-SHA-256

HMAC uses two hash functions to perform the operation. Two SHA-256 hash functions are employed in HMAC design. The initial SHA-256 algorithm must be executed until results are achieved. These findings serve as the input for the second SHA-256 algorithm with message input. Figure 3.78 depicts the concatenation of a key with input from an Ipad and SHA-256 with input from a message. Therefore, the validity of the results of initial SHA-256 remains crucial in acquiring the final HMAC based on SHA-256. The 64 states in this design have generated the sequence input for the first and second SHA-256 algorithms. To process the message for the first SHA-256, 32 states are executed: 16 states

for the first 512 bits and another 16 states for the second 512 bits for SHA-256. The message will utilise the output of the initial 512-bit hash function to derive the output of the subsequent 512-bit iteration of the input message. An HMAC count is used to generate the correct output SHA-256 hash function with the help of a counter.

Input Ipad is the same concept as message input to the second SHA-256. The key Opad output will be combined with the first SHA-256 output. There will be two 512-bit blocks. The state begins at 33 and progresses to 48 for the first 512 bits. The remaining 512 bits will then be executed until 64 states are reached. Like the previous SHA-256 process, the output of the second 512-bit block of the second SHA-256 will use the output of the first 512 bits of the second hash function. Finally, this will produce the resulting output, as obtained in this study.

Secure hash functions authenticate messages; therefore, it is necessary to comply with the hash function requirement. A hash function must possess certain properties. Some of these properties include the generation of outputs of fixed length and the computational infeasibility of determining an x, such that Hash(x) equals hash value. This is true for any given code h. Calculating the hash code is straightforward; however, recovering the original message through reversal is unattainable. In addition, it is computationally impossible to determine $y \neq x$ for any given block x for which Hash(x) = Hash(y). In other words, identical hash codes are impossible to find, and all of these characteristics constitute weak hash functions. Additionally, no pair (x, y) can be found for which Hash(x) = hash value, computationally, (y). If the last property of the hash function criteria is met, the function is considered “strong”. Any message less than 2^{64} bits in length can be entered into the SHA-

256 algorithm. SHA-256 processes 512-bit message inputs and 160-bit starting values to generate a 160-bit hash code output.

Several considerations must be made to generate output. The message must first be padded to almost exactly match 960 modulo 1,024. The pre-processing procedure commences by adding a 1-bit input subsequent to the input message. The message is subsequently padded with zero bits to get the desired overall length. After the message is padded, 64-bit inputs are added to the message. Message padding consists of 512 bits in addition to the message's total length. HMAC-based SHA-256 uses the SHA-256 approach to determine authentication in this design. In other words, the SHA-256 hash algorithm is utilised in the construction of HMAC. Based on the SHA-256 algorithm, the SHA-256 architecture has eight fixed inputs. Hence, the SHA-256 algorithm requires eight variables as initial input during hash computation.

3.7 Summary

This chapter outlines the design methodology used in the development and implementation of cryptographic hash functions on FPGA-based platforms. The workflow initiates with the functional simulation of the proposed design utilising Verilog HDL, confirming logical correctness before hardware mapping. The selected FPGA family is used to synthesise the Verilog code to gate-level representation, which is then translated into a netlist that accurately represents the hardware circuitry. Electronic design automation (EDA) tools are used for the compilation and synthesis procedures, and timing simulations are then used to verify setup and hold constraints, as shown in the output waveform. When verification is completed successfully, the final design is downloaded onto the FPGA board for real-time evaluation.

Methods of unfolding transformation were incorporated into the design to improve the performance of hash function implementation. Unfolding factors 2 and 4 were introduced to minimise the overall latency of the cryptographic algorithm. The design achieves multiple rounds of computation per clock cycle by executing two and four iterations, respectively. The decrease in latency is directly related to a reduction in the number of clock cycles needed for each message block. This enhancement leads to improved throughput and supports a high-performance hash function that is appropriate for real-time cryptographic applications.

In addition, the main performance metrics, including speed, area, and power, are examined in relation to the unfolding techniques. By implementing unfolding factors 2 and 4, the processing time is significantly reduced, resulting in a significant increase in performance. While the area utilisation increases slightly due to resource duplication required for parallel execution, the trade-off is acceptable considering the speed improvements. Furthermore, dynamic power consumption is minimised as the number of toggling events inside the logic parts decreases according to latency, reducing unnecessary switching activities.

The fundamental contribution of this methodology is the invention and incorporation of unfolding-based optimisation into the MD5, SHA-1, RIPEMD-150 and SHA-256 hash function designs. The proposed architecture not only maximises throughput but also accomplishes energy-efficient hardware by reducing dynamic power. These results contribute to the advancement of low-power and high-speed cryptographic cores, which are in accordance with the broader research objective of achieving efficient hash function designs on FPGA platforms and meet the needs of contemporary embedded security applications.

CHAPTER 4

RESULTS AND DISCUSSION

4.1 Introduction

The results of several hash function designs and their use in HMAC are covered in this chapter. A synthesis process must be performed prior to the implementation phase of the design to ensure that there are no syntax errors. ModelSim simulation tool is used in the simulation phase to ensure that the design output is accurate. This chapter includes a brief explanation of simulation findings for several types of hash functions, in conjunction with a simulation graph analysis to evaluate the design outcomes. The designs are coded in Verilog HDL, while Quartus II serves as the CAD tool. The designs are executed on an FPGA device, and the accuracy of the designs is verified through the evaluation of functional simulation and timing simulation results. Compilation entails synthesising and implementing the design. Figure 4.1 illustrates the flowchart representing the design process of the simulation design flow. Functional simulation is utilised to verify the accuracy of the design, whereas timing simulation considers the delay of the design. When designing hardware, gate delay must be considered. This will continue until the simulation generates the accurate output, resulting in the generation of multiple outputs due to the latency during timing simulation. To acquire the output from timing simulation, the ".vo" file with the Arria II GX device must be compiled alongside other Verilog design files. When the designs are successful and produce the expected results, the implementation procedure begins, and the designs are already fitted on the target device, which is the FPGA.

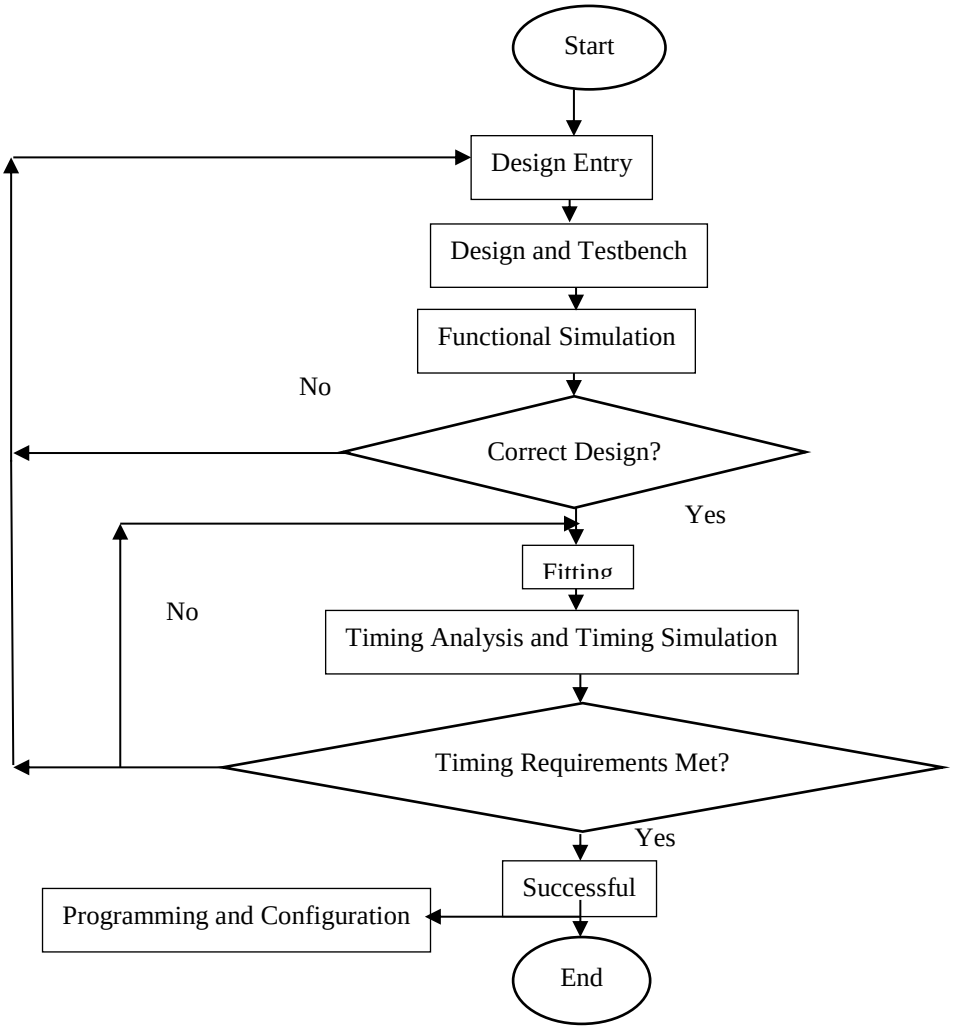


Figure 4.1 CAD Tool Simulation Flow Design

It is imperative to analyse the critical routes of the designs and consider the maximum frequency of the circuit when developing high-performance hash functions and HMAC designs. Three strategies are employed to enhance the performance of the designs: modifying the HDL coding style, utilising the Timing Optimization Advisor, and employing the TimeQuest Timing Analyzer. This includes optimisations for physical synthesis, routing effort, and logic cell insertion. A timing verification process that assesses the time requirements of hash function and HMAC designs is provided. It supports the industry standard Synopsys design constraint (SDC) format. A compilation report will give the

amount of logic utilisation, such as combination ALUTs and total number of registers. This project involved the design of hash functions and HMAC in Verilog HDL, compilation using Altera Quartus II, and simulation with ModelSim software. The design was implemented and downloaded into an FPGA device to check its accuracy. Its frequency and throughput performance were evaluated in comparison to previous design implementations. Improving the implementation's performance is the primary goal of this design.

This study concentrates on the design, optimisation, and implementation of cryptographic hash functions at the hardware level, utilising Verilog HDL on FPGA platforms. The goal is to enhance the performance metrics, specifically speed, area utilisation, and dynamic power consumption, through the implementation of techniques including unfolding transformation and Gray encoding.

Although cryptographic primitives require cryptographic strength and resistance to assaults (e.g., collision resistance, preimage resistance), these evaluations are beyond the scope of this work, which does not propose any new cryptographic primitive or hash algorithm. This research utilises established and extensively studied hash functions, specifically MD5, SHA-1, SHA-256, and RIPEMD-160, which have well-documented theoretical cryptographic properties in the existing literature. This work contributes indirectly to hardware-based security systems by optimising the hardware implementation of these algorithms. This includes applications such as HMAC modules in network security, embedded encryption devices, and FPGA-based cryptographic engines. The proposed design methodology facilitates the secure and efficient deployment of hash functions in resource-constrained environments, such as IoT and embedded systems.

4.2 Results and Discussion of MD5 Design Architecture

The MD5 hash function algorithm is widely used in cryptography. This section briefly explains the architecture of MD5. The design incorporates MD5 data paths and padding blocks. The message is processed by padding the message input with logic '1' at the end of the message input. Subsequently, a sequence of zero values persists until the 448th bit. The final 64 bits are reserved for the message length. In this research, the message input, M for MD5 design is "**MD5 was proposed by Ron Rivest in 1992**". This message is converted to hexadecimal format by referring to the ASCII table in Table 4.1.

Table 4.1 ASCII Code of Message Input of MD5 Design

Bit (MSB–LSB)	**8-bit Binary Hexadecimal:Symbol**							
511–448	01001101 4D: M	01000100 44:D	00110101 35:5	00100000 20:space	01110111 77:w	01100001 61: a	01110011 73:s	00100000 20:space
447–384	01110000 70:p	01110010 72: r	01101111 6F: o	01110000 70: p	01101111 6F: o	01110011 73: s	01100101 65: e	01100100 64: d
383–320	00100000 20:space	01100010 62: b	01111001 79: y	00100000 20:space	01010010 52: R	01101111 6F: o	01101110 6E: n	00100000 20:space
319–256	01010010 52: R	01101001 69: i	01110110 76: v	01100101 65: e	01110011 73: s	01110100 74: t	00100000 20:space	01101001 69: i
235–164	01101110 6E: n	00100000 20:space	00110001 31: 1	00111001 39: 9	00111001 39: 9	00110010 32: 2	00101110 2E: .	

The first step for message padding is the addition of a single bit to the end of the message input. Subsequently, the following bits remain zero until the 448th bit. The final 64 bits will represent the message length. The message length was calculated as $39 \times 8 = 312$ ($2^8 + 2^5 + 2^4 + 2^3$). Finally, 512-bit input messages were converted to the little-endian format to obtain the MD5 hash code, as shown in Table 4.2. From Table 4.2, the messages are translated using the little-endian format with 32-bit words.

Table 4.2 512-bit message input

8-bit Binary Hexadecimal:Symbol Bit (MSB……..LSB)							
00100000 20:space	00110101 35:5	01000100 44:D	01001101 4D:M	00100000 20:space	01110011 73:s	01100001 61: a	01110111 77:w
01110000 70: p	01101111 6F: o	01110010 72: r	01110000 70:p	01100100 64: d	01100101 65: e	01110011 73: s	01101111 6F: o
00100000 20:space	01111001 79: y	01100010 62: b	00100000 20:space	00100000 20:space	01101110 6E: n	01101111 6F: o	01010010 52: R
01100101 65: e	01110110 76: v	01101001 69: i	01010010 52: R	01101001 69: i	00100000 20:space	01110100 74: t	01110011 73: s
00111001 39: 9	00110001 31: 1	00100000 20:space	01101110 6E: n	10000000 80:	00101110 2E: .	00110010 32: 2	00111001 39: 9
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000001	00111000	00000000	00000000

There are 16 message blocks; each message block represents 32-bit message. Therefore, the overall message has 512 bits. In Table 4.3, there are 16 32-bit words written in the hexadecimal format.

Table 4.3 Message in the Little-Endian Format

Message in Hex	**Message in the Little-Endian Format**
4d443520	2035444d
77617320	20736177
70726f70	706f7270
6f736564	6465736f
20967920	20796220
526f6e20	206e6f52
52697665	65766952
73742069	69207473
6e203139	3931206e
39322e80	802e3239
00000000	00000000
00000000	00000000
00000000	00000000
00000000	00000000
00000000, 00000138	00000138,00000000

4.2.1 MD5 Iterative Design

MD5 designs are successfully designed using Verilog HDL. The MD5 designs were synthesised and implemented based on Arria II GX Altera Quartus II. The results of MD5 designs are based on frequency, area, and power consumption. The MD5 designs are simulated using ModelSim to evaluate the output of the designs in terms of both functional and timing simulation. MD5 iterative design consists of five different types of MD5 designs such as MD5 binary, MD5 Gray, MD5 binary with Gating, MD5 Gray with gating and MD5 binary with controller. MD5 Binary represents the MD5 design with binary encoding, MD5 Gray denotes the MD5 design with Gray encoding, MD5 Gating refers to MD5 binary encoding with clock gating, MD5 Gray gating refers to MD5 Gray encoding with clock gating and MD5 binary with controller. The PowerPlay Power Analyser was employed to evaluate the power analysis in order to confirm the impact of switching activity, including state encoding and clock gating, on the MD5 designs.

The top-level RTL architectures of five different MD5 design variants are presented in the Appendix section to illustrate the structural differences resulting from the applied design techniques. This research involves five types of MD5 designs. MD5 binary design, has 12 modules; MD5 Gray design has eleven modules; MD5 binary with gating design has 12 modules, and MD5 Gray with gating design has eleven modules. Meanwhile, MD5 controller design has eight modules. All MD5 algorithms are designed in iterative looping. Thus, there are 64 iterative rounds that need to be processed to obtain the 128-bit MD5 hash output.

4.2.1.1 Results of MD5 Iterative Design

Four circuits comprise the primary loop: F, G, H, and I. For t = 0 to t = 8, the non-linear function F was executed during the initial round of the message, as illustrated in Figure 4.2. A_init, B_init, C_init, and D_init are the initial inputs of MD5, as represented in this figure. The value of buffer initialisations needs to be converted into the little-endian format to obtain the correct results.

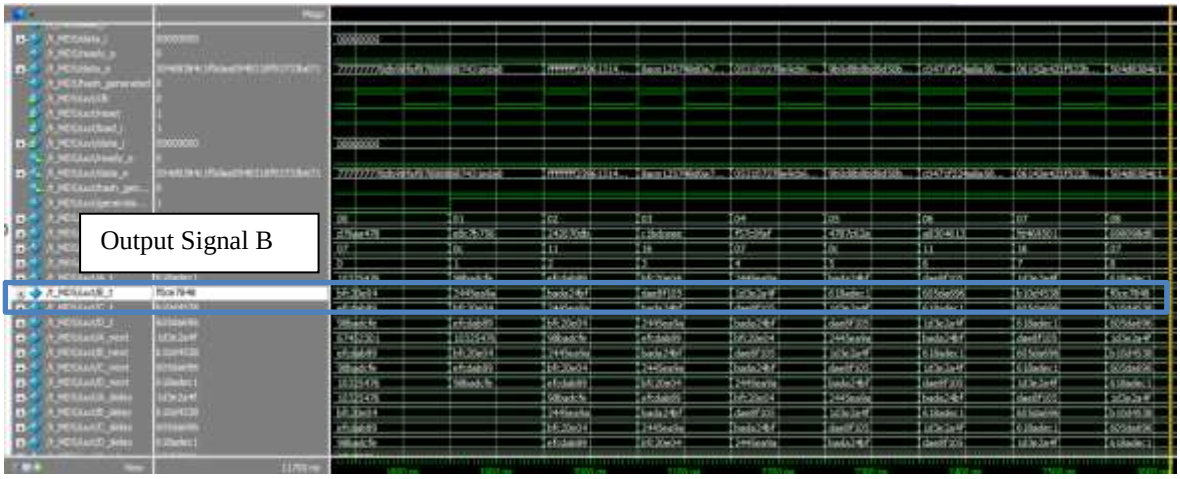


Figure 4.2 Non-Linear Function F for B_t (t = 0 to t = 8)

Table 4.4 Round 1 of Non-linear Function F

Function(a,b,c,d,message,shift,constant)	Output B_t
F0 (a,b,c,d,m_0,7,d76aa478)	bfc20e04
F1 (a,b,c,d,m_1,12, e8c7b756)	2445ea9a
F2 (a,b,c,d,m_2,17, 242070db)	bada24bf
F3 (a,b,c,d,m_3,22,c1bdceee)	dae8f105
F4 (a,b,c,d,m_4,7,f57c0faf)	1d3e2a4f
F5 (a,b,c,d,m_5,12,4787c62a)	618adec1
F6 (a,b,c,d,m_6,17,a8304613)	605da696
F7 (a,b,c,d,m_7,22,fd469501)	b10d4538
F8 (a,b,c,d,m_8,7,698098d8)	f0ce7848
F9 (a,b,c,d,m_9,12,8b44f7af)	adc2ea19
F10 (a,b,c,d,m_{10},17,ffff5bb1)	8ca10c71
F11 (a,b,c,d,m_{11},22,895cd7be)	d06eda96
F12 (a,b,c,d,m_{12},7, 6b901122)	cfc79c1a
F13 (a,b,c,d,m_{13},12,fd987193)	ef0992d6
F14 (a,b,c,d,m_{14},17,a679438e)	419bb7da
F15 (a,b,c,d,m_{15},22, 49b40821)	a41613f9

The total round for MD5 algorithm is 64 rounds. As there are 16 steps in each round, the final output for round 1 of non-linear function F consists of 16 different results of output B_t. This output is based on message input, shift input, and constant input provided, as shown in Table 4.4. The subsequent non-linear function is G. The operation is identical to the previous function. The differences between non-linear function F and non-linear function G are the value of message, shift, and constant. Subsequently, the last two non-linear functions that need to be executed to obtain the results of hash function are non-linear function H and I.

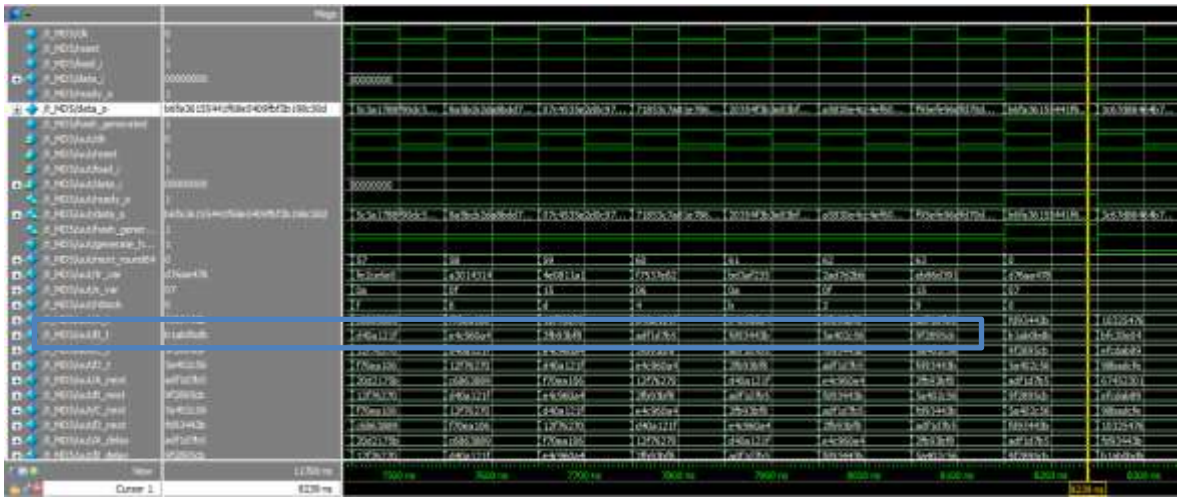


Figure 4.3 Non-linear Function I for B_t (t = 57 to t = 63)

Figure 4.3 shows the simulation waveform for the last non-linear function I for signal output B_t (*t* = 57 to *t* = 63). All simulation results for function F, G, H, and I for output signal B_t is provided in the Appendix section. The second non-linear function simulation output B_t for function G starts from $16 \leq t \leq 31$.Then, this is followed by the next round 3, which consists of the non-linear function H. The simulation results start from $32 \leq t \leq 47$. Finally, the non-linear function I is executed in order to obtain hash output from $48 \leq t \leq 63$. Overall, the results need to be examined thoroughly starting from non-linear function F, G, H, and I to avoid any error, especially the sequence of the message, shift, and

constant value for the specific time. The results of the output signal B_t for the non-linear function I are presented in Table 4.5. Figure 4.3 illustrates the final round of the MD5 algorithm, which is represented by the sequence I63 as shown in Table 4.5.

Table 4.5 Round 4 of Non-linear Function I

Function(a,b,c,d,message,shift,constant)	**Output B_t**
I48 (a,b,c,d, m_0, 6, f4292244)	bc2cf190
I49 (a,b,c,d, m_7, 10, 432aff97)	c43bf785
I50 (a,b,c,d, m_{14}, 15, ab9423a7)	9d557285
I51 (a,b,c,d, m_5, 21, fc93a039)	bf063e88
I52 (a,b,c,d, m_{12}, 6, 655b59c3)	c5ec3319
I53 (a,b,c,d, m_3, 10, 8f0ccc92)	20d2175b
I54 (a,b,c,d, m_{10}, 15, ffeff47d)	c6863889
I55 (a,b,c,d, m_1, 21, 85845dd1)	f70ea106
I56 (a,b,c,d, m_8, 6, 6fa87e4f)	12f76270
I57 (a,b,c,d, m_{15}, 10, fe2ce6e0)	d40a121f
I58 (a,b,c,d, m_6, 15, a3014314)	e4c960a4
I59 (a,b,c,d, m_{13}, 21, 4e0811a1)	2fb93bf8
I60 (a,b,c,d, m_4, 6, f7537e82)	adf1d7b5
I61 (a,b,c,d, m_{11}, 10, bd3af235)	fd93443b
I62 (a,b,c,d, m_2, 15, 2ad7d2bb)	5a402c56
I63 (a,b,c,d, m_9, 21, eb86d391)	**9f2895cb**

An explanation of the computational design and verification results of the MD5 hash function is provided in this section. The MD5 algorithm resolves in the integration of its final output with four primary inputs, which are labelled as A, B, C, and D. The synthesised output from this architectural setup is documented in Table 4.6. Following 64 processing rounds, the output signal B_t is identified with the hexadecimal value "**9f2895cb**". This value, alongside the initial values of the MD5 hash function, undergoes a final transformation to establish the final hash result. This result is then transformed from the little-endian format to finalise the hash function's output.

Table 4.6 Final MD5 Hash Function

Register	A	B	C	D
Initial hash value	32'h67452301	32'hefcdab89	32'h98badcfe	32'h10325476
Round output	32'hadf1d7b5	**32'9f2895cb**	32'h5a402c56	32'hfd93443b
Final transformation	32'h1536fab6	32'h8ef64154	32'hf2fb0954	32'h0dc598b1
Conversion from little-endian format	32'hb6fa3615	32'h5441f68e	32'h5409fbf2	32'hb198c50d
Final hash	**b6fa36155441f68e5409fbf2b198c50d**			

This output was verified using HashCalc software, and the MD5 designs are shown in Figure 4.4. These outcomes precisely match the 128-bit final hash value "**b6fa36155441f68e5409fbf2b198c50d**", which validates the hash output derived from the ModelSim simulation. Table 4.6 provides additional clarification on this hash value, which is the combination of the outputs from signals A, B, C, and D.

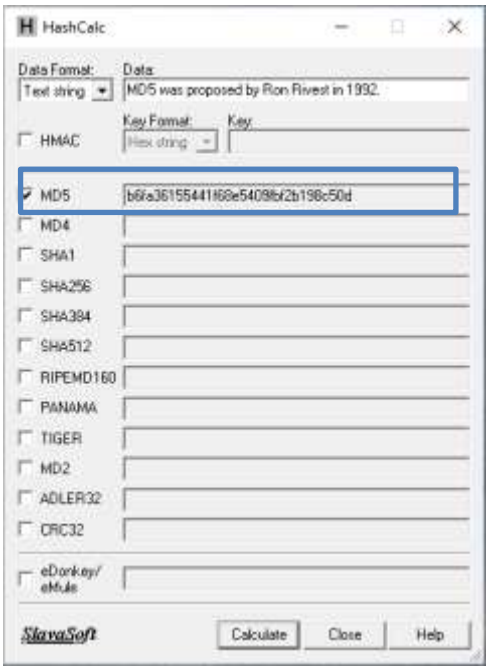


Figure 4.4 Output of MD5 Hash Function

The comparisons of frequency maximum for different types of MD5 designs were analysed. In this analysis, the input message, m, "MD5 was proposed by Ron Rivest in 1992"

was tested. Table 4.7 illustrates the synthesis and the implementation results for MD5 binary, MD5_Gray, MD5_Gating, and MD5_Gray_Gating. Meanwhile, the other two designs of MD5 are for clock-gated implementation. MD5_Gating was clock-gated for encoding, whereas MD5_Gray_Gating demonstrated clock-gated for Gray encoding. Simulation was carried out using ModelSim-Altera to verify the correctness of the Verilog coding output. The results for the maximum frequency and area implementation of MD5 designs are presented in Table 4.7.

Table 4.7 Maximum Frequency and Area Implementation of MD5 Design

Design	**MD5 Binary**	**MD5 Gray**	**MD5 Binary Gating**	**MD5 Gray Gating**
Constraint (SDC)	7	10	7.5	10.5
FMax	154.61 MHz	134.16 MHz	141.62 MHz	137.7 MHz
ALUT	1,623	1,818	1,647	1,793
Total Register	650	974	649	973

From this table, MD5 binary encoding gives the highest maximum frequency ,154.61 MHz) with small area implementation, which is 1,623 and 650 for ALUTs and total number of registers, respectively. The performance of the overall design is almost similar, where the frequency maximum for four different types of the design is in the same range. From Table 4.7, the frequency maximum for MD5_Gray is the lowest among all these designs. Moreover, by gating the clock into the MD5 design, the speed of the design could be improved. From this analysis, the switching bit of the state did not affect the performance in terms of speed of the design. Furthermore, these designs reduced power consumption due to the switching bit of the state during the transition process. Figures 4.5, 4.6, 4.7, and 4.8 show the timing simulation results of MD5 Binary, MD5 Gray, MD5 Binary with Gating, and MD5 Gray with gating, respectively. These results exactly match the output signal of the

128-bit final hash value “b6fa36155441f68e5409fbf2b198c50d”, validating the hash output obtained from the ModelSim simulation.

Modifications to the MD5 design can produce varied effects. The MD5 design's performance was assessed in terms of power consumption, area, and speed. A controller is added to the design to execute the data automatically once the clock is supplied to the design. In other words, the message data are kept in the ROM file. The outcomes of the maximum frequency (FMax) for MD5 designs are presented in Table 4.8. From these results, the input data are already in the ROM file, which is written in Verilog code.

Table 4.8 Maximum Frequency of MD5b Design

Design	Clk Constraint	Fmax (MHz)
MD5 with Controller	10	135.78

The increment of the maximum frequency can produce the high-speed design because a higher throughput design requires a higher speed design with minimum delay. Nevertheless, the area implementation of the design needs to be considered as well. The logical implementation of the MD5 design is categorised into two components: combinational ALUTs and the total number of registers, as seen in Table 4.9. From this table, the usage of area implementation for both designs are almost similar. The number of combinational ALUTs for MD5 with Controller design is 712. In this design, MD5 with Controller design uses only a total of 586 registers.

Table 4.9 Area Implementation of MD5 Design (Arria II GX: EP2AGX45DF25C4)

Design	Logic Utilisation	
	Combinational ALUTs	Total Registers
MD5 with Controller	712	586

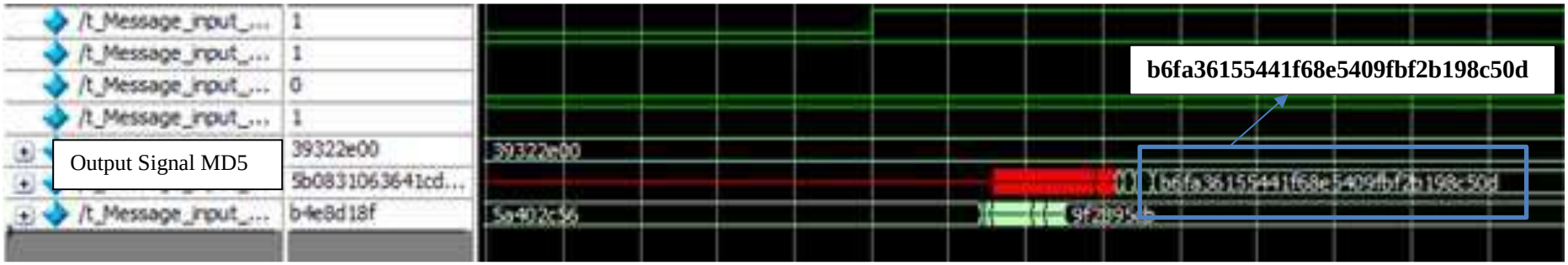


Figure 4.5 Simulation Results of MD5 Binary Design



Figure 4.6 Simulation Results of MD5 Gray Design

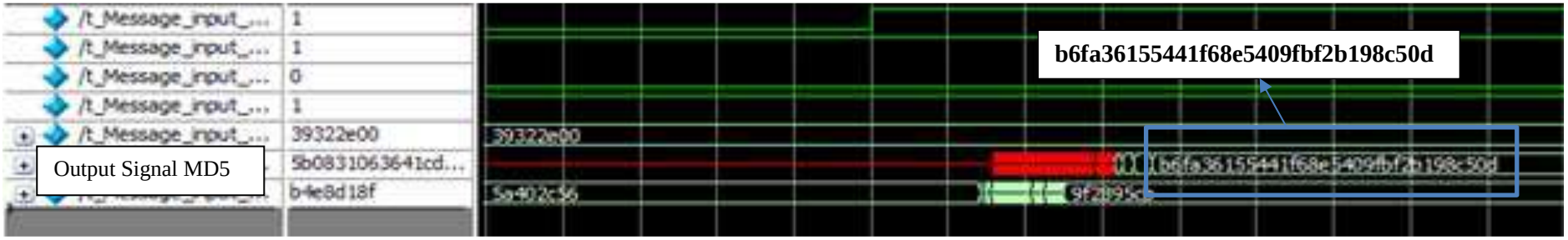


Figure 4.7 Simulation Results of MD5 Binary with Gating Design

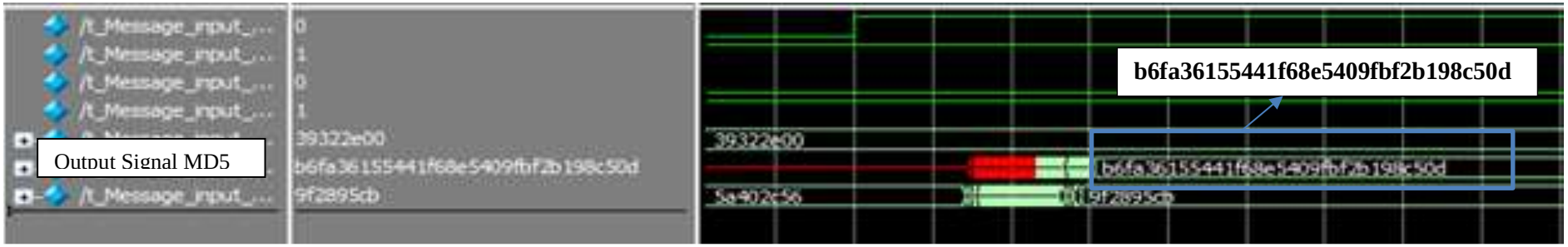


Figure 4.8 Simulation Results of MD5 Gray with Gating Design

The results of MD5 design implementation on reconfigurable hardware depend on the choice of FPGA family devices and the structure of HDL code chosen. Consequently, following the design of the MD5 architecture, the synthesis and implementation stages were required to determine the design's critical path. In this study, the TimeQuest Timing Analyzer was used to optimise and produce better FMax with clock constraint input. Each design must specify the accurate value for clock constraint. Therefore, the iteration process was executed several times for all designs. Moreover, there are several reasons for applying the TimeQuest in designing the MD5 for this project. It is easier to use because it provides a simple GUI and interactive reporting for analysing timing, industry standard, and more powerful with SDC format, allowing for faster, easier description, and analysis of advanced design constructs. The simulation results display all input and output signals within the MD5 architecture. Figure 4.9 shows the simulation results with different input values. The output data_o signal of the MD5 design for the input value "a" is "**Occ175b9c0flb6a831c399e269772661**". The outcome from the ModelSim simulation matched the anticipated result of the MD5 hash code, thereby confirming the successful verification of the MD5 design.

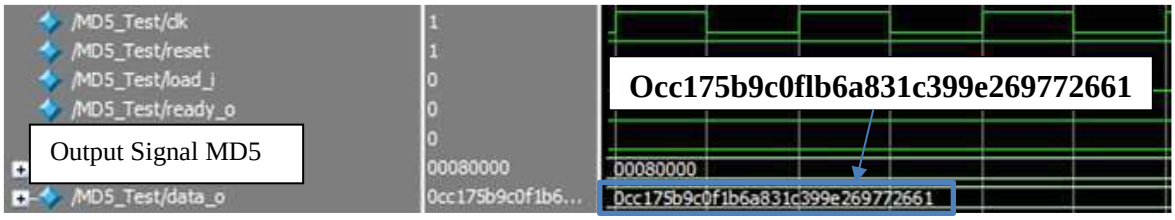


Figure 4.9 Simulation Results of MD5 Design with Input "a"

The primary objective of developing various MD5 designs for this research was to assess their performance regarding speed and area, utilising Arria II GX family devices from Altera Quartus II. Besides, introducing appropriate clock constraint of the design could increase its performance, as well as meet the timing requirement with positive slack. In other

words, in order to create a high-performance design, the circuit's performance design must be improved while the design's critical path might be shortened. Consequently, it is necessary to shorten the MD5 design's critical path in this study. The increment of the frequency maximum can produce a high-speed design because a higher throughput design requires higher speed design with minimization of delay. Additionally, modifying the HDL design style can also be crucial to improving the MD5 design's performance.

4.2.1.2 Result of Low-Power MD5 Iterative Design

The PowerPlay Analyzer may be used to determine the MD5 design's power consumption. It is divided into two parts: design concept and design implementation. During design concept, PowerPlay Early Power Estimator is used based on user input and Quartus Design Profile. For design implementation, it can be analysed using Quartus II PowerPlay Power Analyzer during place and route and from simulation results. Table 4.10 shows the results of simulation-based power estimation of four different types of MD5 designs. Power dissipation can be broken down into three distinct components using this table: I/O thermal power dissipation, static power dissipation, and dynamic thermal power dissipation. As mentioned previously, signal toggling causes dynamic power, leakage currents cause static power, and I/O power takes into account the I/O pin, as well as every characteristic that might possibly describe the off-chip board trace at each I/O pin (Altera, 2010). Altera Quartus II PowerPlay Power Analyzer is a power analysis tool that can provide the most accurate power analysis; it uses the actual design placement and routing and logic configuration. In other words, it is capable of offering a precise assessment of power consumption. The accuracy of the PowerPlay Power Analyzer is usually within a range of ±10% when accurate design information is provided (Altera, 2010). These power predictions

generated by the PowerPlay Power Analyzer tool are accurate as if the designs are verified with the actual silicon.

The experimental results indicate that the MD5 Binary-encoded design demonstrates the highest total thermal power dissipation, measured at 363.70 mW, principally attributable to its increased dynamic power, which exceeds static and I/O power components. In contrast, the static and I/O power values across all MD5 variants remain relatively consistent, indicating that the observed differences are primarily attributed to switching activity within the design logic. Dynamic power dissipation was greatly decreased by using Gray encoding. The MD5 Gray-encoded architecture showed a decrease of 21.52 mW, demonstrating the effectiveness of limiting state transitions to a single bit per cycle. Similarly, incorporating clock gating into the MD5 Binary design reduced dynamic power to 21.16 mW by eliminating unnecessary switching during idle periods. This is important to note that Gray encoding and clock gating were combined to produce the lowest dynamic power dissipation at 19.21 mW, thereby confirming the beneficial impact of both methods in reducing switching activity. This design also maintained a moderate area usage, with 649 registers, and operated at a maximum frequency of 141.62 MHz. While this frequency is slightly lower than the MD5 Binary variant, it is well within acceptable limits for high-throughput applications. As a result, the MD5 Gray-encoded design with clock gating provides the most power-efficient implementation of the four MD5 designs, making it the best choice for high-performance, low-power cryptography applications. With careful register placement and optimised HDL coding techniques, Gray encoding and clock gating were integrated to offer significant dynamic power reductions on the FPGA without sacrificing functionality or timing closure. This result demonstrates that the best way to lower dynamic power in FPGA-based MD5 implementations is to use both clock gating and Gray encoding.

Table 4.10 Simulation-based Power Estimation of Four Different Types of MD5 Designs

Design	MD5 Binary	MD5 Gray	MD5 Binary Gating	MD5 Gray Gating	MD5 Controller
Total Thermal Power Dissipation	363.70 mW	360.51 mW	359.99 mW	358.88 mW	410.26
Core Dynamic Thermal Power Dissipation	24.23 mW	21.52 mW	21.16 mW	19.21 mW	11.43
Core Static Thermal Power Dissipation	321.14mW	320.99 mW	320.54 mW	321.67 mW	317.70
I/O Thermal Power Dissipation	18.33 mW	18.00 mW	18.29 mW	18.00 mW	81.13

The power estimate for each MD5 design variant was determined by utilising the Value Change Dump (.vcd) file, which was produced from the gate-level simulation in ModelSim after functional verification. These files capture the design's switching activity and provide input to the PowerPlay Power Analyser in Altera Quartus II, which generates comprehensive reports on static and dynamic power dissipation. All four MD5 designs, such as binary-encoded, Gray-encoded, binary with clock gating, and Gray with clock gating, were designed using Verilog HDL and synthesised using the Altera Arria II EP2AGX45DF25C4 FPGA. The synthesis process included area, timing, and power analysis.

Pipeline registers were manually inserted at the output of the non-linear function blocks within the MD5 datapath to enhance the maximum operating frequency (FMax). This reduced the critical path delay by breaking long combinational logic paths and allowed the circuit to meet tighter timing constraints. The Quartus II Timing Analyser indicated that register placement significantly improved setup timing and led to an increase in FMax. Additionally, register balancing and pipelining not only improved frequency performance

but also facilitated more precise clock gating control. The implementation of clock gating in specific control logic and idle states resulted in a reduction of unnecessary toggling. The combination with Gray encoding proved particularly effective, as it inherently reduces switching activity through single-bit transitions between states.

The Gray-encoded design with clock gating among the four MD5 designs demonstrated the lowest dynamic power consumption, as validated by the Power Analyser results. This proves that combining low-power methods like selective clock gating and Gray encoding with well-placed registers may greatly increase the power efficiency of FPGA-based cryptography hardware. The Quartus II Resource Optimisation Advisor and Timing Optimisation Advisor were utilised to refine design constraints and synthesis strategies, ensuring that each design variant achieved optimal trade-offs among area, speed, and power. In summary, the MD5 Gray encoding with clock gating design is the most power-efficient implementation of all the versions that were studied, which makes it ideal for FPGA low-power cryptography applications.

4.2.1.3 Discussion of MD5 Iterative Design

Table 4.11 shows the results for the synthesis and implementation of MD5 design based on both Altera and Xilinx from previous research. The output results for MD5 (Y. Wang et al., 2010), which is based on iterative technique, can improve the frequency maximum of the design by using a Carry Save Adder (CSA) with 102.7 MHz of frequency maximum. On the other hand, MD5 (Jarvinen et al., 2005) provides fast performance and large area logic resources for pipelining design, whereas the results based on iterative implementation using Xilinx give small logic area implementation with 78.3 MHz of frequency maximum. Furthermore, the iterative looping for MD5 (Díez et al., 2002) using Xilinx, which involves iterating several times to perform a complete loop, gives 60.20 MHz

of frequency maximum. Meanwhile, the implementation of MD5 (Deepakumara et al., 2001b) shows that the device utilisation of iterative design is significantly small, resulting in 880 slices with 21 MHz of frequency maximum. It also shows that the implementation of MD5 on Stratix II GX gives 103.32 MHz with clock constraint 11, while MD5 (Y. Wang et al., 2010) contributes only 102.7 MHz. However, this design uses large area implementation. Besides, it can be clearly seen that the Arria II GX family device provides high frequency maximum, which is 152.32 MHz, with combinational ALUTs of 1,989, and the usage of total register is 1,713. Hence, Arria II can contribute to high performance of the design, as well as increase the throughput of the design.

Table 4.11 Synthesis and Implementation of MD5 Design

Design	**Device**	**FMax (MHz)**	**Combinational ALUTs (Altera)**	**Total Register (Altera)**	**Slices (Xilinx)**
MD5 Binary	Arria II GX	154.61	1,623	650	-
MD5 Gray	Arria II GX	134.16	1,818	974	-
MD5 Binary Gating	Arria II GX	141.62	1,647	649	-
MD5 Gray Gating	Arria II GX	137.7	1,793	973	-
MD5 Controller	Arria II GX	135.78	712	586	-
MD5(Wang et al., 2010)	Stratix II GX	102.7	1352	462	-
MD5(Jarvinen et al., 2005)	Virtex II	78.30	-	-	1325

Table 4.11 Continued

MD5(Díez et al., 2002)	Virtex II	60.20	-	-	1369
MD5(Deepakumara et al., 2001b)	Virtex V	21	-	-	880

Table 4.11 shows that the highest maximum frequency of the proposed MD5 designs is 154.61 MHz. However, this design employs a large area implementation, with 1,623 combinational ALUTs and 650 registers. The highest frequency of the MD5 design controller is inferior to that of the MD5 binary design; however, it employs a compact area implementation utilising 586 combinational ALUTs. Consequently, Arria II GX can enhance the design's performance and increase its throughput. The design's throughput can be determined using Equation 4.1.

$$Throughput = \frac{(512 \times fMax)}{Number\ of\ cycle\ (Latency)} \quad \text{Equation 4.1}$$

In conclusion, various types of MD5 designs were successfully designed using Verilog. Based on PowerPlay Power Analyzer, the implementation with Gray encoding and clock gating can reduce power consumption. It provides accurate simulation-based power estimation. From this analysis, the dynamic power dissipation of MD5 Gray encoding with clock gating decreased by approximately 20.72% compared to MD5 Binary design. Moreover, MD5 binary encoding can be classified as an efficient implementation of MD5 design, which can provide balance in terms of speed, area, and power.

4.2.2 MD5 Unfolding and Pipelining Design

The top-level RTL architectures of MD5 unfolding design, MD5 unfolding with four-stage pipelining design, and MD5 unfolding with 32-stage pipelining design are presented in the Appendix section to illustrate the structural differences resulting from the applied design techniques. These designs are based on unfolding factor 2 and pipelining architecture. Thus, there are 32 rounds that need to be processed to obtain the 128-bit MD5 hash output.

4.2.2.1 Results of MD5 Unfolding and Pipelining Design

The message input for MD5 unfolding and pipelining design is "**MD5 was proposed by Ron Rivest in 1992.**". This section covers three types of MD5 design: MD5 unfolding design, MD5 unfolding with four-stage pipelining design, and MD5 unfolding with 32-stage pipelining design. The designs were simulated in ModelSim for functional and timing analysis. Figures 4.10 and 4.11 show the functional and timing simulation of MD5 unfolding design. Figures 4.12 and 4.13 illustrate the functional and timing simulation of MD5 unfolding with four-stage pipelining. The last two figures, Figures 4.14 and 4.15, show the functional and timing simulation of MD5 unfolding with 32-stage pipelining. The simulation results indicate that the MD5 unfolding and pipelining design produces the accurate MD5 hash code corresponding to the input message.

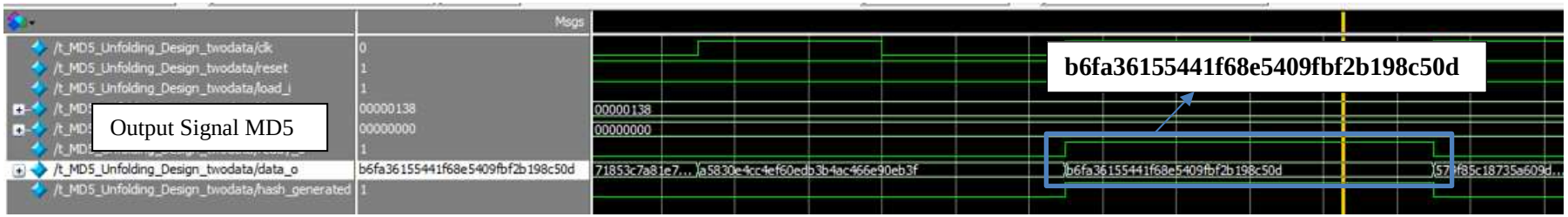


Figure 4.10 Functional Simulation of MD5 Unfolding Design

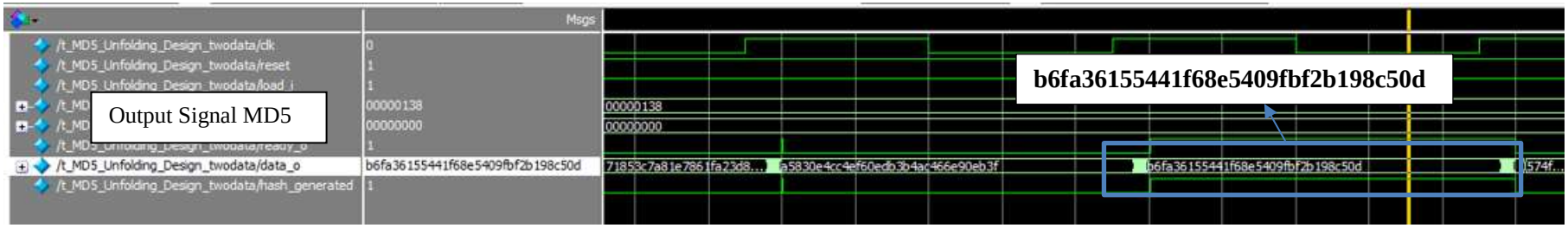


Figure 4.11 Timing Simulation of MD5 Unfolding Design

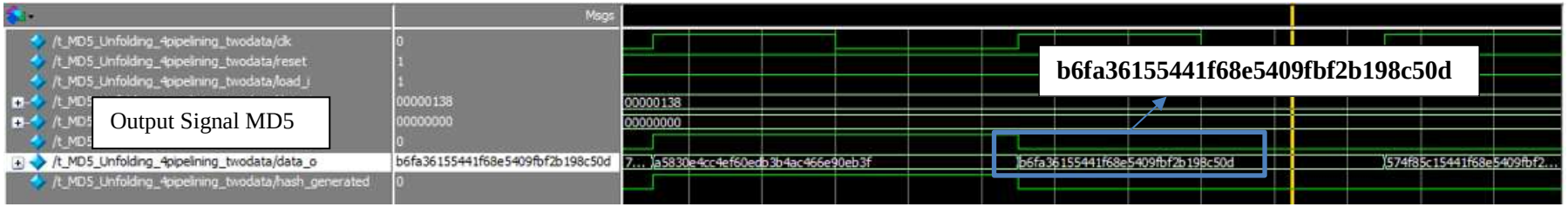


Figure 4.12 Functional Simulation of MD5 Unfolding with Four-stage Pipelining Design



Figure 4.13 Timing Simulation of MD5 Unfolding with Four-stage Pipelining Design

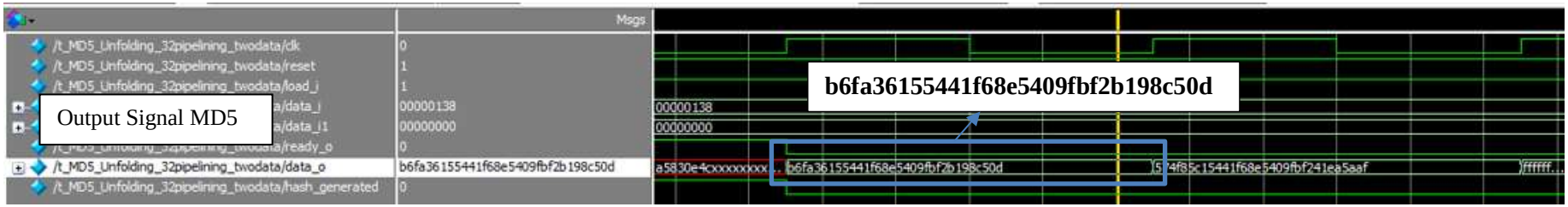


Figure 4.14 Functional Simulation of MD5 Unfolding with 32-stage Pipelining Design

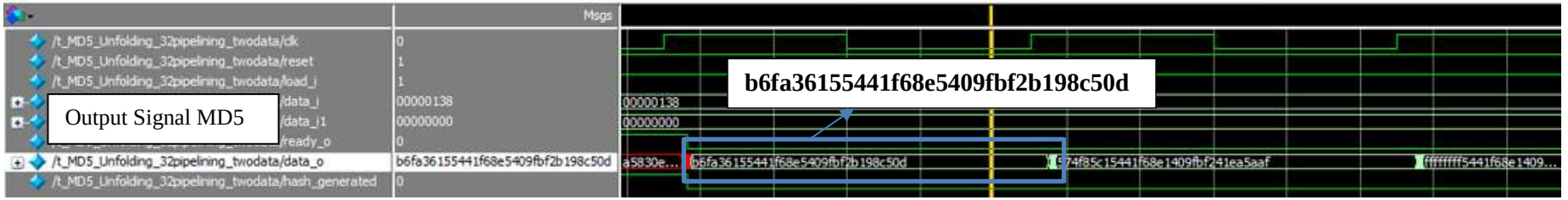


Figure 4.15 Timing Simulation of MD5 Unfolding with 32-stage Pipelining Design

4.2.2.2 Discussion of MD5 Unfolding and Pipelining Design

This study successfully synthesised and implemented MD5 designs using Verilog HDL in Altera Quartus II, with functional and timing verification conducted through simulation in ModelSim. The performance evaluation presented in Table 4.12 compares the proposed MD5 design with previous research, particularly that of Wang et al. (2010). The proposed MD5 design utilises unfolding transformation with 32-stage pipelining, achieving a throughput of 137.972 Gbps, reflecting a 330.7% increase compared to the same design of wang with unfolding with 32-stage pipelining throughput of 32.035 Gbps. This significant performance gain is attributed to the parallelism and pipelining introduced in the datapath, enabling multiple rounds of computation to be executed concurrently within a 4 ns clock period.

The proposed design achieved area efficiency by employing 23,188 combinational ALUTs and 2,715 registers, in contrast to Wang's utilisation of 36,790 ALUTs and 26,758 registers. This indicates a reduction of approximately 36.9% in ALUT usage and an 89.9% reduction in registers, illustrating that the proposed architecture attains high performance while minimising hardware resource consumption. The maximum operating frequency increased from 66.48 MHz in Wang's design to 290.53MHz in my proposed work, resulting in a 337% enhancement in clock speed. The enhancement in frequency, along with optimised state encoding and pipelining techniques, directly resulted in increased throughput while maintaining area constraints suitable for FPGA implementation.2The findings demonstrate that the proposed MD5 architecture achieves high-performance throughput and frequency while maintaining a reduced area footprint relative to previous designs, thereby presenting a more practical and scalable solution for high-speed, low-power cryptographic applications on FPGAs.

In this research, all designs use Arria II GX: EP2AGX45DF29C4 for the synthesis and implementation process. There are several other FPGA family devices, such as Stratix for high-performance FPGAs and Cyclone for low-cost FPGAs. Hence, to obtain a balance between power and performance, the Arria II GX device has been chosen because the Arria series provides a low-cost transceiver for FPGA devices. Moreover, the device also delivers optimal performance and power efficiency. The throughput of the MD5 design increased significantly due to the latency of the design. The latency for MD5 iterative design is 66 cycles. The latencies of the MD5 unfolding design and MD5 unfolding with pipelining are 33.5 and 34.5, respectively. The throughput of the MD5 architecture can be determined using Equation 4.2, where p is the number of stages in pipelining. Table 4.12 indicates that the optimal design of the MD5 hash function regarding speed and area can be achieved by the use of pipelining and unfolding transformations. This method provides small-area implementation, as well as high speed.

$$\text{Throughput} = (512 \times fMax \times p)/\text{Latency} \qquad \text{Equation 4.2}$$

Table 4.12 Comparison of MD5 Designs and Their Implementation

MD5 Design	Device	FMax (MHz)	Throughput (Mbps)	ALUTs/Total Register/Slice
Proposed MD5 Unfolding	Arria II GX: EP2AGX45DF29C4	72.26	1,104.39	1,344/786/-
Proposed MD5 Unfolding 4Pipelining	Arria II GX: EP2AGX45DF29C4	234.05	13,892.56	3,728/980/-
Proposed MD5 Unfolding 32Pipelining	**Arria II GX: EP2AGX45DF29C4**	**290.53**	**137,972.27**	**23,188/2,715/-**
(Fan et al., 2023)	Altera (no device mentioned)	133.6	1068.8	2,334
(Tian et al., 2018)	Altera Cyclone II EP2C35F672C6	50		4,895

Table 4.12 Continued

(Landge & Mishra, 2016)	n.a	Simulation time of 6,900 ns	n.a	n.a
(R. Saeed, 2016)	Xilinx Spartan 3-xc3s1400a-4fg484	-	806.596	-
(Zhong et al., 2016)	n.a	n.a	n.a	n.a
(Noaman, 2013)	Xilinx XC3S1400A	n.a	n.a	-/-/7,499
Wang_32Pipe Wang et al., 2010) (*latency 34)	**Altera Stratix II EP2SGX90FF**	**66.48**	**32035**	**36,790/26,758**
Wang_32Pipe (Wang et al., 2010)	Altera Stratix II EP2SGX90FF	86.32	21,428	36,296/25767
Wang_4Pipe (Wang et al., 2010)	Altera Stratix II EP2SGX90FF	97.95	3,039	5,929/3,484
Wang_ite.(Wang et al., 2010)	Altera Stratix II EP2SGX90FF	102.7	810	1,352/462/-
He and Xue (He & Xue, 2010)	Cyclone II EP2C35F672C6	48.2	4,355	30,134/-/-
(Deepakumara et al., 2001)	Virtex V1000FG680-6	21	165	-/-/880
(Díez et al., 2002)	Virtex 2V3000	60.2	467.3	-/-/1,369
(Jarvinen et al., 2005)	Virtex-II XC2V4000-6	78.3	607	-/-/1,325

A comparison of the performance to area ratio of MD5 designs is presented in Figure 4.16. This graph shows that MD5 unfolding with 32-stage pipelining gives the highest value; thus, this design provides better results than other designs.

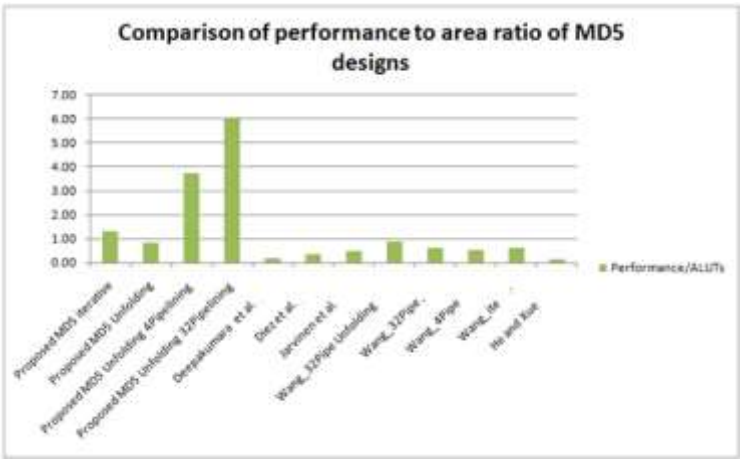
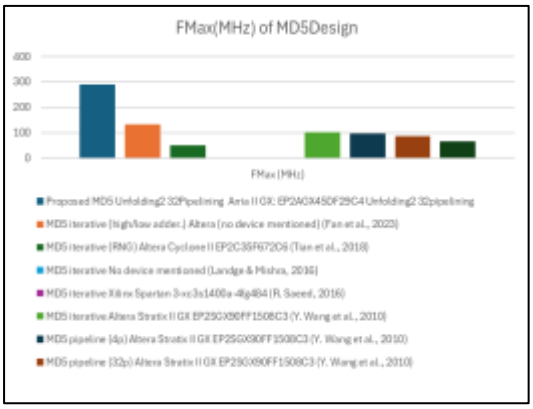


Figure 4.16 Comparison of Performance-to-Area Ratio of MD5 Designs

MD5 designs based on Quartus II Arria II GX: EP2AGX45D F29C4 were successfully synthesised and implemented. According to the findings, the MD5 unfolding that was proposed, which included 32 stages of pipelining, is capable of providing a high-performance MD5 design in terms of speed, area, and throughput. The unfolding transformation gives a high-throughput MD5 design by reducing its latency. Moreover, the pipelined MD5 design has high speed in the maximum frequency. Therefore, an MD5 design combining the unfolding transformation and pipelining can improve the performance significantly. In this study, the maximum frequency (FMax) for MD5 unfolding with 32-stage pipelining is 290.53 MHz with a throughput of 137.97 Gbps. This maximum frequency is approximately 144.3% higher than that of an iterative design.

The proposed MD5 hash function's performance was assessed regarding maximum operating frequency (FMax) and throughput and compared with previously published designs. Figure 4.17 illustrates that the proposed MD5 design, executed through unfolding with 32-stage pipelining on the Altera Arria II GX EP2AGX45DF29C4 FPGA, attained a maximum frequency (FMax) of 290.53 MHz, exceeding all other designs referenced in the literature.



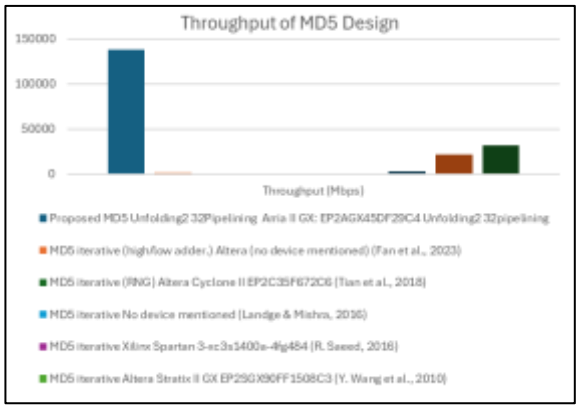


Figure 4.17 FMax of MD5 Design **Figure 4.18** Throughput of MD5 Design

In comparison, Wang et al. (2010) achieved 66.48 MHz on a Stratix II GX device, whereas Fan et al. (2023) reported 133.6 MHz without specifying the FPGA model. This reflects a 337.1% increase in operating frequency relative to Wang's design and a 123.4% increase compared to Fan's work. Figure 4.18 illustrates that the combination of a high clock frequency and deep pipelining yields a throughput of 137,972.27 Mbps, the highest recorded among all evaluated MD5 implementations. In comparison, Wang et al.'s 32-stage pipelined design obtained a throughput of 32.035 Gbps, resulting in a 330.7% enhancement in throughput relative to the MD5 proposed design. These improvements are attributed to the effective integration of unfolding transformation and pipelining, which reduce the critical path delay and allow concurrent execution of MD5 rounds.

4.3 Results and Discussion of SHA-1 Design Architecture

In this research, the message input for the SHA-1 design architecture is "**abc**". Table 4.13 displays the ASCII code corresponding to the input message. A single 1 bit will be added to the end of the message input. Subsequently, it is succeeded by zero bits until the 448th bit. Similar to the MD5 architecture, the final 64 bits include the message length,

but formatted in the big-endian format. Table 4.14 shows the overall 512-bit message for the input message "**abc**" in binary format.

Table 4.13 ASCII Code of Message Input for SHA-1 Design

Bit (MSB-LSB)	**8-bit Binary Hexadecimal: Symbol**		
23–0	01100001 61: a	01100010 62: b	01100011 63:c

Table 4.14 512-bit Message Input

8-bit Binary Hexadecimal: Symbol Bit (MSB….LSB)							
01100001 61: a	01100010 62: b	01100011 63:c	10000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00011000

Table 4.15 shows the input message in the big-endian format, where a, b, and c represent 61, 62, and 63 in hexadecimal, respectively by referring to the ASCII table. A 1-bit message will be added to the end, but the remaining input will be zero, as the input message consists solely of "**abc**". The final 64 bits are allocated for the bit length of the input message "**abc**". The message length is $8 \times 3 = 24$.

Table 4.15 Message in the Big-Endian Format

Message in Hex	**Message in the Big-Endian Format**
61626380	61626380
00000000	00000000
00000000	00000000
00000000	00000000

Table 4.15 Continued

00000000	00000000
00000000	00000000
00000000	00000000
00000000	00000000
00000000	00000000
00000000	00000000
00000000	00000000
00000000	00000000
00000000	00000000
00000000	00000000
00000000, 00000018	00000000,00000018

4.3.1 SHA-1 Iterative Design

The top-level architecture of SHA-1 design is presented in the Appendix section to illustrate the structural. There are eleven modules in this design: sha1_init, dff_sel, mux1, sha_counter, coder_round4, dff_round4, sha_schedule2, KConst, dff_KConst, StepFunction1, and sha1_hash. The design starts with sha1_init module; five buffer initialisations are given to the StepFunction1 module. The main function of sha_counter is to count the process of SHA-1 until it reaches the 80th step of processing. In the StepFunction1 module, there are four rounds with four different non-linear functions. This round executes 20 steps for each round. Furthermore, there are four different constants for every 20 steps. This design only involves the iterative design; all the output hashes will be channelled back to the input by using one block unit module. The Sha_schedule2 module is a message schedule module; the message for t greater or equal to 16 will use the following Equation 4.3.

$$W_t = ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \quad \text{Equation 4.3}$$

4.3.1.1 Results of SHA-1 Iterative Design

The simulation results commence with the initialisation of buffers A, B, C, D, and E. Figure 4.19 illustrates the simulated outcomes for Ain output from t = 1 to t = 10. From this simulation, the message input for SHA-1 is directly from the data input with a fixed constant value. The results for Ain output are tabulated in Table 4.16. The simulation calculation relies on Ain output, as the design's critical path is determined by the value of shift A, non-linear function f(b,c,d), input E , message, and constant value. In this simulation section, the output Ain is divided into eight parts. Thus, the output simulation of Ain output can be seen clearly. Each simulation consists of ten data. By referring to the table of Ain output, the results from ModelSim can be evaluated.

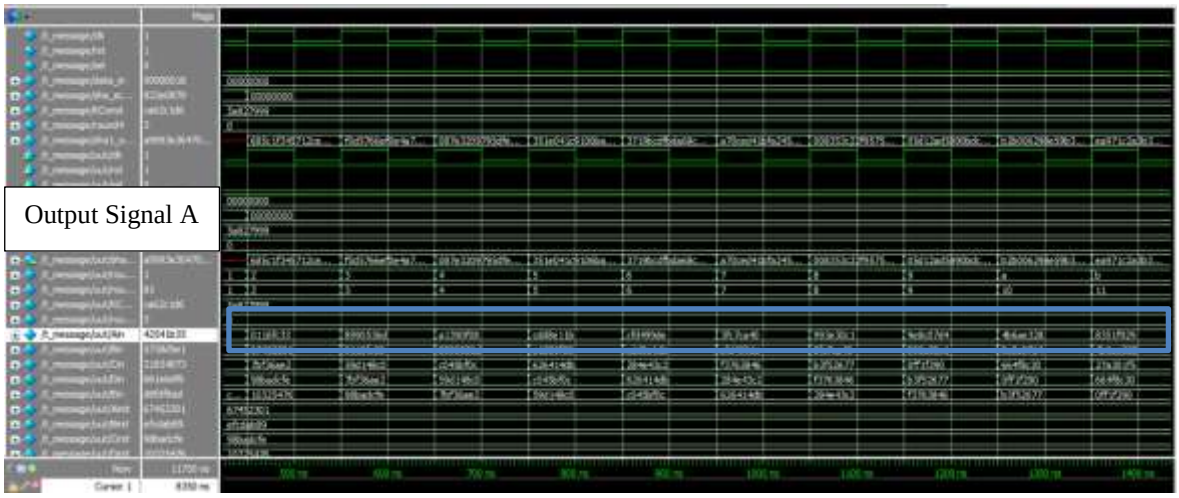


Figure 4.19 Ain(*t*= 1 to *t* = 10)

Table 4.16 Ain Output (*t* = 1 to *t* = 10)

Shift A, Function(b,c,d),message, E, constant)	**Ain Output**
A<<5,func(B,C,D),E,W_1,5a827999	0116fc33
A<<5,func(B,C,D),E,W_2,5a827999	8990536d
A<<5,func(B,C,D),E,W_3,5a827999	a1390f08
A<<5,func(B,C,D),E,W_4,5a827999	cdd8e11b
A<<5,func(B,C,D),E,W_5,5a827999	cfd499de
A<<5,func(B,C,D),E,W_6,5a827999	3fc7ca40
A<<5,func(B,C,D),E,W_7,5a827999	993e30c1
A<<5,func(B,C,D),E,W_8,5a827999	9e8c07d4

Table 4.16 Continued

A<<5,func(B,C,D),E,W_9,5a827999	4b6ae328
A<<5,func(B,C,D),E,W_{10},5a827999	8351f929

The next round is the simulation for output Ain from t = 11 to t = 20. The simulation results of Ain can be derived by assigning varying values to A, B, C, D, and E in the function. The simulation was executed until the 80th round to obtain correct results. The correct results are obtained by employing a similar non-linear function and constant value for the first 20 cycles of the SHA-1 design in the Ain output. The subsequent round of SHA-1 design will commence with a new non-linear function and constant value, following the evaluation of the initial 20 cycles. As mentioned earlier, the input message that is greater or equal to 16 will use Equation 4.3 to calculate the message. Thus, for this simulation, the messages of SHA-1 are obtained from the equation. By using the same non-linear function and constant value, this round will continue until the 40th round with different data for A, B, C, D, and E. The subsequent simulation starts from t = 31 to t = 40, and the process is similar to the previous operation. There are 80 rounds to obtain the final hash code. Finally, the last 10 rounds will be executed, as shown in Figure 4.20, to obtain the SHA-1 hash code. The simulation results from ModelSim are provided by the Ain outputs, as illustrated in Table 4.18. The SHA-1 hash function obtains the correct hash code by assigning varying values to A, B, C, D, and E, as indicated by the simulation results. "**a9993e364706816aba3e25717850c26c9cd0d89d**" is the output of the SHA-1 hash code, with the message input "**abc**". This output is verified in the following section using HashCalc software.

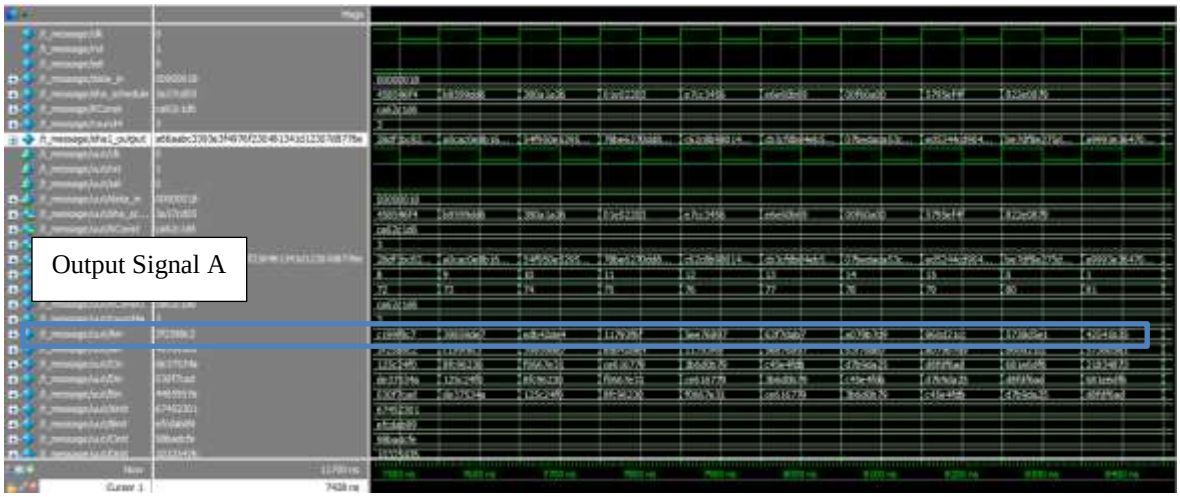


Figure 4.20 Ain (t = 71 to t = 80)

Table 4.17 Ain Output (t = 71 to t = 80)

Shift A, Function(b,c,d), message, E, constant)	Ain Output
A<<5,func(B,C,D),E,W_{71}, ca62c1d6	c199f8c7
A<<5,func(B,C,D),E,W_{72}, ca62c1d6	39859de7
A<<5,func(B,C,D),E,W_{73}, ca62c1d6	edb42de4
A<<5,func(B,C,D),E,W_{74}, ca62c1d6	11793f6f
A<<5,func(B,C,D),E,W_{75}, ca62c1d6	5ee76897
A<<5,func(B,C,D),E,W_{76}, ca62c1d6	63f7dab7
A<<5,func(B,C,D),E,W_{77}, ca62c1d6	a079b7d9
A<<5,func(B,C,D),E,W_{78}, ca62c1d6	860d21cc
A<<5,func(B,C,D),E,W_{79}, ca62c1d6	5738d5e1
A<<5,func(B,C,D),E,W_{80}, ca62c1d6	42541b35

4.3.1.2 Discussion of SHA-1 Iterative Design

Table 4.18 also shows comparative overview of early SHA-1 iterative hardware designs (2002–2009). More recent SHA-1 implementations are discussed separately in next section; these results are presented for legacy comparison purposes. While legacy designs from 2002 to 2009 provide foundational insights, this study prioritizes performance benchmarking against more recent works (2020–2024) to ensure relevance to modern FPGA technologies and synthesis tools. Jarvinen (2004) implemented the SHA-1 hash algorithm in VHDL, which was then directly implemented on an FPGA. Jiang et al. (2009) introduced

four distinct varieties of SHA-1 architecture, including iterative, four-stage pipeline, and 40-stage pipeline, which are capable of both iterative and loop unrolling. The proposed design methodology is based on the inner structure of the main block of the SHA-1 algorithm, which can reduce the design delay. The implementation of the message schedule SHA-1 hash function using CSA to perform addition operations in order to increase the speed of execution was proposed by Miao et al. (2009) and Dai & Zhou (2003). Díez et al. (2002) proposed two unrolled SHA-1 round with partial unrolling architecture for the implementation.

Table 4.18 Comparative Overview of Early SHA-1 Iterative Hardware Designs (2002–2009).

Design	**Device**	**Slices/ ALUTs**	**FMax (MHz)**	**Throughput (Mbps)**
SHA-1 (Proposed)	Virtex-II 2V2000-6	936	117.10	726.73
SHA-1 (Proposed)	Stratix-II GX	530	119.92	744.23
SHA-1 (Lee et al., 2011)	Xilinx XCV1000	NA	84.0	935
SHA-1 (Lee et al., 2011)	Xilinx XCV1000	NA	36.5	1,170
ite (Jiang et al., 2009)	Stratix-II GX	947	253.07	1,561
ite_4p (Jiang et al., 2009)	Stratix-II GX	3576	241.16	5,880
ite_40p (Jiang et al., 2009)	Stratix-II GX	32582	212.40	51,785
u_nr_40p (Jiang et al., 2009)	Stratix-II GX	33764	163.70	76,195
SHA-1(Miao et al., 2009)	Stratix	1131	171.821	727.853
SHA-1 (Michail et al., 2005)	Xilinx V150	NA	55	2,816

Table 4.18 Continued

SHA-1 (Proposed)	Virtex-II 2V2000-6	936	117.10	726.73
SHA-1 (Proposed)	Stratix-II GX	530	119.92	744.23
SHA-1 (Lee et al., 2011)	Xilinx XCV1000	NA	84.0	935
SHA-1 (Lee et al., 2011)	Xilinx XCV1000	NA	36.5	1,170
ite (Jiang et al., 2009)	Stratix-II GX	947	253.07	1,561
ite_4p (Jiang et al., 2009)	Stratix-II GX	3576	241.16	5,880
ite_40p (Jiang et al., 2009)	Stratix-II GX	32582	212.40	51,785
u_nr_40p (Jiang et al., 2009)	Stratix-II GX	33764	163.70	76,195
SHA-1(Miao et al., 2009)	Stratix	1131	171.821	727.853
SHA-1 (Michail et al., 2005)	Xilinx V150	NA	55	2,816
SIG-SHA-1 (Jarvinen, 2004)	Virtex II 2V2000-6	1275	117.5	734
SHA-1 (Dai & Zhou, 2003)	Altera EP1K100-1	1622	43.08	268.99
SHA-1 (Díez et al., 2002)	Virtex-II 2V3000	1550	38.6	899.8

The unfolding transformation technique was proposed for the design method by Lee et al. (2011). Another technique was introduced by Michail et al. (2005), which involved the modification of the expressions A, B, C, D, and E, as well as the operation of the K constant and the message schedule of the SHA-1 algorithm. The proposed SHA-1 algorithm on Xilinx was implemented with a clock frequency of 117.1 MHz and 936 slices in this design. This result is similar with the SHA-1 algorithm implemented by Jarvinen (2004), where the clock

frequency is 117.5 MHz, but the number of slices is 1,275. The implementation of the proposed SHA-1 algorithm on Altera Stratix II GX requires only 530 combinational ALUTs. This result shows that the area implementation is reduced in comparison to other publications on Stratix II (Jiang et al., 2009;Miao et al., 2009). However, the clock frequency of this implementation is only 119.92 MHz. Although some of the SHA-1 designs referenced are from 2002–2009, they are still frequently cited in hardware-based cryptographic literature and are valuable for comprehending the architectural evolution of SHA-1 on early FPGA platforms. However, this study also compares to recent works to ensure its current relevance in unfolding design section of SHA-1.

The architecture of SHA-1 algorithm was successfully synthesised and implemented with both Xilinx Virtex II and Altera Stratix II GX using Verilog HDL. The critical path of the SHA-1 design implementation is exemplified by the maximum clock frequency. In order to achieve a high-performance design, it is necessary to evaluate the area in addition to the speed. Pipelining will give high performance, but this structure tends to increase resource utilisation. Therefore, HDL coding styles also affect the implementation result of the design. In order to optimise the implementation's throughput, it may be necessary to implement an alternative methodology or technique that increases the clock frequency.

4.3.2 SHA-1 Unfolding and Pipelining Design

The top-level architecture of SHA-1 unfolding design is presented in the Appendix section to illustrate the structural. There are ten modules in the SHA-1 unfolding design: sha1_init, dff_sel, mux1, sha_counter, message_unfolding, round4, Kt, func, sha1_hash, and sha1. In this design, message and constant are divided into two parts, namely message1, message2, constant1, and constant2. This is because the unfolding design executes two hash

operations in one cycle. In other words, this design is unfolded by two factors. The rest of the design is similar to the SHA-1 design architecture. This design has the potential to enhance the throughput of the SHA-1 design by reducing the number of cycles required to generate the SHA-1 hash code from 80 cycles to 40 cycles. . The top-level architecture of SHA-1 designs are SHA-1 inner pipelining design, SHA-1 with four-stage pipelining, SHA-1 with 40-stage pipelining, and SHA-1 unfolding with 40-stage pipelining. The structures of the SHA-1 designs are almost similar. The difference of these designs is the modification on unfolding and the number of stages of pipelining.

4.3.2.1 Results of SHA-1 Unfolding Design

In this section, simulation results of the SHA-1 unfolding design are explained. The process of SHA-1 unfolding design operates two hash operations in one cycle. Therefore, there are two messages and two constants that are executed in parallel form during the operation. In other words, in terms of cycles for the SHA-1 unfolding design, the number decreases from 80 cycles to 40 cycles. Figure 4.21 shows the simulation results for signal output A from t = 1 to t = 10. From this simulation, the outputs of A will be executed after two operations of SHA-1 hash function are executed in one cycle. The data for A outputs are given in Table 4.19; the two operations were executed simultaneously. These results provide a complete operation for 20 cycles with the same non-linear function and constant, which decreased to 10 cycles.

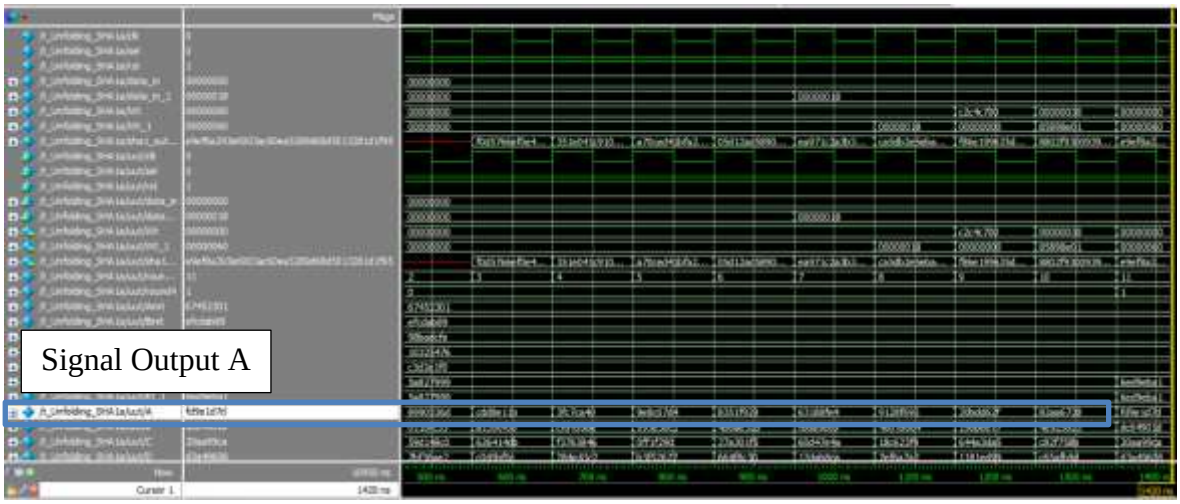


Figure 4.21: A (t = 1 to t = 10)

Table 4.19 A Output (t = 1 to t = 10)

Shift A, Shift Temp,Function (b,c,d),Function(a,b<<30,c),Message1,Message2,D,E,Constant1, Constant2	**A Output**
A<<5,Temp<<5,F(b,c,d),F(a,b<<30,c),W_1, W_2,D,E,5a827999, 5a827999	8990536d
A<<5,Temp<<5,F(b,c,d),F(a,b<<30,c),W_3, W_4,D,E,5a827999, 5a827999	cdd8e11b
A<<5,Temp<<5,F(b,c,d),F(a,b<<30,c),W_5, W_6,D,E,5a827999, 5a827999	3fc7ca40
A<<5,Temp<<5,F(b,c,d),F(a,b<<30,c),W_7, W_8,D,E,5a827999, 5a827999	9e8c07d4
A<<5,Temp<<5,F(b,c,d),F(a,b<<30,c),W_9, W_{10},D,E,5a827999, 5a827999	835189f9
A<<5,Temp<<5,F(b,c,d),F(a,b<<30,c),W_{11}, W_{12},D,E,5a827999, 5a827999	63188fe4
A<<5,Temp<<5,F(b,c,d),F(a,b<<30,c),W_{13}, W_{14},D,E,5a827999, 5a827999	9128f695
A<<5,Temp<<5,F(b,c,d),F(a,b<<30,c),W_{15}, W_{16},D,E,5a827999, 5a827999	20bdd62f
A<<5,Temp<<5,F(b,c,d),F(a,b<<30,c),W_{17}, W_{18},D,E,5a827999, 5a827999	82aa6728
A<<5,Temp<<5,F(b,c,d),F(a,b<<30,c),W_{19}, W_{20},D,E,5a827999, 5a827999	fd9e1d7d

The waveform simulation results for A output continue from t = 11 to t = 20. This operation implements the second non-linear function of the SHA-1 hash function with a distinct constant value. Similar to the previous operation, the number of cycles decreases from 20 cycles to 10 cycles to complete the operation of this function. The third non-linear function is executed together with a new constant value for 20 cycles. As the operation of

the SHA-1 unfolding design runs in parallel form for two hash operations, the number of cycles decreased from 20 cycles to 10 cycles.

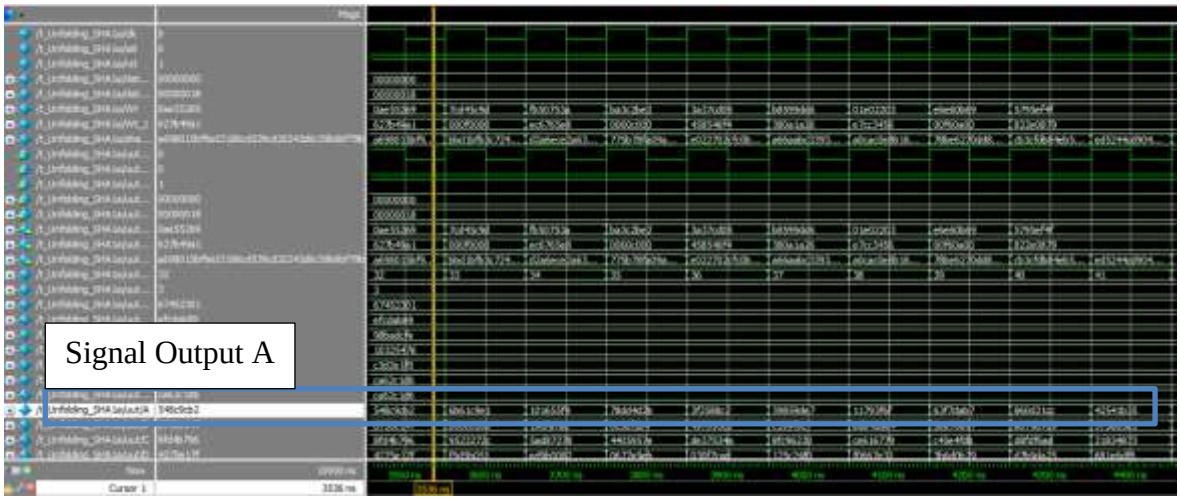


Figure 4.22: A (t = 31 to t = 40)

Previous simulation results of output A executed for t = 21 to t = 30. The output of value A can be obtained by providing the operation with a variety of values for A, B, C, D, and E. Figure 4.22 illustrates the output signal A for the time interval from t=31 to t=40. The value for output A is presented in Table 4.20.

Table 4.20 A Output (t = 31 to t = 40)

Shift A, Shift Temp,Function (b,c,d),Function(a,b<<30,c),Message1,Message2,D,E,Constant1, Constant2	**A Output**
A<<5,Temp<<5,F(b,c,d),F(a,b<<30,c),W_{61}, W_{62},D,E, ca62c1d6, ca62c1d6	548c9cb2
A<<5,Temp<<5,F(b,c,d),F(a,b<<30,c),W_{63}, W_{64},D,E, ca62c1d6, ca62c1d6	6b61c9e1
A<<5,Temp<<5,F(b,c,d),F(a,b<<30,c),W_{65}, W_{66},D,E, ca62c1d6, ca62c1d6	101655f9
A<<5,Temp<<5,F(b,c,d),F(a,b<<30,c),W_{67}, W_{68},D,E, ca62c1d6, ca62c1d6	78dd4d2b
A<<5,Temp<<5,F(b,c,d),F(a,b<<30,c),W_{69}, W_{70},D,E, ca62c1d6, ca62c1d6	3f2588c2
A<<5,Temp<<5,F(b,c,d),F(a,b<<30,c),W_{71}, W_{72},D,E, ca62c1d6, ca62c1d6	39859de7

Table 4.20 Continued

A<<5,Temp<<5,F(b,c,d),F(a,b<<30,c),W_{73}, W_{74},D,E, ca62c1d6, ca62c1d6	11793f6f
A<<5,Temp<<5,F(b,c,d),F(a,b<<30,c),W_{75}, W_{76},D,E, ca62c1d6, ca62c1d6	63f7dab7
A<<5,Temp<<5,F(b,c,d),F(a,b<<30,c),W_{77}, W_{78},D,E, ca62c1d6, ca62c1d6	860d21cc
A<<5,Temp<<5,F(b,c,d),F(a,b<<30,c),W_{79}, W_{80},D,E, ca62c1d6, ca62c1d6	42541b35

Figure 4.23 illustrates the simulation outcomes for the output of the SHA-1 unfolding design following 40 cycles. From this simulation, the output of SHA-1 unfolding design is similar to that of the simulation from SHA-1 design. Thus, both SHA-1 designs and SHA-1 unfolding design are successfully completed and simulated.

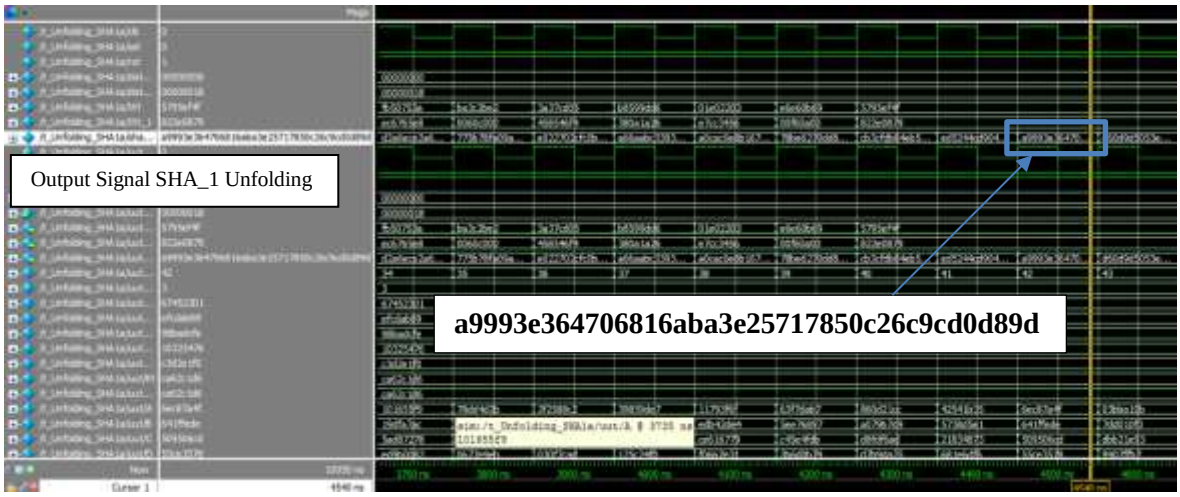


Figure 4.23 Final Simulation Output of SHA-1 Hash Function

Table 4.21 presents the final output of the SHA-1 hash function. The initial inputs of hash values A, B, C, D, and E are incorporated into the final output round of the SHA-1 hash function to derive the final hash. As the outputs of the SHA-1 hash function are expressed in the big-endian format, they stay unchanged in the final transformation.

Table 4.21 Final SHA-1 Hash Function

Register	A	B	C	D	E
Initial Hash Value	67452301	efcdab89	98badcfe	10325476	c3d2e1f0
Round Output	**42541b35**	5738d5e1	21834873	681e6df6	d8fdf6ad
Final Transformation	a9993e36	4706816a	ba3e2571	7850c26c	9cd0d89d
Final Hash	**a9993e364706816aba3e25717850c26c9cd0d89d**				

The HashCalc software was employed to assess the output of the SHA-1 hash function and to confirm the accuracy of the results, as illustrated in Figure 4.24. The graphic indicates that the findings resemble the simulation outputs generated by ModelSim.

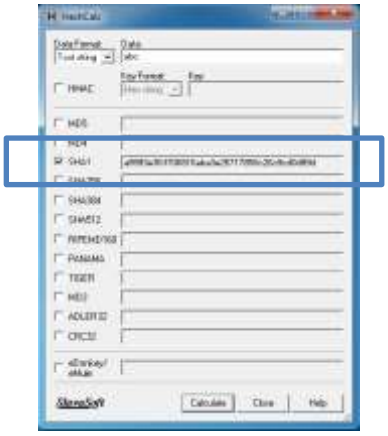


Figure 4.24 Output of SHA-1 Hash Function

4.3.2.2 Results of SHA-1 Unfolding and Pipelining Design

In this section, four different types of SHA-1 designs are simulated in terms of functional and timing simulation. The process of message input is similar to the SHA-1 design to obtain the hash code. The application of unfolding and pipelining techniques to the SHA-1 hash algorithm substantially enhances design performance. Figures 4.25, 4.26, 427, 4.28, 4.29, 4.30, 4.31 and 4.32 present the simulation results for SHA-1 inner pipelining design, SHA-1 with four-stage pipelining, SHA-1 with 40-stage pipelining, and SHA-1

unfolding with 40-stage pipelining in terms of functional and timing simulation, respectively.

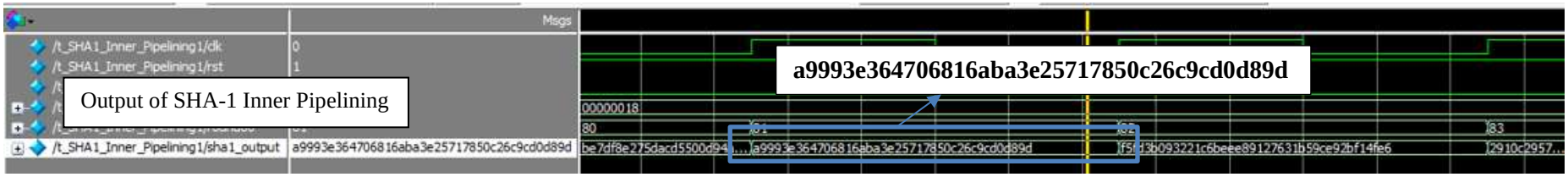


Figure 4.25 Functional Simulation of SHA-1 Inner Pipelining Design

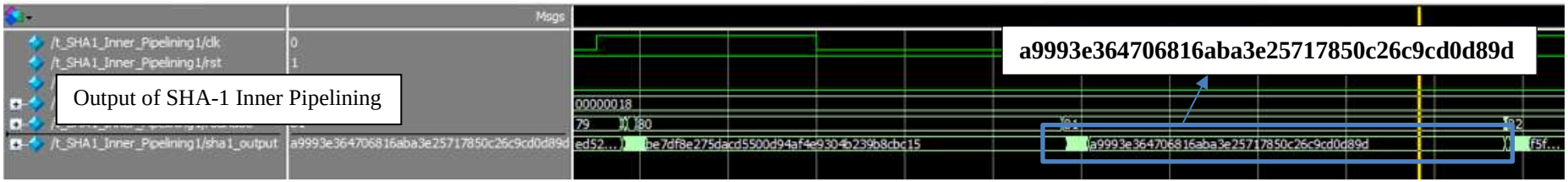


Figure 4.26 Timing Simulation of SHA-1 Inner Pipelining Design

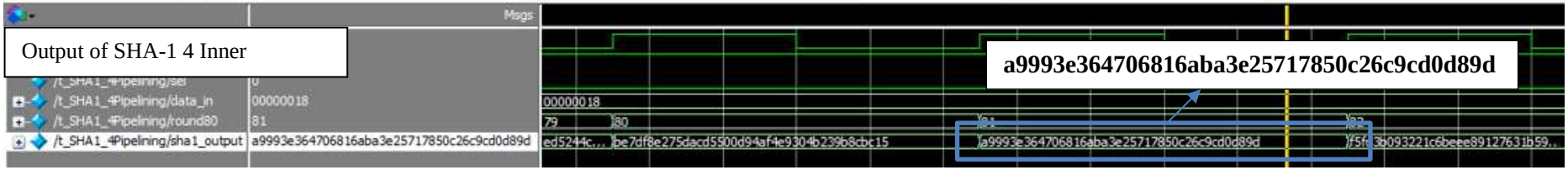


Figure 4.27 Functional Simulation of SHA-1 with Four-stage Pipelining Design

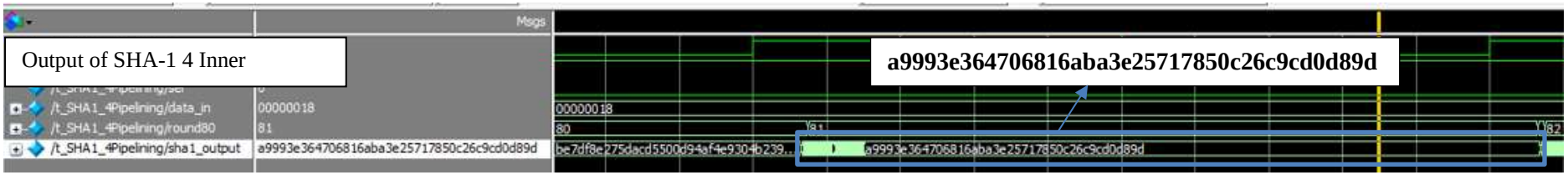


Figure 4.28 Timing Simulation of SHA-1 with Four-stage Pipelining Design

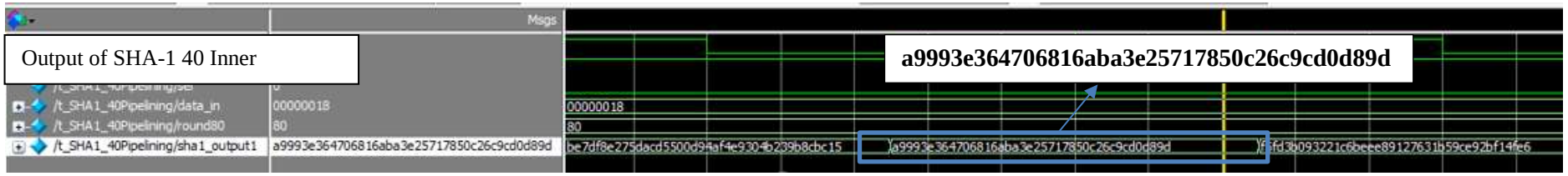


Figure 4.29 Functional Simulation of SHA-1 with 40-stage Pipelining Design

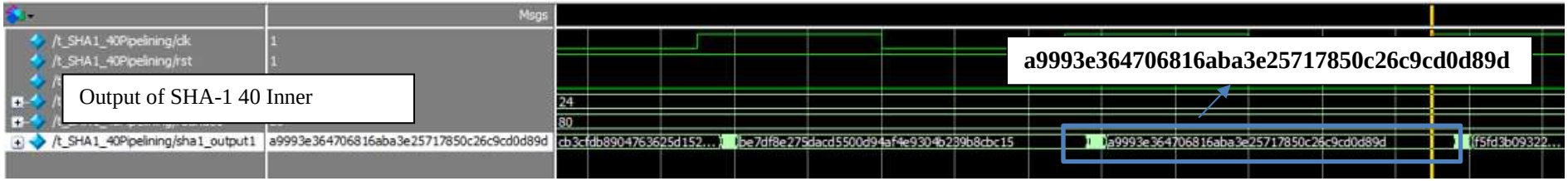


Figure 4.30 Timing Simulation of SHA-1 with 40-stage Pipelining Design

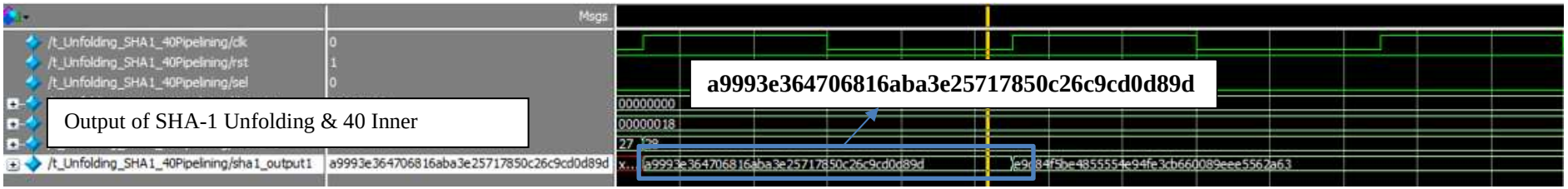


Figure 4.31 Functional Simulation of SHA-1 Unfolding with 40-stage Pipelining Design

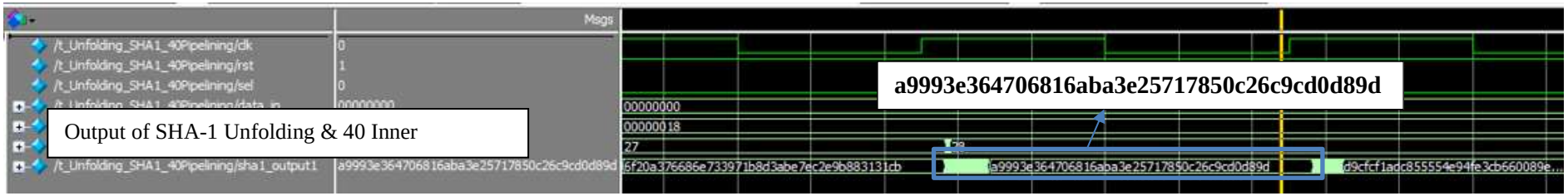


Figure 4.32 Timing Simulation of SHA-1 Unfolding with 40-stage Pipelining Design

4.3.2.3 Discussion of SHA-1 and SHA-1 Unfolding Design

The synthesis and implementation of the SHA-1 algorithms were successfully designed using Verilog HDL based on Altera Quartus II Arria II: EP2AGX45DF29C4. These designs focus on iterative design and unfolding; the one block step function is used for 80 rounds and 40 rounds processes. The designs were verified through simulation using ModelSim-Altera 10.1e, focusing on functional and timing analysis regarding area implementation, maximum frequency, and power analysis of the designs.

The outcomes of the SHA-1 design and SHA-1 Unfolding design are displayed in Table 4.22. The suggested SHA-1 designs and SHA-1 unfolding utilise only 423 and 548 combinational ALUTs, respectively. Furthermore, the total number of registers in the SHA-1 unfolding design increased from 693 to 907. The comparison of area implementation and speed of SHA-1 design depends on the FPGA family devices. The designer must select a suitable device to minimise logic utilisation and enhance design performance. The total power consumption was reduced by 27.1%, as the predicted power dissipation decreased from 625.86 mW in the SHA-1 iterative design to 456.21 mW in the SHA-1 unfolding design. This improvement demonstrates the unfolding approach's efficiency not just in increasing throughput but also in reducing power consumption, making it more suitable for energy-constrained FPGA-based applications. The data indicate that the throughput of the SHA-1 unfolding design increase significantly, achieving a maximum frequency of 174.73 MHz. The throughput of the SHA-1 unfolding design is around 2,236.54 Mbps, surpassing 1,754.88 Mbps of the SHA-1 design. Consequently, the throughput of the design may be determined using Equation 4.4 with a block size of 512 bits.

$$Throughput = \frac{Frequency \times Blocksize}{Latency}$$ Equation 4.4

Table 4.22 FPGA-based SHA-1 Implementation

Device Arria II GX: EP2AGX45DF29C4		
Latency	80 cycles	40 cycles
Design (Proposed)	SHA-1	SHA-1 unfolding (factor 2)
Combinational ALUTs	423	548
Total Register	693	907
Maximum Frequency, *fMax* (MHz)	274.2	174.73
Throughput (Mbps)	1,754.88	2,236.54
Total Thermal Power Dissipation (mW)	625.86	456.21

This research compares the effects of area implementation in terms of combinational ALUTs, logic elements, slices, and the total number of registers using two varieties of FPGAs, Altera and Xilinx, as CAD tools for design implementation. The varying implementations of SHA-1 necessitate an evaluation of design comparisons for area and speed based on the specific target devices. In other words, the designer can choose a device that will provide high-performance implementation. The area implementation of the SHA-1 hash function on various types of FPGA family devices by other researchers is shown in Table 4.23. From this table, the latency in previous papers can be considered as normal SHA-1 operation, where the authors did not mention the value of latency.

Table 4.23 Area Implementations of SHA-1 and SHA-1 Unfolding

SHA-1 Design	Device	Latency	Combinational ALUTs/LE/ Slices	Total Register (Altera)
SHA-1 (proposed)	Arria II GX: EP2AGX45 DF29C4	80	423	693
SHA-1 unfolding (proposed)	**Arria II GX: EP2AGX45 DF29C4**	**40**	**548**	**907**
(Jarvinen, 2004)	Virtex-II 2V2000-6	-	1,275	-
(Kang et al., 2002)	Altera EP20K1000 EBC652-3	-	14,604 (processor)	-
(Miao et al., 2009)	Stratix II: EP2S15F48 4C3	-	1,131	874
(Díez et al., 2002)	Virt(Dai & Zhou, 2003)50	-		
(Dai & Zhou, 2003)	Altera EP1K100Q C208-1	-	1,622	-
(Jiang et al., 2009)	Stratix II GX: EP2SGX90 FF	83	947	782
*** (Jiang et al., 2009) - pipelined**	**Stratix II GX: EP2SGX90 FF**	**44**	**33,764**	**32048**
* (Sklavos et al., 2003)	Xilinx v100ECS14	-	1,578	-

4.23 Continued

(Lee et al., 2006)	Virtex XC2V1000	82	1,446	-
(Lee et al., 2006)	Virtex XC2V1000	42	1,575	-
* (Kim et al., 2012)	Virtex-6	42	1,649	-

* Pipeline

An efficient area implementation is essential for applications requiring low power and optimal hardware utilisation. The proposed SHA-1 designs, implemented on the Arria II GX EP2AGX45DF29C4, demonstrates an area of 548 combinational ALUTs and 907 registers. This is a significant reduction compared to the 33,764 ALUTs and 32,048 registers reported by Jiang et al. (2009) on the Stratix II GX. This indicates a 98.38% reduction in ALUTs and a 97.17% reduction in registers, highlighting the significant area efficiency of the proposed design. The proposed design utilises unfolding transformation to achieve a balance between area and performance, resulting in minimal resource utilisation and a latency of 40 cycles, making it appropriate for compact, high-speed cryptographic applications.

Table 4.24 Maximum Frequency of SHA-1 Design

Design (* pipeline)	**Device**	**FMax (MHz)**	**Throughput (Mbps)**
SHA-1 (proposed)	Arria II GX: EP2AGX45DF29C4	274.15	1,754.56
(Jarvinen, 2004)	Virtex-II 2V2000-6	117.5	734
(Kang et al., 2002)	Altera EP20K1000EBC652-3	18	114

Table 4.24 Continued

(Miao et al., 2009)	Stratix II: EP2S15F484C3	171.821	727.853
(Díez et al., 2002)	Virtex-II 2V3000	38.6	899.8
(Dai & Zhou, 2003)	Altera EP1K100QC208-1	43.08	268.99
(Jiang et al., 2009)	Stratix II GX: EP2SGX90FF	253.07	1561
* (Sklavos et al., 2003)	Xilinx v100ECS14	72	1,700
(Lee et al., 2006)	Virtex XC2V1000	78.1	488

The maximum frequency of the previous SHA-1 designs, and the various types of family devices are compared in Table 4.24. The SHA-1 design that has been proposed offers the highest maximal frequency of 274.15 MHz and a throughput of 1,754.56. While it is possible that reducing ALUTs could result in increased latency because of reduced parallelism or simplified datapaths, this was not observed in the proposed SHA-1 design. Through efficient architectural optimisation and unfolding, the quantity of ALUTs was reduced without sacrificing latency. Latency decreased from 44 cycles (Jiang et al., 2009) to 40 cycles in the proposed architecture. Therefore, the reduction in ALUTs reflects a more efficient logic structure, not a trade-off in performance.

Table 4.25 shows the maximum frequency (FMax) of SHA-1 unfolding. The proposed SHA-1 unfolding design achieves the highest maximum frequency of 174.73 MHz in comparison to other SHA-1 unfolding designs, as indicated by the table. This results in a high throughput of the SHA-1 unfolding design. The throughput from the design by Jiang and Kim is high, owing to the pipelined design. However, this design used large area implementation. Table 4.25 presents the maximum frequency (FMax) and throughput results

for SHA-1 unfolding designs. The proposed SHA-1 unfolding (factor 2) design attains a maximum frequency of 174.73 MHz, indicating a 6.74% enhancement compared to the 163.70 MHz documented by Jiang and Kim. Although the frequency increase is modest, it contributes meaningfully to the overall system performance.As can be seen from Table 4.24, the same authors used many combinational ALUTs of approximately 33,764 and 1,649, compared to this SHA-1 unfolding design that only used 548 combinational ALUTs.

Table 4.25 Maximum Frequency of SHA-1 Unfolding

Design (*pipeline)	**Device**	**FMax (MHz)**	**Throughput (Mbps)**
SHA-1 (proposed)	Arria II GX: EP2AGX45DF29C4	174.73	22,36.54
* (Jiang et al., 2009)	Stratix II GX: EP2SGX90FF	163.70	76,195
(Lee et al., 2006)	Virtex XC2V1000	56.8	693
(Lee et al., 2011)	Virtex XC2V1000	84.0	935
* (Kim et al., 2012)	Virtex-6	150.7	7350

The proposed SHA-1 and SHA-1 unfolding designs have been assessed in comparison to existing published implementations, with a particular emphasis on their maximum frequency (FMax). Despite this, there is no standard benchmark for FPGA hardware implementations of SHA-1 because different device models, synthesis tools, and optimisation techniques differ. Frequency performance is not absolute and varies on various design-dependent aspects, including: target FPGA family and speed grade, clock limitations, RTL-level optimisations, synthesis and placement schemes. Therefore, while this study does

not refer to a standardized benchmark (as no fixed benchmark exists for SHA-1 on FPGA), it uses published, comparable designs as baseline references to demonstrate the effectiveness of the proposed techniques.

This section discusses the outcomes of SHA-1 design and SHA-1 unfolding design regarding maximum frequency and area implementation. Implementation on Arria II GX gives small area implementation. From Figure 4.33, the area implementation for both proposed SHA-1 and SHA-1 unfolding is almost identical. However, the SHA-1 unfolding uses more logic utilisation in terms of combinational ALUTs and total number of registers compared to the SHA-1 design.

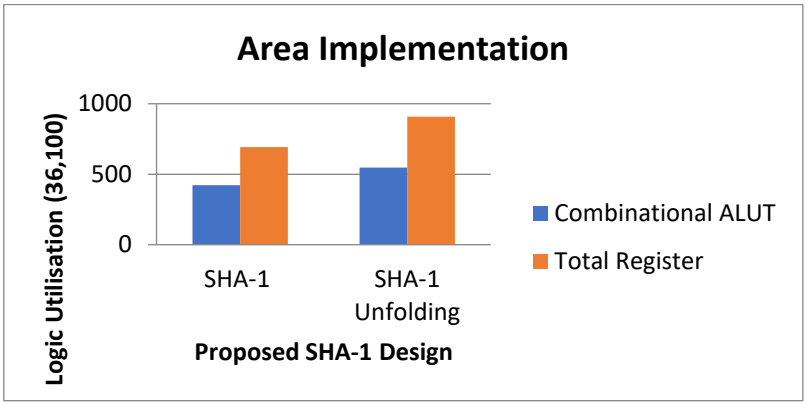


Figure 4.33 Area Implementations for Both Proposed SHA-1 Designs

The maximum frequency and throughput for the proposed designs of SHA-1 and SHA-1 unfolding are illustrated in Figure 4.34. Maximum frequency is one of the criteria that need to be considered in designing a high-performance design; the critical path of the design needs to be evaluated. The proposed SHA-1 design has a slightly higher frequency compared to the SHA-1 unfolding, despite the latter's frequency being lower than that of the SHA-1 design. Furthermore, from this graph, the SHA-1 unfolding design gives a high throughput.

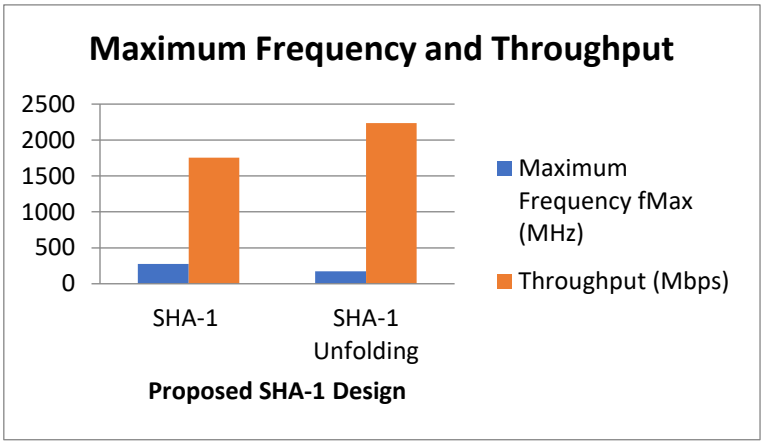


Figure 4.34 Maximum Frequency and Throughput for Both Proposed SHA-1 Designs

As mentioned earlier, Altera provides the most accurate and comprehensive PLD power analysis and optimisation tools in the industry. In this research, Altera Quartus II PowerPlay Power Analyzer is used with input from the Verilog SHA-1 design and signal activities to obtain the power analysis report. The total thermal power dissipation for both the proposed SHA-1 designs and the SHA-1 unfolding design is depicted in Figure 4.35. From this figure, the total estimated power dissipation of the SHA-1 unfolding design decreased significantly compared to the SHA-1 design.

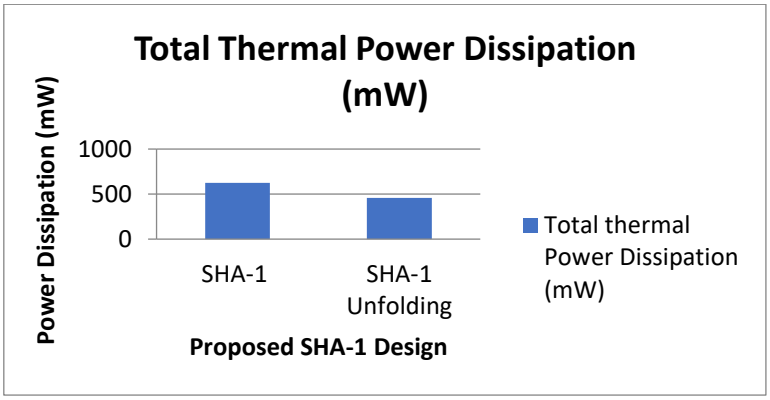


Figure 4.35 Total Thermal Power Dissipation for Both Proposed SHA-1 Designs

Verilog HDL effectively synthesises and implements the SHA-1 and SHA-1 unfolding designs for Altera Arria II: EP2AGX45DF29C4. Meanwhile, in area utilisation, the number of combinational ALUTs and total number of registers are 548 and

907, respectively, and the maximum frequency of the design is 174.73 MHz. The maximum frequency of the SHA-1design implementation illustrates the critical path of the design. To obtain a high-performance design, both speed and area implementation should also be considered. There are also more approaches or tools one may use to increase the design's throughput and maximum frequency. High performance with efficient design incorporates considerations of small area implementation, high maximum frequency, and small estimated power consumption; this in turn will result in a high design throughput.

i. SHA-1 Unfolding and Pipelining Design

There are five proposed SHA-1 designs to be developed and evaluated as follows: (1) iterative SHA-1, (2) inner-round pipelining SHA-1, (3) four-stage pipelining SHA-1, (4) 40-stage pipelining SHA-1, and (5) 40-stage pipelining with unfolding transformation factor 2 SHA-1. These architectures were designed using Verilog code and synthesised with both Altera and Xilinx tools. The primary aim of this study is to achieve high-speed design. Therefore, four and forty stages of pipelining SHA-1 design were proposed. Incorporating registers at various stages of the round can enhance the design's maximum frequency. Four registers were utilised for A, B, C, D, and E in the compression function with 20 rounds per stage, while forty registers were employed for A, B, C, D, and E in the compression function with two rounds each stage. The unfolding transformation technique has been developed to enhance the throughput of the SHA-1 architecture. This study presents an unfolding mechanism designed based on factor 2. Figure 4.36 illustrates the structure of compression function of SHA-1 algorithm using the unfolding transformation technique. From this figure, there are two non-linear functions (i.e., Func and $Func_{t+1}$) with three different inputs. Furthermore, eight addition operations were performed in parallel form in this architecture. It consists of four shifters: circular left shift of A by 30, circular left shift of B by 30, circular

left shift of A by 5, and circular left shift of Temp by 5. This procedure decreases the processing cycles of the SHA-1 design from 80 to 40, with two hash operations performed per cycle. Small number of latencies can increase the throughput of SHA-1 design. Hence, efficient and high-performance design can be obtained. The outputs of SHA-1 unfolding algorithms are shown in the following equation. $ROTL^{a}(b)$ represents circular left shift or left rotation operation of b by a position to the left, and $func_t(p,q,r)$ represents non-linear function at time t for three different inputs of p,q and r.

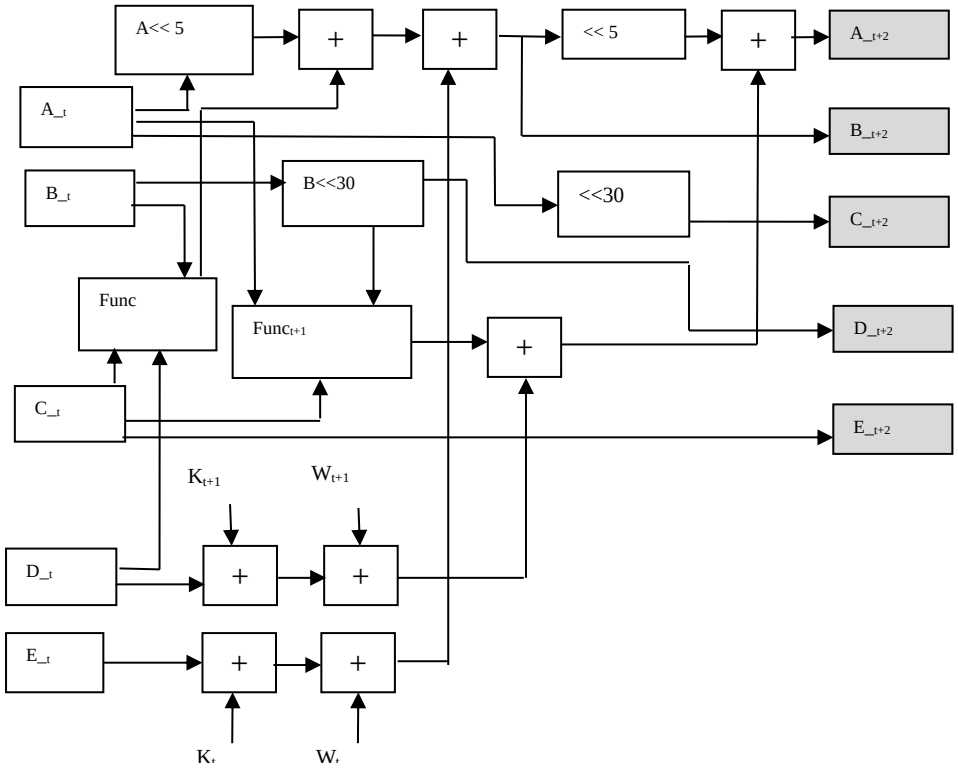


Figure 4.36 Unfolding Transformation of SHA-1 Algorithm Compression Function

Figure 4.37 shows the architecture of four-stage pipelining with five different inputs. This figure can be used to construct the architecture of 40-stage pipelining with unfolding factor 2, where each stage consists of two rounds. Modifications have been implemented regarding the input and output of the modules to achieve the output of the specified design.

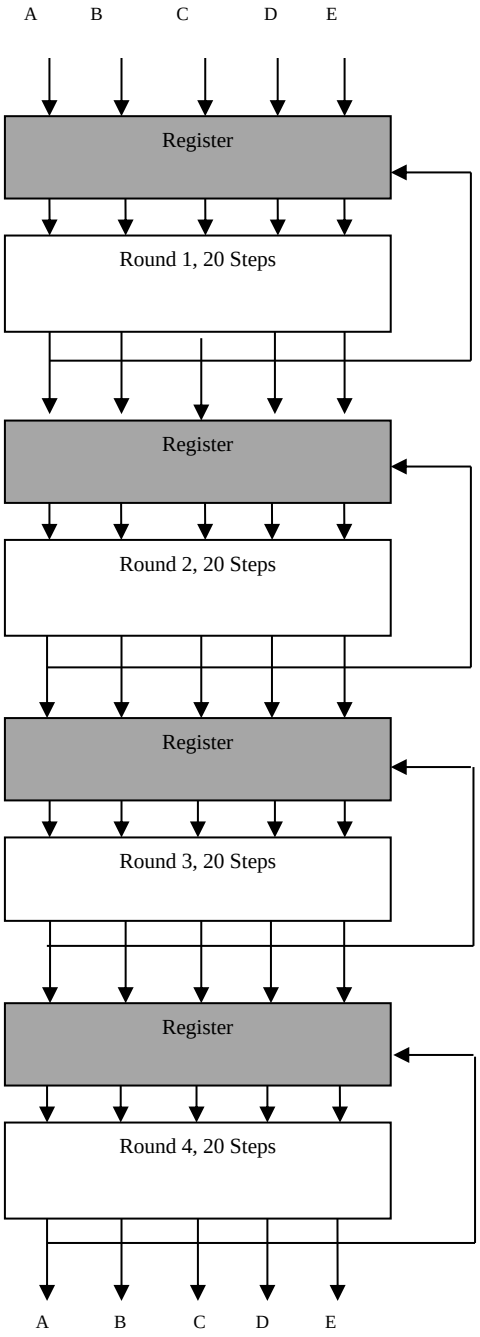


Figure 4.37 Four Stages of SHA-1 Algorithm Architecture

This design starts with buffer initialisation, which is _init module. In this module, there are five different fixed values of buffer initialisation values. Eighty rounds of counter module need to be designed to execute schedule module. Schedule module is a module that executes message input to step function. The message input for the first 15 data input of SHA-1 comes from 512-bit message input, which has been divided into 15 blocks of 32-bit

message input. In this schedule module, the message after 15 rounds is calculated using the formula given in Equation 4.3.

The five SHA-1 designs such as (1) iterative, (2) inner pipelining, (3) four-stage pipelining, (4) 40-stage pipelining, and (5) 40-stage pipelining with an unfolding factor of two were successfully implemented using Verilog code. The designs were synthesised and implemented utilising Altera Quartus II 15.0 and Xilinx ISE Design Suite 14.7. The proposed SHA-1 designs were simulated using ModelSim for both functional and timing simulation to verify the correctness of the design. The TimeQuest Timing Analyzer was used to optimise the design by giving appropriate clock constraint of SDC file to meet the timing requirement. Tables 4.37 and 4.38 show the five proposed SHA-1 design results obtained on Arria II GX and Virtex 5 XC5VLX50T after synthesis and implementation. The throughput of the SHA-1 design can be determined using Equation 4.5. The number of pipeline stages in the SHA-1 design is denoted by p.

$$Throughput = \frac{512 \times Maximum\ Frequency \times p}{Latency} \qquad \text{Equation 4.5}$$

This design is unique in that it employs a combination method that considers the number of pipelining stages and unfolding, as illustrated in Tables 4.26 and 4.27. The result of the 40-stage pipelining for unfolding SHA-1 is a high throughput of approximately 150.27 Gbps on Arria II GX, as shown in Table 4.26. The maximum frequency of this design is 458.60 MHz, which is a significant improvement over Xilinx Virtex 5 XC5VLX50T, as presented in Table 4.27. As the latency for this unfolding SHA-1 design decreased from 80 cycles to 40 cycles, the throughput of this design increased up to 223.62 Gbps. Besides, the total register of the SHA-1 unfolding transformation with 40-stage pipelining decreased for both Altera and Xilinx implementation.

Table 4.26 Synthesis and Implementation based on Altera Arria II GX

Design	Pipeline	FMax (MHz)	ALUTs	Total Registers	Latency	Throughput (Mbps)
SHA1_ite	1	279.64	402	720	82	1,746.04
SHA1_inner	1	281.53	411	681	82	1,757.85
SHA1_4	4	316.76	976	715	82	7,911.27
SHA1_40	40	321.54	6715	3229	82	80,306.57
SHA1_ Unfold40	40	308.17	9799	718	42	150,269.56

Table 4.27 Synthesis and Implementation based on Xilinx Virtex5 XC5VLX50T

Design	Pipeline	FMax (MHz)	Slice Register	Slice LUT	Latency	Throughput (Mbps)
SHA1_ite	1	233.899	720	897	82	1,460.44
SHA1_inner	1	232.918	681	950	82	1,454.32
SHA1_4	4	430.024	974	1332	82	10,740.11
SHA1_40	40	427.546	5953	6465	82	106,782.22
SHA1_Unfold 40	40	458.593	3448	11994	42	223,618.68

In conclusion, the five SHA-1 designs of SHA-1 iterative, SHA-1 inner pipelining, SHA-1 Four-stage pipelining, SHA-1 40-stage pipelining, and SHA-1 40-stage pipelining with unfolding factor 2 transformation were designed using both Altera Quartus II and Xilinx ISE design suite 14.7 and subsequently evaluated. The simulation results indicated that the SHA-1 design achieved a higher speed and throughput owing to the 40-stage pipelining with

unfolding factor 2 transformation. By using the techniques of pipelining and unfolding, the throughput increased up to 159.3 Gbps on Arria II GX and 223.6 Gbps on Virtex 5 XC5VLX50T. The maximum frequency of unfolding SHA-1 40-stage pipelining was 308.17 MHz and 458.6 MHz on Arria II GX and Virtex 5 XC5VLX50T, respectively. This design only used 9,799 combinational ALUTs and 718 total registers on Arria II GX. For Virtex 5 XC5VLX50T, the usage of slice LUT and slice register was about 11,994 and 3,448, respectively. Hence, by using an appropriate FPGA device, the high frequency and small area implementation can be obtained. This proposed SHA-1 unfolding and pipelning design demonstrates better performance compared with traditional designs. Consequently, the proposed method has the potential to enhance the effectiveness of the SHA-1 design, which is beneficial for the development of security systems and other hash functions.

Although the FMax of 174.73 MHz in the proposed SHA-1 unfolding design is not the greatest among the designs studied, the enhancement is attributed to the architectural optimisation realised through unfolding and pipelining approaches. This methodology allows the design to achieve a throughput about 12 times greater than the baseline iterative SHA-1 implementation, while employing merely 548 ALUTs. This signifies a considerable benefit for area efficiency, especially in contrast to bigger pipelined architectures that utilise far more FPGA resources. The proposed design exhibits an optimal balance of speed, throughput, and space, rendering it exceptionally efficient and appropriate for resource-limited applications. In conclusion, while 174.73 MHz is not the absolute maximum frequency, it is a significant achievement when evaluating performance per area. The efficiency achieved via architectural optimisation, rather than merely speed, is the strength of the proposed unfolding design.

4.4 Results and Discussion of RIPEMD-160 Design Architecture

In this design, several proposed RIPEMD-160 algorithms were designed using Verilog HDL codes, including iterative, pipelined, unfolding RIPEMD-160 with factor 2, and unfolding RIPEMD-160 with factor four. The codes were compiled to check for syntax error. The register-transfer level (RTL) codes are synthesised during logic synthesis. Logic synthesis is a translation of RTL description into gate level netlist. Logic optimisation can be applied to the design by giving clock constraint and choosing the appropriate FPGA device for technology mapping. Then, the optimised designs are obtained after this process. The testbench files are used to test and verify the simulation results of these designs. Prior to testing the designs, the input message needs to be padded. Initially, a single 1 bit of logic 1 is inserted after the message input is received, followed by logic 0. The final 64 bits are designated for the message's length.

The input for RIPEMD-160 is "**abc**". The message requires padding to generate the RIPEMD-160 hash code. The RIPEMD-160 hash function operates on communications in the little-endian format. Thus, the hexadecimal format must be converted into the little-endian format so that the hash function can process the message. The message padding commences at the end of the message with the addition of a single 1 bit, followed by zero bit until the 448th bit. Similar to other hash algorithms, the final 64 bits represent the message length. Table 4.28 displays the ASCII code for the input "**abc**" in hexadecimal format.

Table 4.28 ASCII Code of Message Input RIPEMD-160 Design

Bit (MSB-LSB)	**8-bit Binary Hexadecimal: Symbol**		
23–0	01100001 61: a	01100010 62: b	01100011 63:c

Table 4.28 shows the input message in the little-endian format, where a, b, and c represent 61, 62, and 63 in hexadecimal format, respectively, according to the ASCII table. A 1-bit message will be added to the end of the message, but the remaining input will be zero, as the input message consists solely of "**abc**". The final 64 bits are allocated for the bit length and the input message "**abc**". The message length is 8 ×3 = 24. The total message has 512 bits, as shown in Table 4.29. The component messages are segmented into 16 blocks, each containing 32-bit message words. As RIPEMD-160 operates in the little-endian format, the messages must be converted to the little-endian format, as illustrated in Table 4.30.

Table 4.29 512-bit Message Input

8-bit Binary **Hexadecimal: Symbol** **Bit (MSB…….LSB)**							
10000000 80:	01100011 63:c	01100010 62:b	01100001 61: a	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00011000	00000000	00000000

Table 4.30 Message in the Little-Endian Format

Message in Hex	**Message in the Little-Endian Format**
61626380	80636261
00000000	00000000
00000000	00000000
00000000	00000000
00000000	00000000
00000000	00000000
00000000	00000000
00000000	00000000
00000000	00000000

Table 4.30 Continued

00000000	00000000
00000000	00000000
00000000	00000000
00000000	00000000
00000000	00000000
00000000, 00000018	00000018,00000000

4.4.1 RIPEMD-160 Iterative and Pipelining Design

The top-level architecture of RIPEMD-160 design is presented in the Appendix section to illustrate the structural. There are eight modules in this architecture: counter, coder, Kconst_ripemd, message_ripemd, mux1, function_ripemd_parallel, hash, and hash output. In this design, two parallel designs are executed simultaneously for both left and right lines. As mentioned earlier, the non-linear functions for the left and right lines are executed in the opposite order. The final output for RIPEMD-160 is 160 bits. The last block module employs registers to enhance the design's performance.

The top-level architecture of pipelined RIPEMD-160 design is presented in the Appendix section to illustrate the structural. There are 23 modules of pipelined RIPEMD-160 design: counter, coder, Kconst_ripemd, message_ripemd, mux1, XK_1, XK_2, selector_XK1, selector_XK2, sel_PreComp11, Pre_Computation1, sel_PreComp1, Pre_Computation2, sel_PreComp2, Pre_Computation3, sel_PreComp3, Pre_Computation4, sel_PreComp4, Pre_Computation5, Final_Selector, Final_Calculation, hash, and hash_output. The pipelined RIPEMD-160 is divided into five different parts because there are five different non-linear functions in this design. Each round consists of 16 step numbers with different values of shift and message from the left and the right.

4.4.1.1 Results of RIPEMD-160 Iterative Design

Both RIPEMD-160 and pipelined RIPEMD-160 designs use the same message to obtain the hash code. The following section contains the simulation results for both designs. These simulation results are separated into two sections. Figure 4.38 shows the simulation results for both B_in0 and B_in1 from t = 1 to t = 16. From this simulation, the results for B_in0 and B_in1 are similar to the data tabulated in Table 4.31.

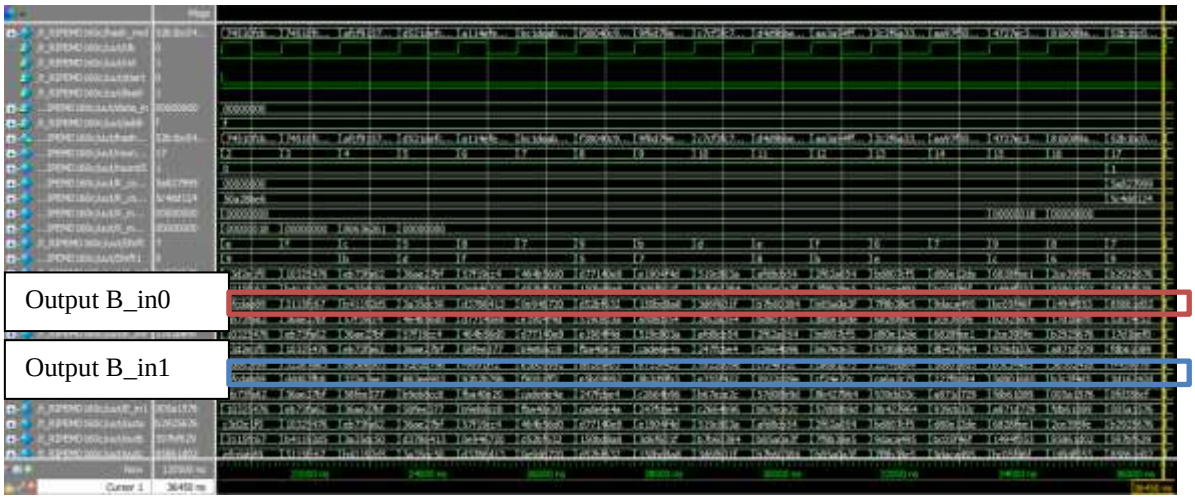


Figure 4.38 Output B_in0 and B_in1 for $\mathbf{1 \leq t \leq 16}$

The output for both B_in0 and B_in1 is based on constant, message input, non-linear function of data B, C, and D, and value of A. All these values will be shifted with different values of shift for each round. Then, the results will be added to the E value.

Table 4.31 Output B_in0 and B_in1 for $\mathbf{1 \leq t \leq 16}$

rol s(t) (A, f(b,c,d),Message[t],Constant), E	**Output B_in0**	**rol s(t) (A, f(b,c,d),Message[t],Constant), E**	**Output B_in1**
$S_{11}(A, f_{1(b,c,d)}, M_0$,00000000),E	3115fc67	$S'_8(A, f_{5(b,c,d)}, M'_5$,50a28be6),E	ddd63fb8
$S_4(A, f_{1(b,c,d)}, M_1$,00000000),E	b41192d5	$S'_9(A, f_{5(b,c,d)}, M'_{14}$,50a28be6),E	322e7ae3
$S_{15}(A, f_{1(b,c,d)}, M_2$,00000000),E	3a35dc50	$S'_9(A, f_{5(b,c,d)}, M'_7$,50a28be6),E	883ee903
$S_{12}(A, f_{1(b,c,d)}, M_3$,00000000),E	d3786413	$S'_{11}(A, f_{5(b,c,d)}, M'_0$,50a28be6),E	92b2b79b
$S_5(A, f_{1(b,c,d)}, M_4$,00000000),E	0e946720	$S'_{13}(A, f_{5(b,c,d)}, M'_9$,50a28be6),E	f9091ff2
$S_8(A, f_{1(b,c,d)}, M_5$,00000000),E	d52bf632	$S'_{15}(A, f_{5(b,c,d)}, M'_2$,50a28be6),E	e5b09992
$S_7(A, f_{1(b,c,d)}, M_6$,00000000),E	150bd8a8	$S'_{15}(A, f_{5(b,c,d)}, M'_{11}$,50a28be6),E	8b2d9fb3

Table 4.31 Continued

$S_9(A, f_{1(b,c,d)}, M_7$,00000000),E	3d6f601f	$S'_5(A, f_{5(b,c,d)}, M'_4$,50a28be6),E	e755f422
$S_{11}(A, f_{1(b,c,d)}, M_8$,00000000),E	b7b60384	$S'_7(A, f_{5(b,c,d)}, M'_{13}$,50a28be6),E	5922d09e
$S_{13}(A, f_{1(b,c,d)}, M_9$,00000000),E	b85a0a3f	$S'_7(A, f_{5(b,c,d)}, M'_6$,50a28be6),E	cf24e72c
$S_{14}(A, f_{1(b,c,d)}, M_{10}$,00000000),E	7f8b38e5	$S'_8(A, f_{5(b,c,d)}, M'_{15}$,50a28be6),E	ca6a1c75
$S_{15}(A, f_{1(b,c,d)}, M_{11}$,00000000),E	9daca495	$S'_{11}(A, f_{5(b,c,d)}, M'_8$,50a28be6),E	227f6d84
$S_6(A, f_{1(b,c,d)}, M_{12}$,00000000),E	bc05f46f	$S'_{14}(A, f_{5(b,c,d)}, M'_1$,50a28be6),E	5d801685
$S_7(A, f_{1(b,c,d)}, M_{13}$,00000000),E	1494f053	$S'_{14}(A, f_{5(b,c,d)}, M'_{10}$,50a28be6),E	b3c3f4d5
$S_9(A, f_{1(b,c,d)}, M_{14}$,00000000),E	85861d02	$S'_{12}(A, f_{5(b,c,d)}, M'_3$,50a28be6),E	3d16242d
$S_8(A, f_{1(b,c,d)}, M_{15}$,00000000),E	597bf629	$S'_6(A, f_{5(b,c,d)}, M'_{12}$,50a28be6),E	ff459078
$S_{12}(A, f_{1(b,c,d)}, M_3$,00000000),E	d3786413	$S'_{11}(A, f_{5(b,c,d)}, M'_0$,50a28be6),E	92b2b79b

The next round starts from t = 17 to t = 32 for both B_in0 and B_in1. By using different values of constant and specific message inputs and shift values, the output of B_in0 and B_in1 can be obtained. These results are derived from the left and right lines of RIPEMD-160. The operation of RIPEMD-160 continues from t = 49 to t = 64, and it will be processed for a total of 16 cycles. The left and right lines of RIPEMD-160 will be subjected to different non-linear functions. Then, the simulation results will continue for B_in0 and B_in1 for t = 65 and t = 80. The outputs of B_in0 and B_in1 are executed in 16 rounds to obtain the final hash code with different constants and non-linear functions for both lines.

Lastly, as shown by Equation 4.6, the final hash code can be derived by adding the initial values a, b, c, d, and e from the left and right to the final round of RIPEMD-160. Figure 4.39 shows the final output of RIPEMD-160 design with data output "**8eb208f7e05d987a9b044a8e98c6b087f15a0bfc**" with message input "**abc**".

h0_out = init_value[127:96] + left_outc + right_outd

h1_out = init_value[95:64] + left_outd + right_oute

h2_out = init_value[63:32] + left_oute + right_outa Equation 4.6

h3_out = init_value[31:0] + left_outa + right_outb

h4_out = init_value[159:128] + left_outb + right_outc

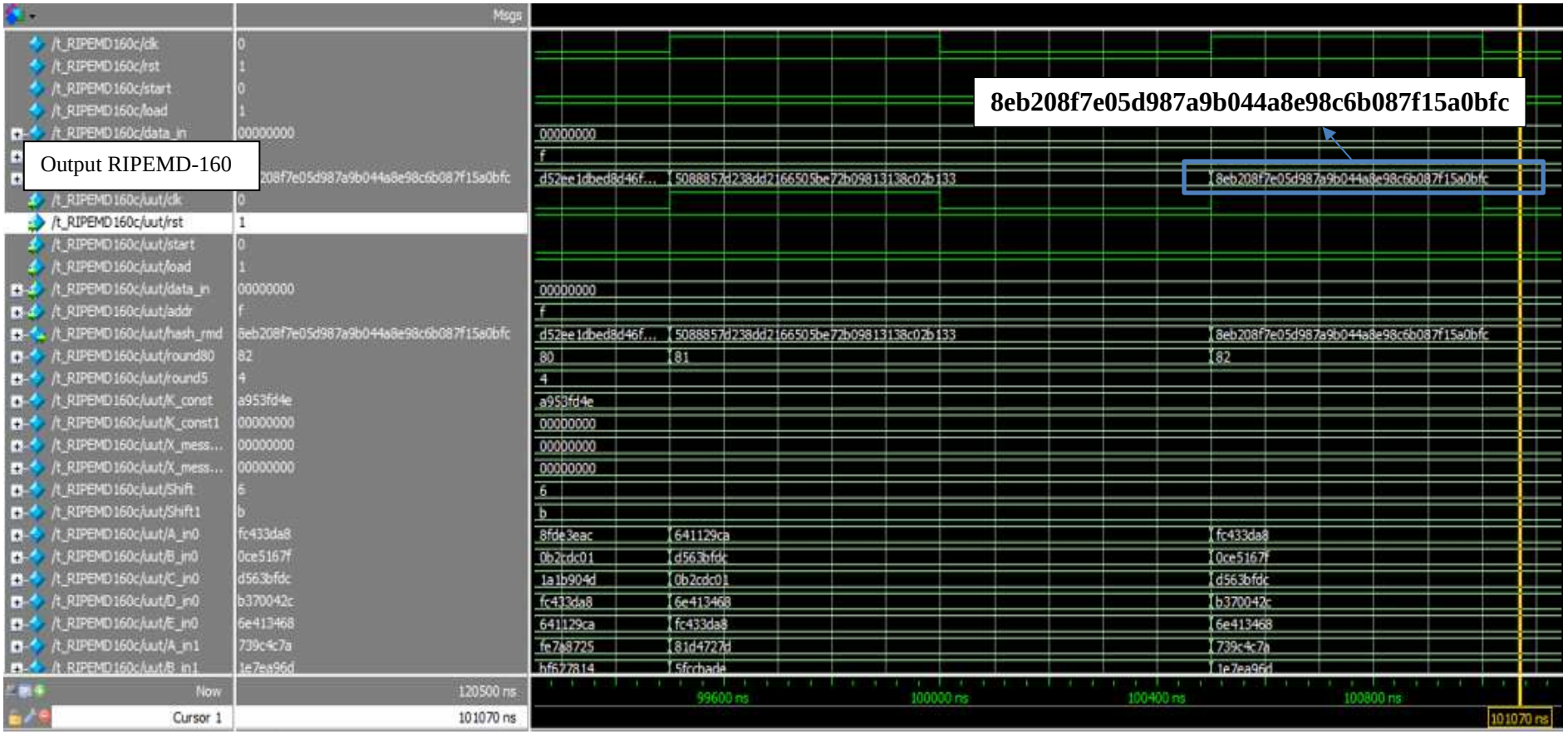


Figure 4.39 Final Output of RIPEMD-160 Design

4.4.1.2 Results of RIPEMD-160 Pipelining Design

In this section, the simulation results also focus on the output of outb and routb. This simulation consists of outputs a, b, c, d, e, z, and h from both lines, the left and the right. There are extra outputs z and h, based on the pipelined architecture of RIPEMD-160. Figure 4.40 shows the simulation results for outb and routb from t = 1 and t = 11. From this simulation, the outputs of outb and routb are stated in Table 4.32.



Figure 4.40 Output of outb and routb for $\mathbf{1 \leq t \leq 11}$

Table 4.32 Output of outb and routb for $\mathbf{1 \leq t \leq 11}$

rol s(t) (A, f(b,c,d),Message[t],Constant), E	**Output outb**	**rol s(t) (A, f(b,c,d),Message[t],Constant), E**	**Output routb**
$S_{11}(A, f_{1(b,c,d)}, M_0$,00000000),E	3115fc67	$S'_8(A, f_{5(b,c,d)}, M'_5$,50a28be6),E	ddd63fb8
$S_4(A, f_{1(b,c,d)}, M_1$,00000000),E	b41192d5	$S'_9(A, f_{5(b,c,d)}, M'_{14}$,50a28be6),E	322e7ae3
$S_{15}(A, f_{1(b,c,d)}, M_2$,00000000),E	3a35dc50	$S'_9(A, f_{5(b,c,d)}, M'_7$,50a28be6),E	883ee903
$S_{12}(A, f_{1(b,c,d)}, M_3$,00000000),E	d3786413	$S'_{11}(A, f_{5(b,c,d)}, M'_0$,50a28be6),E	92b2b79b
$S_5(A, f_{1(b,c,d)}, M_4$,00000000),E	0e946720	$S'_{13}(A, f_{5(b,c,d)}, M'_9$,50a28be6),E	f9091ff2
$S_8(A, f_{1(b,c,d)}, M_5$,00000000),E	d52bf632	$S'_{15}(A, f_{5(b,c,d)}, M'_2$,50a28be6),E	e5b09992
$S_7(A, f_{1(b,c,d)}, M_6$,00000000),E	150bd8a8	$S'_{15}(A, f_{5(b,c,d)}, M'_{11}$,50a28be6),E	8b2d9fb3
$S_9(A, f_{1(b,c,d)}, M_7$,00000000),E	3d6f601f	$S'_5(A, f_{5(b,c,d)}, M'_4$,50a28be6),E	e755f422
$S_{11}(A, f_{1(b,c,d)}, M_8$,00000000),E	b7b60384	$S'_7(A, f_{5(b,c,d)}, M'_{13}$,50a28be6),E	5922d09e
$S_{13}(A, f_{1(b,c,d)}, M_9$,00000000),E	b85a0a3f	$S'_7(A, f_{5(b,c,d)}, M'_6$,50a28be6),E	cf24e72c
$S_{14}(A, f_{1(b,c,d)}, M_{10}$,00000000),E	7f8b38e5	$S'_8(A, f_{5(b,c,d)}, M'_{15}$,50a28be6),E	ca6a1c75

The next simulation starts from t = 12 to t = 22. Like the RIPEMD-160 design, the pipelined RIPEMD-160 also uses different values of constant, and also different non-linear functions for the left and the right lines and message input. The constant values for both sides of the pipelined RIPEMD-160 are also different. The duration of the next simulation will continue until the final output of hash function. The left and right operations of pipelined RIPEMD-160 generate these outputs. The final hash code of the pipelined RIPEMD-160 design is obtained using the same procedure as the RIPEMD-160 design. The initial inputs a, b, c, d, and e are added to the final outputs from the left and right of the non-linear functions, as shown in Equation 4.7.

Pipe_h0_out = init_value[127:96] + left_outc + right_outd

Pipe_h1_out = init_value[95:64] + left_outd + right_oute

Pipe_h2_out = init_value[63:32] + left_oute + right_outa Equation 4.7

Pipe_h3_out = init_value[31:0] + left_outa + right_outb

Pipe_h4_out = init_value[159:128] + left_outb + right_outc



Figure 4.41 Output outb and routb for $\mathbf{78 \leq t \leq 80}$

Figure 4.41 shows the last three rounds of pipelined RIPEMD-160. Using the subsequent equation, the final hash code was obtained by combining the outputs a, b, c, d, and e from the left and right lines derived from this simulation. The final hash code for the pipelined RIPEMD-160 design is **"8eb208f7e05d987a9b044a8e98c6b087f15a0bfc"** with input message "**abc**". Table 4.33 shows similar results of outputs outb and routb from both the left and the right.

Table 4.33 Output outb and routb for $\mathbf{78 \leq t \leq 80}$

rol s(t) (A, f(b,c,d),Message[t],Constant), E	**Output outb**	**rol s(t) (A, f(b,c,d),Message[t],Constant), E**	**Output routb**
$S_8(A, f_{1(b,c,d)}, M_6$, a953fd4e),E	1a1b904d	$S'_{13}(A, f_{5(b,c,d)}, M'_3$, 00000000),E	c13f038a
$S_5(A, f_{1(b,c,d)}, M_{15}$, a953fd4e),E	0b2cdc01	$S'_{11}(A, f_{5(b,c,d)}, M'_9$, 00000000),E	bf627814
$S_6(A, f_{1(b,c,d)}, M_{13}$, a953fd4e),E	d563bfdc	$S'_{11}(A, f_{5(b,c,d)}, M'_{11}$, 00000000),E	5fccbade

Figure 4.42 shows the output of RIPEMD-160 using HashCalc Software with input "**abc**". From this figure, the output of RIPEMD-160 is **"8eb208f7e05d987a9b044a8e98c6b087f15a0bfc"**, which is similar to the simulation results obtained from ModelSim for both RIPEMD-160 and pipelined RIPEMD-160 designs. Based on the results, the simulation produces the correct output of the RIPEMD-160 hash function.

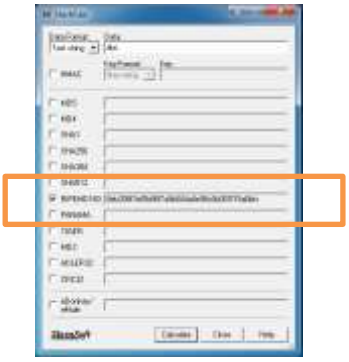


Figure 4.42 Output of RIPEMD-160 Hash Function

4.4.1.3 Discussion of RIPEMD-160 Iterative and Pipelined Design

The proposed RIPEMD-160 for both iterative design and pipelined design were successfully synthesised and implemented using Altera Quartus II on Arria II GX. The designs are written in Verilog code and simulated in functional and timing simulations using ModelSim-Altera 10.3d. The RIPEMD-160 hash function's synthesis and implementation results are illustrated in Table 4.34. The maximum frequency for pipelined design increased considerably from 135.61 MHz to 250.0 MHz in comparison to the iterative design. In addition, Quartus II Timing Optimization Advisor can also increase the speed of the design. Besides, the usage of registers decreased from 581 to 517. This method can be done using Quartus II Resource Optimization Advisor.

Table 4.34 Synthesis and Implementation of RIPEMD-160 Hash Function

Proposed Design	Device	ALUTs	Reg	FMax (MHz)	Throughput (Mbps)
RIPEMD160	Arria II GX	1192	581	134.35	838.87
RIPEMD160_pipeline	Arria II GX	2200	517	250.0	7804.88

The throughput of the design can be calculated using the same equation of throughput. For this design, the pipelined RIPEMD-160 design achieved a high throughput of 7.805 Gbps with 82 clock cycles. Table 4.35 shows the power analysis of RIPEMD-160 and pipelined RIPEMD-160 designs. From this analysis, the RIPEMD-160 design provides low-power consumption of dynamic thermal power dissipation; the power is generated from signal toggling. Numerous registers exist within the pipelined architecture. Consequently, the dynamic thermal power dissipation of the pipelined RIPEMD160 design requires substantial power consumption. Furthermore, I/O thermal power dissipation is also one of the factors explaining the high-power consumption of the pipelined RIPEMD160 compared with the RIPEMD-160 design.

Table 4.35 Power Analysis of Both RIPEMD-160 and Pipelined RIPEMD-160 Designs

Design	RIPEMD160 Design	Pipelined RIPEMD160 Design
Total Thermal Power Dissipation (mW)	461.87	585.29
Core Dynamic Thermal Power Dissipation (mW)	23.39	43.29
Core Static Thermal Power Dissipation (mW)	320.00	323.34
I/O Thermal Power Dissipation (mW)	118.48	218.65

Table 4.36 presents a comparative analysis of RIPEMD-160 hash function implementations across various platforms. The proposed designs on Arria II GX FPGA, both iterative and pipelined, demonstrate substantial improvements in terms of maximum operating frequency (FMax) and throughput compared to existing works. The proposed pipelined design achieves an FMax of 250 MHz and throughput of 7,804.88 Mbps, while the iterative version reaches 134.35 MHz with 838.87 Mbps throughput. Compared to Michail et al. (2010), whose design achieved 94.6 MHz and 3,020 Mbps, the proposed pipelined design shows a 164.4% increase in throughput and a 164.3% improvement in FMax. Against Knežević et al. (2008) who reported 100.05 MHz and 624 Mbps, the pipelined version offers a 149.9% increase in FMax and a 1,150.9% increase in throughput. These metrics underscore the effectiveness of the proposed architectural improvements.

From an area perspective, the pipelined design uses 2,200 ALUTs and 517 registers, while the iterative version uses only 1,192 ALUTs and 581 registers, significantly less than older designs such as Knežević's, which required 4,410 ALUTs. This translates to a 50.1% reduction in area compared to Knežević et al.'s design. In summary, the proposed RIPEMD-160 architectures strike a strong balance between performance and area efficiency, demonstrating that unfolding and pipelining techniques, when applied appropriately, can

yield high-speed, low-resource cryptographic designs suitable for modern FPGA implementations.

Table 4.36 Comparison of RIPEMD-160 Design with Previous Studies

Design	Device	ALUTs/ CLB	Reg	FMax (MHz)	Throughput (Mbps)
Proposed RIPEMD-160 Iterative	Arria II GX	1,192 ALUTs	581	134.35	838.87
Proposed Pipelined RIPEMD-160	Arria II GX	2,200 ALUTs	517	250.0	7,804.88
(Duong Le et al., 2023) - pipeline	ASIC-based	-		-	3090
(Kuznetsov et al., 2019) - Iterative	HashCat software and GPU	-	-	-	-
(H. Michail et al., 2010)	Xilinx Virtex-II	2548	-	94.6	3020
(Michail et al., 2008)	Xilinx Virtex-II	1985		74.6	2380
(Michail et al., 2008) - pipeline	Xilinx Virtex-E	-	-	87.6	2803
(Knežević et al., 2008)	XC2VP30	4,410 ALUTs	-	100.05	624
(Ng et al., 2004)	EPF10K50SBC 356-1	-	-	26.6	84
(Dominikus, 2002)	Xilinx Virtex 300E	1,004CL B	-	42.9	65

In conclusion, both proposed designs are successfully synthesised and implemented on Quartus II Arria II GX. The design performance of the pipelined design is enhanced in terms of the maximum frequency and area implementation. Besides, high throughput can be achieved using the pipelined technique. The maximum frequency increased by about 46%, and the percentage increment of throughput is approximately 89% compared to the iterative design. The RIPEMD-160 hash function design can be enhanced in terms of throughput and frequency by employing the pipelined design. The throughput and frequency are approximately 7.805 Gbps and 250 MHz, respectively, with small area implementation.

4.4.1.4 Discussion of RIPEMD-160 Unfolding Factors 2 and 4 Design

In this research, three proposed RIPEMD-160 algorithms were designed using Verilog HDL code: iterative, unfolding RIPEMD-160 with factor 2, and unfolding RIPEMD-160 with factor 4. The codes were compiled to check for syntax errors. The RTL codes were synthesized during logic synthesis. Logic synthesis is a translation of RTL description into gate level netlist. Logic optimisation can be applied to the design by giving clock constraint and choosing the appropriate FPGA device for technology mapping. Then, the optimised designs are obtained after this process. The testbench files are used to test and verify the simulation results of these designs. Prior to testing the designs, the input message needs to be padded. Initially, a single 1 bit of logic 1 is inserted after the message input is received, followed by logic 0. The last 64 bits are reserved for the length of the message.

The RIPEMD-160 designs were compiled with Arria II GX to identify the area, maximum frequency, and throughput of the designs. The design's performance can be enhanced by employing Quartus II advisors. The maximum frequency of the RIPEMD-160 design was determined using the TimeQuest Timing Analyzer. By giving an appropriate clock constraint to the designs, the timing requirement of the design will be met. This process is repeated until the RIPEMD-160 design achieves the target of the timing requirement of the design. In this study, an 8 ns clock constraint was given to the RIPEMD-160 iterative and RIPEMD-160 unfolding with factor four. Meanwhile, for RIPEMD-160 unfolding with factor 2, a 9 ns clock constraint was applied to the design. All the RIPEMD-160 designs meet the timing requirement.

Table 4.37 shows the synthesis and implementation results of the three proposed RIPEMD-160 designs. Other previous RIPEMD-160 designs are also shown in this table for comparison. They are evaluated from the viewpoints of the area, maximum frequency, and

throughput of the design. The throughput of the design can be calculated using the same equation. The RIPEMD-160 design's structure results in a variation in the number of clock cycles for each design. The number of clock cycles for iterative design is 101.50. The number of clock cycles is 60.5 for unfolding with a factor 2 design. Ultimately, the number of clock cycles decreased to 39.5 by employing a factor 4 design.

Table 4.37 Synthesis and Implementation Results of RIPEMD-160

RIPEMD-160 Design	FPGA Device	ALUTs/ CLBs	Reg	FMax (MHz)	Throughput (Mbps)
RIPEMD-160 (iterative)	Arria II GX	1,269 ALUTs	677	133.03	671.05
Proposed Pipelined RIPEMD-160	Arria II GX	2,200 ALUTs	517	250.0	7,804.88
RIPEMD-160 (Unfolding-2)	Arria II GX	2,015 ALUTs	530	121.65	1,029.50
RIPEMD-160 (Unfolding-4)	Arria II GX	3,386 ALUTs	851	135.92	1,761.80
(Duong Le et al., 2023) - pipeline	ASIC-based	-		-	3090
(Kuznetsov et al., 2019) - Iterative	HashCat software and GPU	-	-	-	-
(H. Michail et al., 2010)	Xilinx Virtex-II	2548	-	94.6	3020
(Michail et al., 2008)	Xilinx Virtex-II	1985		74.6	2380
(Michail et al., 2008) - pipeline	Xilinx Virtex-E	-	-	87.6	2803
(Knežević et al., 2008)	XC2VP30	4,410 ALUTs	-	100.05	624
RIPEMD-160 (iterative) (Khan et al., 2007)	XC2V250	14,911 LUTs	-	43.47	137.4
(Ng et al., 2004)	EPF10K50SBC356-1			26.6	84
(Dominikus, 2002)	Xilinx Virtex 300E	1,004CLB	-	42.9	65

There are several other FPGA family devices, such as Stratix for high-performance FPGAs and Cyclone for low-cost FPGAs. Hence, to obtain balance of power and performance, Arria II GX device was chosen because Arria series provide a low-cost transceiver of FPGA device. Besides, it also delivers optimal performance and power efficiency. The proposed RIPEMD-160 design can increase the frequency maximum of the design. The RIPEMD-160 design necessitates the use of a greater number of registers, which results in an increase in the maximum frequency. By using unfolding and following the guidelines of writing better HDL coding, the frequency of the design can be improved significantly. Besides, the architecture of FPGA device also plays important roles in the RIPEMD-160 design. By selecting an appropriate FPGA device, the performance of MD5 design can be enhanced. Table 4.36 shows the previous implementation of RIPEMD-160 design. It is difficult to find the same device for the same design due to the limitation of budget and devices.

From this table, the area implementation of the design increases from iterative design to unfolding design. However, the throughput of the RIPEMD-160 unfolding with factor four design increases significantly. The percentage of increment is about 62%. Furthermore, this design shows the highest throughput in comparison with other RIPEMD-160 designs. The number of cycles for RIPEMD-160 unfolding factor 2 decreased by two, which is from 80 cycles to 40 cycles. Then, the improvement of throughput will be obtained. Similarly, the RIPEMD-160 unfolding factor four decreased by four, in which the number of cycles decreased from 80 cycles to 20 cycles. The design's throughput and performance can be enhanced by a small number of cycles.

The simulation results for the RIPEMD-160 unfolding design with factor 4 are depicted in Figure 4.44. The message input for this simulation is "**abc**", which is "**616263**" in hexadecimal form. The message input needs to be padded before starting the processing. The output hash code is "**8eb208f7305d987a9b044a8e98c6b087f15a0bfc**". From the simulation waveform, the output hash code provides the correct results in terms of both functional and timing simulation.

The proposed RIPEMD-160 design demonstrates a notable enhancement in throughput while maintaining area efficiency. A detailed comparison was conducted among several implementations, including those by Khan et al. (2007), Knežević et al. (2008), Michail et al. (2008), and H. Michail et al. (2010). Wang et al. (2005) was included, drawing on estimated values from prior datasets. The performance-to-area ratio, quantified in Mbps per CLB or ALUT, was assessed for each design and is depicted in Figure 4.43.

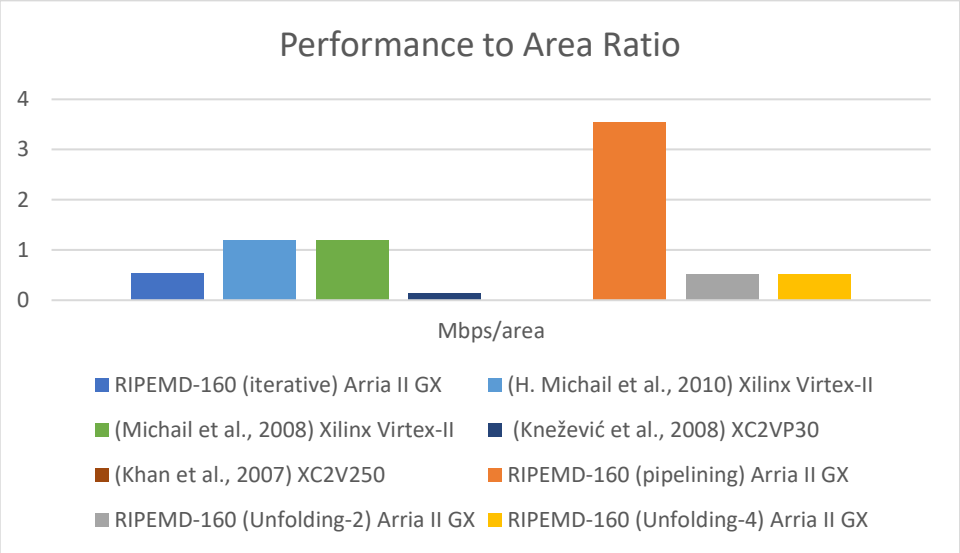


Figure 4.43 Comparison of performance-to-area ratio

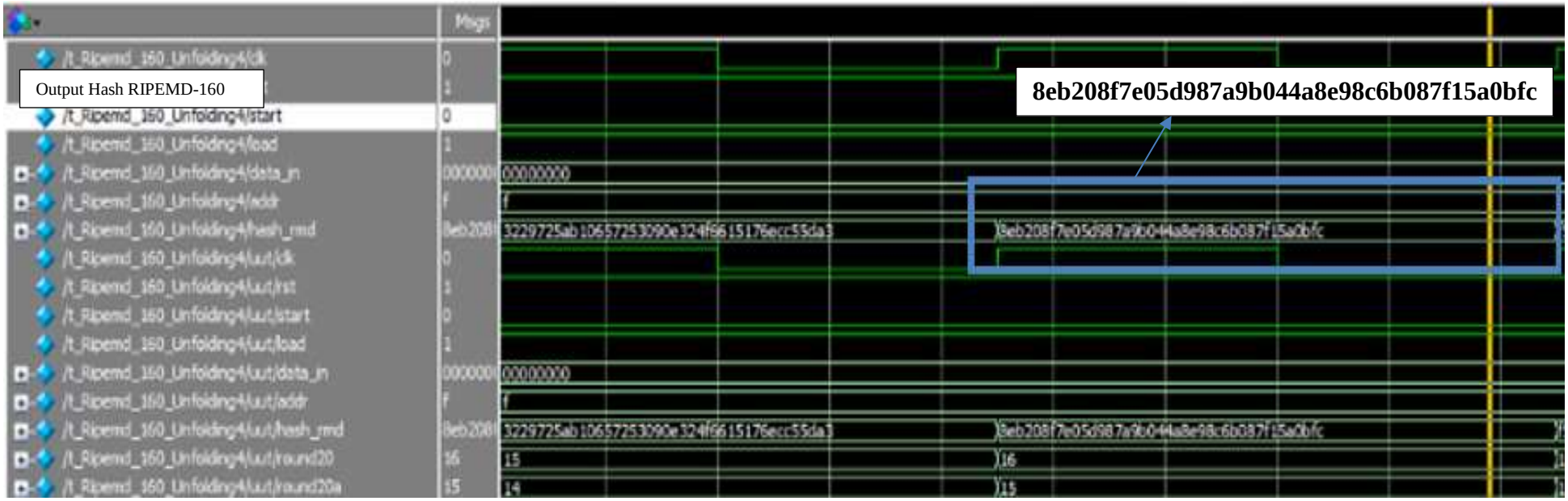


Figure 4.44 Simulation of RIPEMD-160 unfolding with factor 4 design

The most efficient design is the pipelined RIPEMD-160 on Arria II GX, which achieves a ratio of 3.5477 Mbps/ALUT, far greater than all other designs. In addition, the unfolding-based designs (factor 2 and factor 4) demonstrate competitive ratios of 0.5109 Mbps/ALUT and 0.5203 Mbps/ALUT, respectively, reflecting effective trade-offs between performance and resource utilisation. Earlier designs, such as those by Khan et al. (2007), demonstrate significantly lower efficiency at 0.0092 Mbps/CLB, while Wang et al. (2005) show an efficiency of 0.0705 Mbps/CLB. These results demonstrate how well architectural optimisations like pipelining and unfolding may improve the performance of cryptographic hardware. The RIPEMD-160 design utilising an unfolding factor of 4 achieves a maximum throughput of 1,761.80 Mbps, demonstrating the efficiency of parallelism in unfolding techniques. In summary, the use of unfolding transformation results in a high-performance and effective FPGA-based hash function design by significantly increasing throughput while preserving an acceptable area footprint, in addition to lowering processing delay by reducing the number of cycles.

4.5 Results and Discussion of SHA-256 Design Architecture

The Verilog code was employed to design the proposed SHA-256 design and SHA-256 unfolding design. Altera Quartus II was employed to analyse, synthesise, and placed and routed both designs. The synthesis and implementation results of the SHA-256 design and SHA-256 unfolding design are detailed in this section. ModelSim was employed to simulate these designs. In this work, the SHA-256 iterative design is used as a baseline reference architecture to produce a performance benchmark. This version was developed using Verilog HDL and functions as the foundational model for comparison. The proposed modifications aim to implement unfolding transformation (with factors of 2 and 4) on this baseline to decrease latency and improve throughput. The unfolding-based designs correspond with the

research objectives aimed at enhancing the speed and efficiency of SHA-256 on FPGA. Therefore, the iterative design is not the core contribution but provides essential context to evaluate the effectiveness of the proposed methods.

4.5.1 SHA-256 Design Architecture

The proposed SHA-256 iterative and SHA-256 unfolding factors 2 and 4 designs were successfully developed and evaluated. Altera Quartus II was used to compile all the designs written in Verilog code. The functionality of the design was simulated and verified using ModelSim in terms of timing and functional simulation. Finally, the throughput of these designs was calculated using the same formula to calculate the throughput of hash function. The top-level architecture of the SHA-256 design and the SHA-256 unfolding design is illustrated in Appendix section. Six modules, including counter_sha256, message_schedule, constant_sha256, mux_sha256, compression_function, and output_sha256 are included in these two SHA-256 designs. These modules were integrated together to obtain the output of SHA-256 design. Modification needs to be done for SHA-256 unfolding design to fulfil the requirement of unfolding factor 2. The input message for both designs is "**abc**".

4.5.1.1 Results of SHA-256 Iterative Design

The Verilog code was employed to effectively design the proposed SHA-256 design. Altera Quartus II was employed to analyse, synthesise, and placed and routed the design. The simulation result was obtained from ModelSim. It is evident from the results that the SHA-256 design's output provides accurate results in terms of functional and timing simulation. Figures 4.45 and 4.46 present the simulation results of SHA-256 design in terms of functional and timing simulation, respectively.

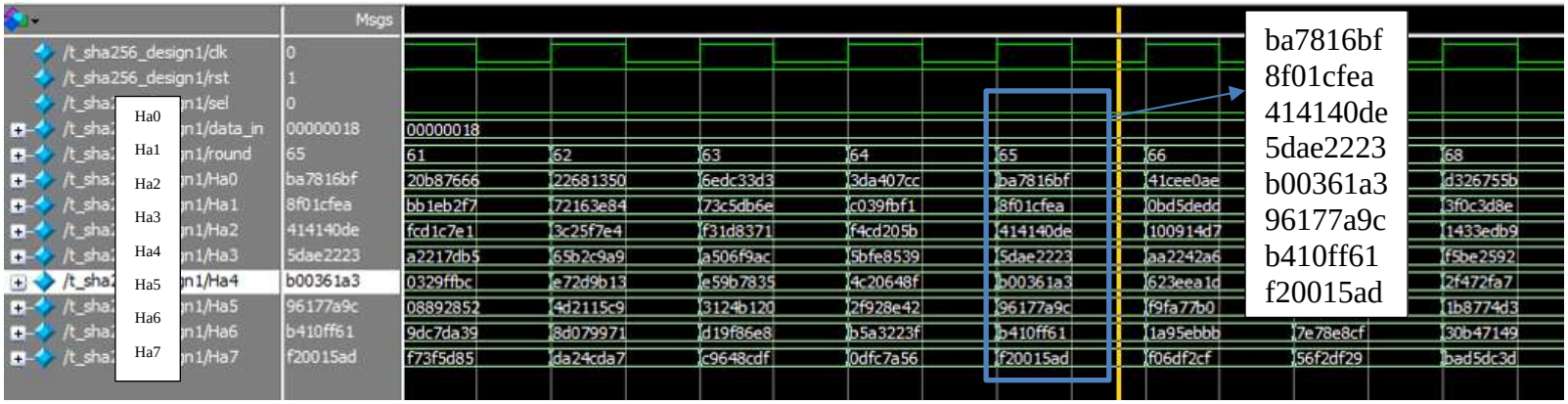


Figure 4.45 Functional Simulation of SHA-256 Design

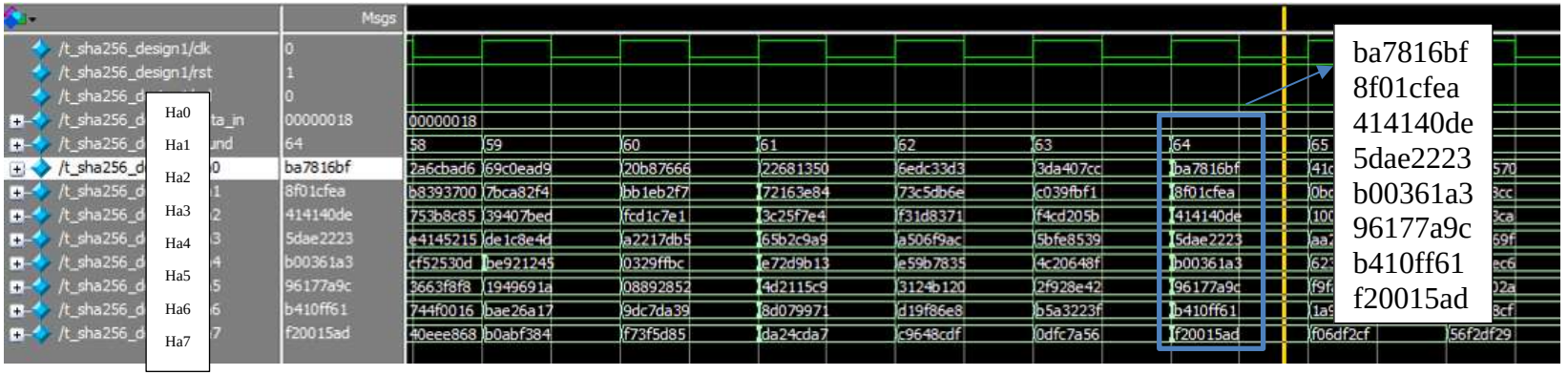


Figure 4.46 Timing Simulation of SHA-256 Design

4.5.1.1 Results of SHA-256 Unfolding Design

The latency of the SHA-256 unfolding design requires some modification. In this research, SHA-256 unfolding design was designed and synthesised using Altera Quartus II. The result was simulated using Modelsim. Figures 4.47 and 4.48 show the functional and simulation results of SHA-256 unfolding design, respectively.

From this figure, the output of SHA-256 unfolding is **"ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad"**, which is similar to the simulation results obtained from ModelSim for the SHA-256 design. Consequently, the simulation results yield the accurate output of the SHA-256 unfolding hash function.

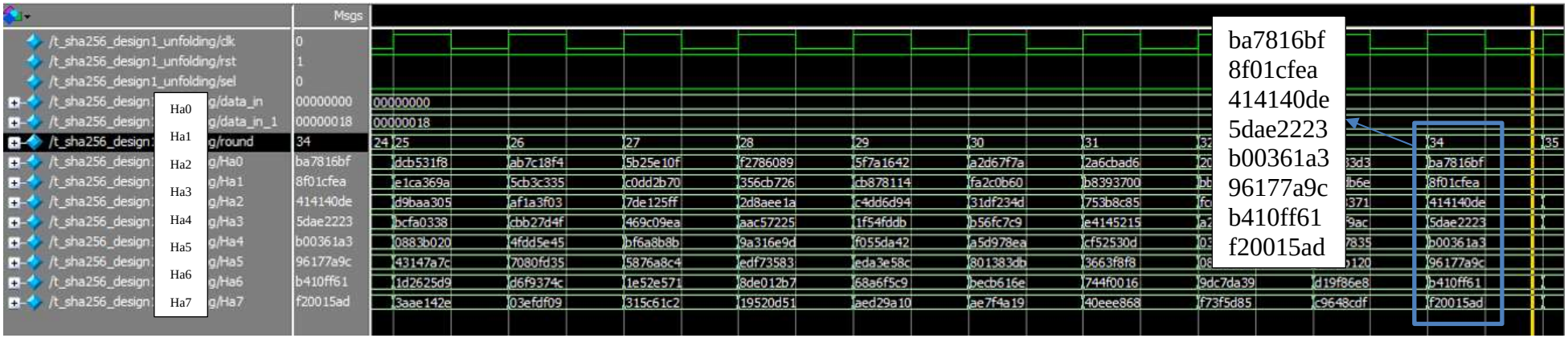


Figure 4.47 Functional Simulation of SHA-256 Unfolding Design

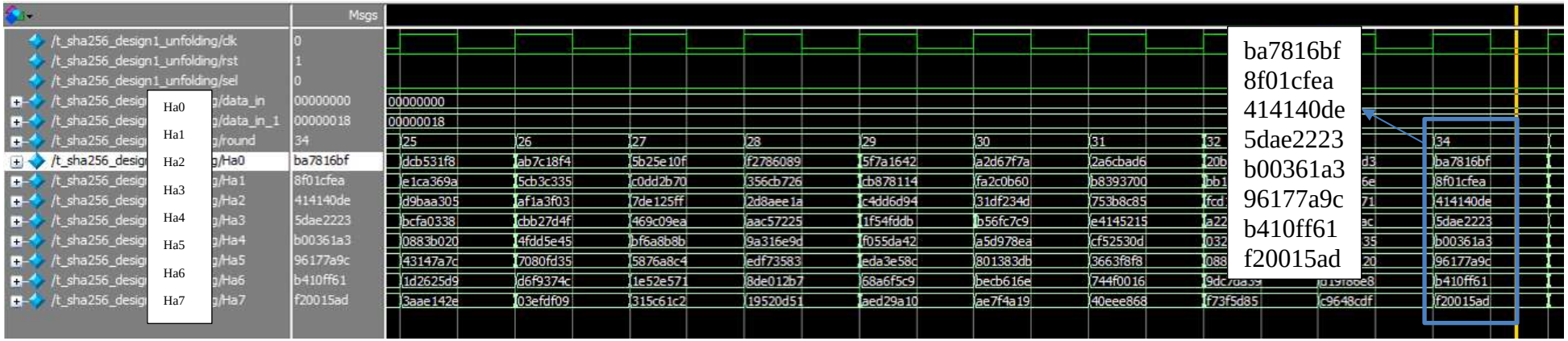


Figure 4.48 Timing Simulation of SHA-256 Unfolding Design

4.5.1.2 Discussion of SHA-256 Iterative and SHA-256 Unfolding Factor 2 Design

The synthesis and implementation results of the SHA-256 design and SHA-256 unfolding design are illustrated in Table 4.38. ModelSim was employed to simulate these designs. The throughput of these designs can be determined by employing the throughput equation, as previously mentioned.

Table 4.38 Synthesis and Implementation Results of SHA-256 Design and SHA-256 Unfolding Factor 2 Design

Design	**SHA- 256 Design**	**SHA-256 Unfolding Factor 2 Design**
Device	Arria II GX	Arria II GX
Clk Constraint (ns)	5	9
FMax (MHz)	218.9	156.59
ALUTs	1,301	1,215
Register	807	871
Throughput (Mbps)	1,660.40	2,429.52

Table 4.39 presents the proposed SHA-256, SHA-256 Unfolding, and other related SHA-256 publications. According to this table, the proposed SHA-256 employs 1,301 ALUTs and has a maximum clock frequency of 218.9 MHz. In comparison to other SHA-256 designs in various FPGA architectures, the proposed SHA-256 design achieved the highest maximal frequency with a throughput of 1,660.40 Mbps. The proposed SHA-256 design has a throughput that is nearly identical to that of the SHA-2 design by Sharif et al. (2011). This is due to the utilisation of a distinct FPGA architecture in the development of SHA-256. Additionally, Sharif et al. (2011) did not specify the type of SHA-2 that has been designed. The unfolding transformation design can be employed to enhance the throughput of the SHA-256 design. The SHA-256 design provides the highest throughput, 2,429.52 Mbps, with a maximum frequency of 156.59 MHz, as indicated by the table. In order to

facilitate a comparison of FPGA implementations, Table 4.39 also includes additional SHA-256 designs that are based on different types of FPGA devices. The current study demonstrates that the proposed SHA-256 yields superior outcomes for maximum frequency. Additionally, it employs a small area implementation utilising 1,301 ALUTs. This study presents a novel design of SHA-256 utilising the unfolding transformation approach. This technique can enhance the throughput of the SHA-256 design due to its reduced latency compared to the conventional design. The clock cycles of SHA-256 decreased from 64 cycles to 32 cycles. The increased throughput of the SHA-256 design can be achieved by the application of the unfolding transformation approach.

Table 4.39 Comparison of Implementation Results with Other SHA-256 Designs

Design	**FPGA Device**	**ALUTs/ CLBs**	**FMax (MHz)**	**Throughput (Mbps)**
SHA-256 Design	Arria II GX	1,301 ALUTs	218.9	1,660.40
SHA-256 Unfolding Design	Arria II GX	1,215 ALUTs	156.59	2,429.52
SHA-256 (Mestiri et al., 2015)	Virtex 5 XC5VFX70T	387 slices	202.54	1,580
SHA-2 (Sharif et al., 2011)	Virtex 5	320 CLBs	218.2	1,719
SHA-2 (Sharif et al., 2011)	Stratix III	795 ALUTs	205.8	1,621
SHA-256 (Miao et al., 2009)	Stratix II	2,150 ALUTs	143.164	909.816
SHA(256,384,512) (Sun et al., 2007)	Virtex v200pq 240-6	2,207 CLBs	74	291
SHA-256 (Chaves et al., 2006)	XC2PV-7	755 slices	174	1,370
SHA-256 (McEvoy, Crowe, Murphy, & Marnane, 2006b)	Virtex-II xc2v2000-bf957	1,373 slices	133.06	1,009
SHA-256 (Ahmad & Das, 2005)	-	-	41.97	335.9
SHA-256 (Sklavos & Koufopavlou, 2003)	Virtex v200pq240	1,060 CLBs	83	326

The suggested SHA-256 and its unfolding design have been successfully completed and validated through simulation and synthesis. These designs are comparable to existing SHA-256 architectures in terms of area utilization and maximum operating frequency. Notably, the proposed SHA-256 unfolding architecture achieved a peak throughput of 2,429.52 Mbps, outperforming earlier designs reported in the literature. The comparative evaluation in this section is based on studies published between 2003 and 2015, which serve as the performance baseline for iterative SHA-256 implementations (Kakarountas et al., 2006; Sklavos & Koufopavlou, 2003; Lin & Yu, 2015). Updated comparisons using more recent publications will be presented in the subsequent section discussing the SHA-256 design with an unfolding factor of 4. Given its high throughput, the proposed SHA-256 design is a promising candidate for future integration in Keyed-Hash Message Authentication Code (HMAC) applications.

4.5.1.3 Discussion of SHA-256 Iterative and SHA-256 Unfolding factor Four Design

Table 4.40 summarizes the results of other SHA-256 designs. Iterative and pipelining methodologies were employed by previous SHA-256 algorithms. The unfolding method utilises a parallel operation contingent upon the number of unfolding variables. The data in this table indicate that the unfolding SHA-256 with factor 4 design exhibited the highest throughput among all SHA-256 designs. Three types of SHA-256 algorithms were introduced to enhance throughput: iterative SHA-256 design, unfolding SHA-256 with factor 2, and unfolding SHA-256 with factor 4. When unfolding transformation was applied to the SHA-256 algorithm, the design operation improved significantly because the design is handled in parallel. By minimising the number of latencies, this strategy can assist in enhancing the performance of SHA-256 designs. Different devices produced different results

when the SHA-256 design was implemented. As a result, better results can be obtained by selecting an appropriate family device for implementation.

Table 4.40 Comparison of Synthesis and Implementation Results with Other SHA-256 Designs

SHA-256 Hash Function	Area	FPGA Device	FMax (MHz)	Throughput (Mbps)
Proposed SHA-256 Design	855 ALUTs	Arria II GX	228.15	1,756.58
Proposed SHA-256 Unfolding Design Factor 2	1,345 ALUTs	Arria II GX	251.07	3,621.07
Proposed SHA-256 Unfolding Design Factor 4	2,064 ALUTs	Arria II GX	159.82	4,196.30
SHA-256 (Bensalem et al., 2021)	67,150	Arria10	243	3,970
SHA-256 iterative(Kammoun et al., 2021)	1305	Xilinx XC7Z020	135	1063
SHA-256(Pham et al., 2020)	1,895 slices/ LUTs	Virtex 5	411.3	3,290.40
SHA-256(Chen & Li, 2020)	979 slices	Virtex 4	255.7	1,984
SHA-256 (Li et al., 2020)	-	Alpha-Data ADMPCIE-7V3	100	787
SHA-256 - Cascade(Li et al., 2019)	-	SIMC 40 nm technology (Synopsys)	227.27	3,600
SHA-256 (He et al., 2018)	7,219 cells	Cyclone II	116.2	875.22
SHA-256 (Padhi & Chaudhari, 2017)	610 slices	Virtex-4	170.75	1,344.98
SHA-256 (Kahri et al., 2015)	387 slices	Virtex-5 XC5VFX70T	202.54	1,580
SHA-2 (Shahid et al., 2011)	320 CLBs	Virtex 5	218.2	1,719
SHA-2 (Shahid et al., 2011)	795 ALUTs	Stratix III	205.8	1,621
SHA-256 (Michail et al., 2010)	1,534 CLBs	Virtex	35.1	2,077

Table 4.40 Continued

SHA-256 (Michail et al., 2010)	1,655 CLBs	Virtex E	36.4	2,190
SHA-256 (Miao et al., 2009)	2,150 ALUTs	Stratix II	143.164	909.816
SHA(256,384,512) (Sun et al., 2007)	2,207 CLBs	Virtex v200pq 240-6	74	291
SHA-256 (Chaves et al., 2006)	755 slices	XC2PV-7	174	1,370
SHA-256 (McEvoy et al., 2006)	1,373 slices	Virtex-II xc2v2000-bf957	133.06	1,009
SHA-256 (Michail et al., 2005)	-	Virtex E	64.1	2,052.1
SHA-256 (Ahmad & Das, 2005)	-	-	41.97	335.9
SHA-256 (Sklavos & Koufopavlou, 2003)	1,060 CLBs	Virtex v200pq240	83	326

The proposed design utilised 855 ALUTs and achieved a maximum clock frequency of 228.15 MHz. Consequently, the efficacy of the proposed design was markedly enhanced through the application of this technique. The findings indicate that the proposed design achieved the highest throughput of 4,196.30 Mbps and a maximum frequency of 159.82 MHz, attributable to the internal pipelining design employing 1,159 registers. Compared to traditional architecture, this technique enhanced the design by eliminating round cycles. As a result, the clock cycle count of the SHA-256 unfolding architecture decreased from 66.5 cycles to 19.5 cycles. The hash function design of SHA-256 with excellent throughput was accomplished using the unfolding transformation approach.

The performance of the hash function design can be enhanced by the proposed unfolding SHA-256 design. The design frequency can be considerably increased by

employing the unfolding method and following the criteria for developing better HDL coding. In addition, the architecture of FPGA device plays a vital part in the SHA-256 design. Consequently, the SHA-256 design's performance can be enhanced by determining the appropriate FPGA device. The earlier implementation of the SHA-256 design is shown in Table 4.41. Due to budget and device constraints, it is difficult to locate the same device for the same design. The area implementation of the design increases from iterative design to unfolding design, as shown in this table. The throughput of SHA-256 unfolding with factor 4 design, on the other hand, increases significantly. SHA-256 improves the throughput by 13.7% compared to unfolding factor 2 and by 58.1% compared to the traditional SHA-256 design. Figure 4.49 illustrates the timing simulation outcomes for the conventional SHA-256 design method, requiring 64 cycles to produce the final output result. The output of the SHA-256 hash function is represented by eight 32-bit signals: {Ha_0, Ha_1, Ha_2, Ha_3, Ha_4, Ha_5, Ha_6, Ha_7}, yielding the value "**ba7816bf8f01cfea41410de5dae2223b0036177a9cb410ff61f20015ad**".

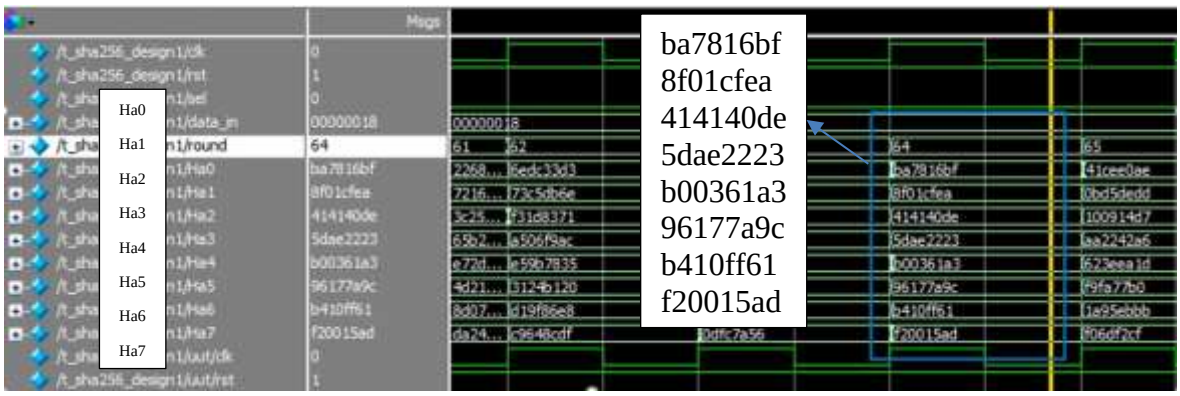


Figure 4.49 Timing Simulation Waveform of SHA-256 Design

According to the throughput equation, the number of cycles for unfolding SHA-256 with factor 2 decreased by two, from 64 to 32, as shown in Figure 4.50. Due to the clock

delay in the timing simulation, the number of cycles given in this simulation waveform was 34. Subsequently, there would be an increase in the throughput of SHA-256 design.

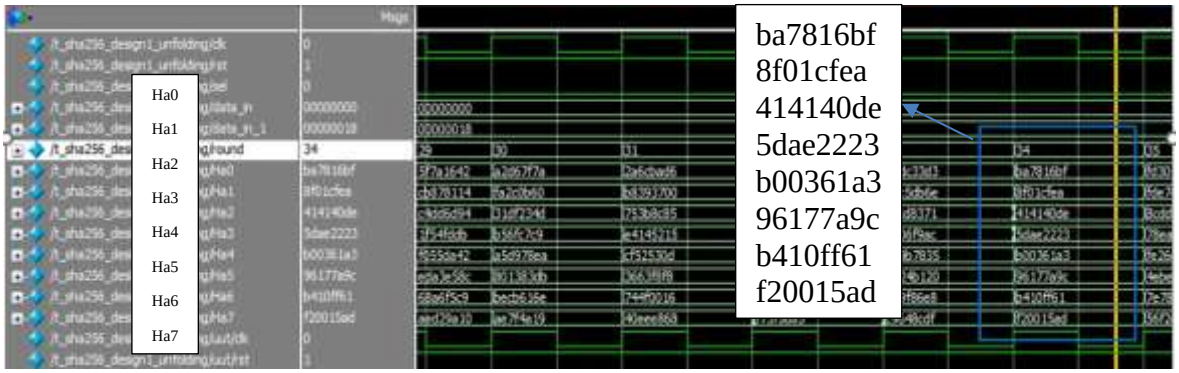


Figure 4.50 Timing Simulation Waveform of Unfolding SHA-256 with Factor 2 Design

Correspondingly, the cycles of unfolding SHA-256 factor 4 decreased by four, reducing the total number of cycles from 64 to 18. The throughput and performance of the SHA-256 design can be enhanced by a minimal number of cycles. Figure 4.51 presents the timing simulation outcomes for the unfolding SHA-256 with factor 4 design.

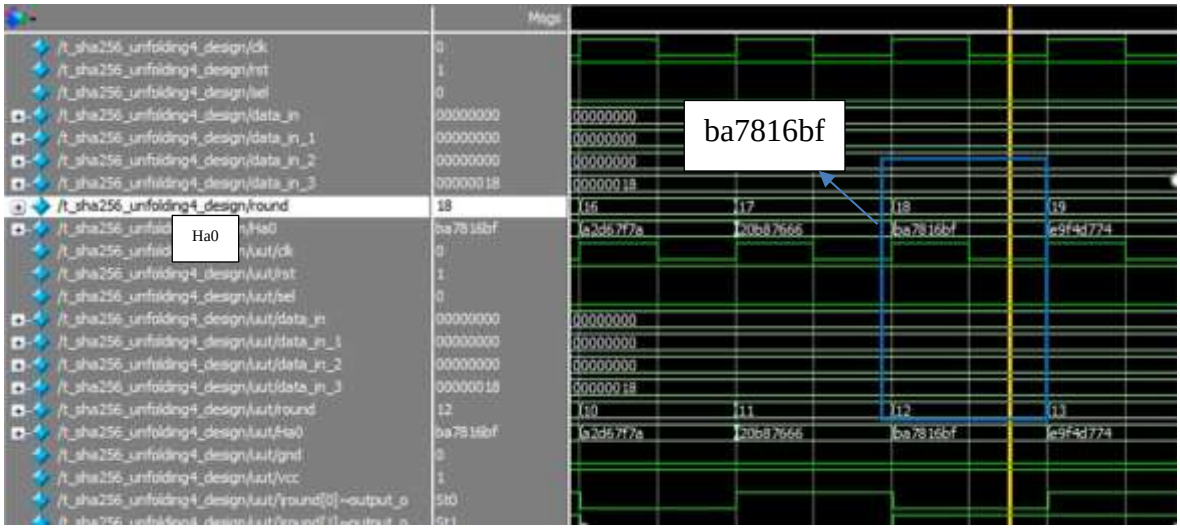
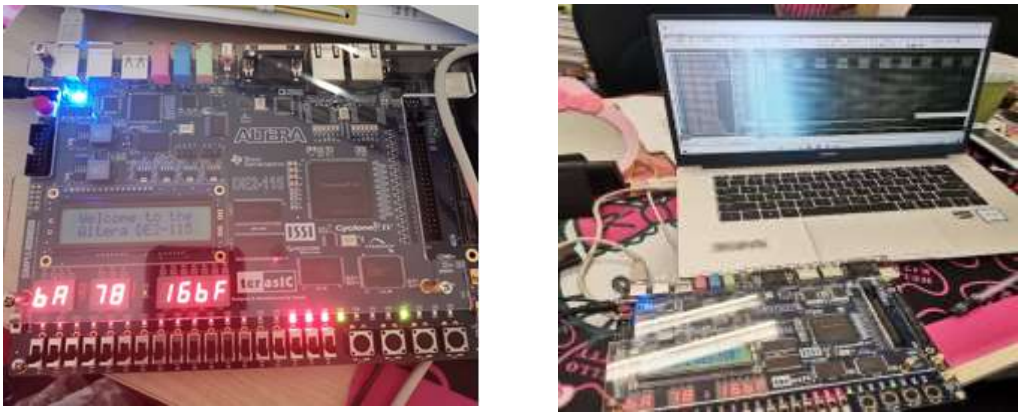


Figure 4.51 Timing Simulation Waveform of Unfolding SHA-256 with Factor Four Design

i. FPGA Implementation

The unfolding SHA-256 with factor four design was downloaded into Cyclone IV E: EP4CE115F29C7 FPGA family device to verify the hardware implementation. The SHA-256 input in this instance is the text "**abc**". As a result, the text output of the SHA-256 hash function for 256 bits should be "**ba7816bf8f01cfea41410de5dae2223b0036177a9cb410ff61f20015ad**". Due to concurrent processing that restricts the input-output (I/O) FPGA pads, only the 32-bit most significant bit (MSB) results are presented in Figure 4.52 (a). On the Cyclone IV E family device, Figures 4.52(a) and 4.52(b) show the hardware implementation of this hash function. The 32-bit MSB part of the output on the target device EP4CE115F29C7 is "**ba7816bf**". The result was evaluated using seven segments, and three red LED bits yielded "**7**" in decimal number and "**111**" in binary form. If the result of this hash function provides a correct output of 256 bits, the three LED outputs would show "**7**" in decimal number, which is "**111**" in binary form.



(a) (b)

Figure 4.52 (a) FPGA Output Implementation Display and (b) Hardware Implementation of SHA-256 Hash Function

The high-performance design of the hash function is crucial in security design. Implementing this unfolding method with factor 4 on the SHA-256 hash function substantially enhances the design's throughput. This is the optimal way to enhance the performance of the hash function. The maximum frequency of the SHA-256 design implementation demonstrates the design's critical route. To achieve a high-performance design, the speed must be prioritised, resulting in the high throughput of the SHA-256 design. ModelSim was employed to simulate the SHA-256 architecture, which was subsequently tested on an FPGA device for hardware implementation to validate the hash value output utilising six blocks of seven-segment displays and three LEDs. The proposed SHA-256 unfolding factors 2 and 4 designs are comparable to other SHA-256 methodologies in the area and maximum frequency.

According to Table 4.40, the proposed SHA-256 design with an unfolding factor of 4 had the greatest throughput of 4,196.30 Mbps, indicating a considerable improvement over earlier SHA-256 architecture. In comparison to the iterative SHA-256 design by Kammoun et al. (2021), which attained a throughput of 1,063 Mbps, the proposed design demonstrates a 294.5% enhancement in throughput. In comparison to the SHA-256 design by Kahri et al. (2015), which achieved a throughput of 1,580 Mbps, the proposed architecture exhibits a 165.5% enhancement in throughput.

The proposed design attained a maximum frequency (Fmax) of 159.82 MHz. While this is not the highest among the compared designs, it demonstrates an effective balance between speed and logic utilisation. For example, while the SHA-256 design by Pham et al. (2020) attained a maximum frequency (Fmax) of 411.3 MHz, its throughput of 3,290.40 Mbps remains 21.5% lower than that of the proposed design. This underscores the benefit of

minimising latency via unfolding rather than solely enhancing operating frequency. In terms of area, the proposed design with unfolding factor 4 employed 2,064 ALUTs, which is much less than Bensalem et al.'s (2021) design, which used 67,150 logic elements. This results in an estimated 96.9% reduction in area resources while achieving a higher throughput, with a 13% improvement over Bensalem et al.'s 3,970 Mbps.

The application of this unfolding technique has resulted in a significant improvement in the throughput of the SHA-256 design, primarily attributed to a reduction in latency. In the traditional iterative architecture, each message block requires 64 cycles. Applying an unfolding factor of 2 results in a reduction to 34 cycles, while an unfolding factor of 4 further decreases this to 18 cycles. This reduction in cycle count directly contributes to the reported throughput increases. The unfolding factor 4 approach is an effective strategy for achieving high-performance SHA-256 implementations. The percentage improvement of SHA-256 unfolding factor 4 was 13.7% and 58.1% compared to SHA-256 unfolding with factor 2 and conventional design of SHA-256, respectively. Compared to iterative design, the proposed unfolding design possesses the drawback of large area implementation. This design, however, can be enhanced by combining pipelining and unfolding techniques to provide the greatest effect. This design is essential for security applications and could be used in future innovations, such as other hash functions, HMAC, and security applications.

4.6 Results and Discussion of HMAC Design Architecture

Some applications, such as message authentication code (MAC), rely on different hashing operations. There are various hash functions, including MD5, RIPEMD-160, SHA-1, and SHA-256, among others. This section examines the execution of the hash-based message authentication code (HMAC), utilising its algorithm to ensure data validity and

integrity while enhancing hardware efficiency and design effectiveness through the SHA-1 and SHA-256 algorithms. HMAC-SHA-1 and HMAC-SHA-256 algorithms were successfully designed using Altera Quartus II and simulated using ModelSim-Altera for both functional and timing simulations to verify the output result.

4.6.1 Results of HMAC-SHA1 Design

The design and implementation of high-speed HMAC-SHA-1 was successfully conducted. The maximum frequency of the design was 260.01 MHz with 2,267 LUTs. The implementation of SHA-1 design was based on iterative design. Implementation on Arria II GX can provide high speed in comparison to other FPGA family devices. The design can be enhanced by structuring the HDL code of the design. Besides, with the help of the advisor Altera Quartus II, the design can be improved significantly. The TimeQuest Timing Analyzer can meet the time requirement of the design by giving appropriate SDC clock constraint to the design.

The top-level architecture of RTL architecture of HMAC-SHA-1 design is presented in the Appendix section to illustrate the structural. There are three modules in this design: two SHA-1 designs and a register module. The register module consists of data that need to be hashed in the second SHA-1 design. In this module, two blocks of 512 bits need to be executed in the second SHA-1 design. The rest of the design, such as the state for key with ipad and key with opad, are executed in the top-level design. In this design, the message input given for HMAC-SHA1 design is "**Sample #1**". Table 4.41 shows the input message in hexadecimal format. Using the HMAC-SHA1 algorithm entails the examination of the results of Ipad with a key. This is because these data need to be executed in the first block input of the SHA-1 design, while the messages will be concatenated at the end of the data.

Consequently, the message text can be segmented into a 32-bit hexadecimal format, as presented in Table 4.42.

Table 4.41 ASCII Code of Message Input "Sample #1"

Bit (MSB–LSB)	**8-bit Binary Hexadecimal: Symbol**				
511–471	53: S	61: a	6d: m	70: p	6c: l
470–438	65: e	20: space	23: #	31: 1	

Table 4.42 Message in Hexadecimal Format

Text:	**"Sample #1"**		
	53616d70	6c652023	31

In this project, the key input of HMAC-SHA-1 is 512 bits. Thus, it is divided into 16 blocks with 32-bit key input. Table 4.43 shows the key input for HMAC-SHA1. The value for ipad is "**36363636**" in hexadecimal format. The input key will initially be XORed with the ipad value to yield the results presented in Table 4.44. Then, the output of key ⊕ ipad will be concatenated with the text input, as shown in Table 4.45.

Table 4.43 Key Input for HMAC-SHA1

Ko/Key:	**00010203**	**04050607**	**08090a0b**	**0c0d0e0f**
	10111213	14151617	18191a1b	1c1d1e1f
	20212223	24252627	28292a2b	2c2d2e2f
	30313233	34353637	38393a3b	3c3d3e3f

Table 4.44 Key ⊕Ipad for HMAC-SHA1

Ko⊕ipad:	**36373435**	**32333031**	**3e3f3c3d**	**3a3b3839**
	26272425	22232021	2e2f2c2d	2a2b2829
	16171415	12131011	1e1f1c1d	1a1b1819
	06070405	02030001	0e0f0c0d	0a0b0809

Table 4.45 (Ko⊕Ipad) || Text for HMAC-SHA1

(Ko⊕ipad) || text:	**36373435**	**32333031**	**3e3f3c3d**	**3a3b3839**
	26272425	22232021	2e2f2c2d	2a2b2829
	16171415	12131011	1e1f1c1d	1a1b1819
	06070405	02030001	0e0f0c0d	0a0b0809
	53616d70	6c652023	31	

SHA-1 design is used to generate the output of hash code. Data key ⊕Ipad with text will be hashed together. Thus, two blocks of 512 bits need to be executed separately as the input hash is more than 512 bits. The output of the first SHA-1 is shown in Table 4.46. The second SHA-1 design utilises 160 bits of hash code. This 160-bit SHA-1 output is concatenated with the output of key ⊕ Opad. Thus, the value of key ⊕ Opad needs to be examined first. The value for Opad is "**5C5C5C5C**" in hexadecimal format. Table 4.47 presents the 512-bit output of key ⊕ opad, while Table 4.48 shows the combination of key ⊕ Opad and the output of the first SHA-1 design.

Table 4.46 Output of the First SHA-1 Design

Hash ((Ko⊕Ipad) || text):	bcc2c68c	abbbf1c3	f5b05d8e	7e73a4d2
	7b7e1b20			

Table 4.47 Ko ⊕ Opad

Ko⊕opad:	**5c5d5e5f**	**58595a5b**	**54555657**	**50515253**
	4c4d4e4f	48494a4b	44454647	40414243
	7c7d7e7f	78797a7b	74757677	70717273
	6c6d6e6f	68696a6b	64656667	60616263

Table 4.48 (Ko⊕Opad) || Hash((Key ⊕Ipad)||Text)

(Ko⊕opad) || Hash ((Key ⊕Ipad)||text):				
	5c5d5e5f	58595a5b	54555657	50515253
	4c4d4e4f	48494a4b	44454647	40414243
	7c7d7e7f	78797a7b	74757677	70717273
	6c6d6e6f	68696a6b	64656667	60616263
	bcc2c68c	abbbf1c3	f5b05d8e	7e73a4d2
	7b7e1b20			

The subsequent process, based on the HMAC-SHA1 algorithm, involves concatenating the initial output of the SHA-1 hash code with the output key ⊕ Opad, as seen in Table 4.49. Consequently, there are two 512-bit input message blocks for the second SHA-1 design, as the message input exceeds 512 bits. Then, the messages will be hashed together with the data from the first SHA-1 design. Table 4.50 shows the output generated from the second SHA-1 design, which is also known as the output for HMAC-SHA-1 design. Finally, 160-bit output of HMAC based on SHA-1 is obtained, as shown in Table 4.51.

Table 4.49 Output of the Second SHA-1 Design

HMAC(Key, Text) = Hash((Ko⊕opad) || Hash((Key ⊕ipad)||text)):				
	4f4ca3d5	d68ba7cc	0a1208c9	c61e9c5d
	a0403c0a			

Table 4.50 20-byte of HMAC-SHA-1 Output

20-byte HMAC(Key, Text):				
	4f4ca3d5	d68ba7cc	0a1208c9	c61e9c5d
	a0403c0a			

The simulation results for HMAC-SHA-1 consist of two outputs of SHA-1 design. Therefore, the output of the design needs to be examined thoroughly. The simulation results for the initial SHA-1 hash function in the HMAC-SHA-1 architecture are presented in Figure 4.53. There are 1,024 bits of message input for SHA-1. The output of key ⊕ Ipad is

concatenated with the text message. Hence, the output from the first 512 bits becomes the initial input for the next 512 bits of SHA-1 message input. In this research, the text input for simulation is "**Sample #1**". The output key ⊕ Ipad with the text input is similar to the previous section. Therefore, the output for the first SHA-1 hash function is "**bcc2c68c abbbf1c3 f5b05d8e 7e73a4d2 7b7e1b20**", as shown in Table 4.52.

Table 4.51 First SHA-1 Hash Function

Hash ((Ko⊕Ipad) || text):	**bcc2c68c**	**abbbf1c3**	**f5b05d8e**	**7e73a4d2**
	7b7e1b20			

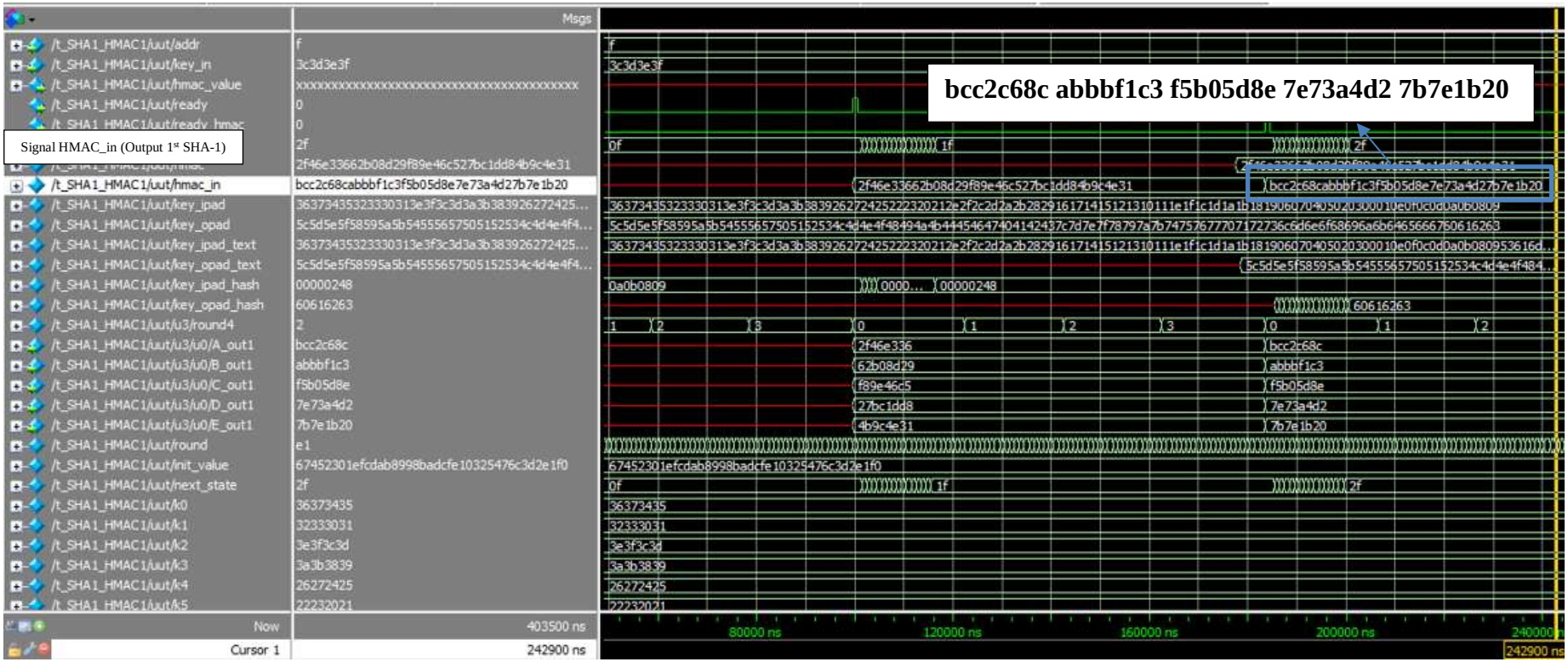


Figure 4.53 HMAC-SHA-1: Hash ((Ko⊕Ipad) || Text)

After getting the output from the first SHA-1 design, the results need to be hashed one more time by using the second SHA-1 design. However, the output of key ⊕ Opad needs to be examined. Based on the HMAC-SHA-1 algorithm, the outputs from the first SHA-1 design are concatenated with the output key ⊕ Opad. From this simulation, there are 1,024 bits of input to obtain the results of HMAC-SHA-1. The output from the first 512 bits of the second SHA-1 design will become the buffer initialisation for the second 512 bits of the second SHA-1 design. Hence, by generating the second SHA-1 hash function with 1,024 bits of input, the output HMAC based on SHA-1 can be obtained, as shown in Figure 4.72.

The 160-bit output of the HMAC, which is based on the SHA-1 design, is presented in Table 4.52. The output of HMAC-SHA-1 is "**4f4ca3d5 d68ba7cc 0a1208c9 c61e9c5d a0403c0a**".

Table 4.52 HMAC-SHA-1 Output

HMAC (Key, Text) = Hash((Ko⊕Opad) || Hash((Key ⊕Ipad)||text)):				
	4f4ca3d5	d68ba7cc	0a1208c9	c61e9c5d
	a0403c0a			

Figure 4.55 illustrates the output results of HMAC-SHA-1, which were computed using HashCalc software. The text input for HMAC is "**Sample #1**" and the key is "**000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f202122232425262728292a2b2c2d2e2f303132333435363738393a3b3c3d3e3f**". According to this figure, the output of HMAC-SHA-1 is "**4f4ca3d5d68ba7cc0a1208c9c61e9c5da0403c0a**". The outcome indicates that the simulation results of HMAC-SHA-1 from ModelSim are consistent with the output of HMAC-SHA-1. Therefore, the results obtained from HMAC-SHA-1 are accurate.

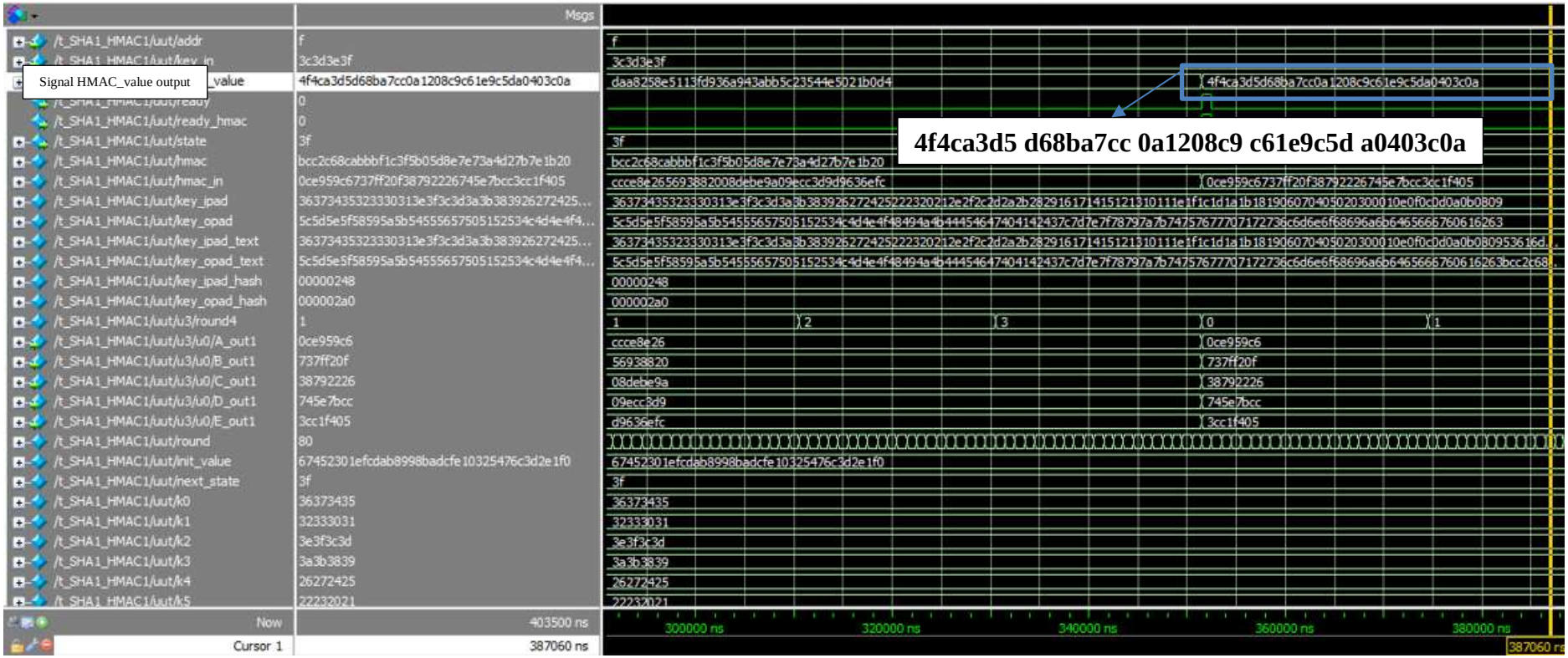


Figure 4.54 HMAC-SHA-1: Hash((Ko⊕Opad) || Hash((key⊕Ipad)||Text))



Figure 4.55 Output of HMAC- SHA-1

4.6.2 Results of HMAC-SHA256 Design

Figure 4.56 shows the timing simulation waveform results of the HMAC-SHA-256 design with the message input text "**Sample #1**". HashCalc software was used to validate the HMAC values to ensure output accuracy. Based on the simulation waveform results, the output of HMAC value provides the correct result of HMAC value without error, which is similar to the calculation from HashCalc software, as shown in Figure 4.57. FMax is the maximum clock frequency of HMAC-SHA-256 that a digital design can operate at, and it improves greatly when a clock constraint is applied to the design. Figure 4.58 depicts the maximum clock frequency of HMAC-SHA-256 with SDC 5.3 clock limitations.

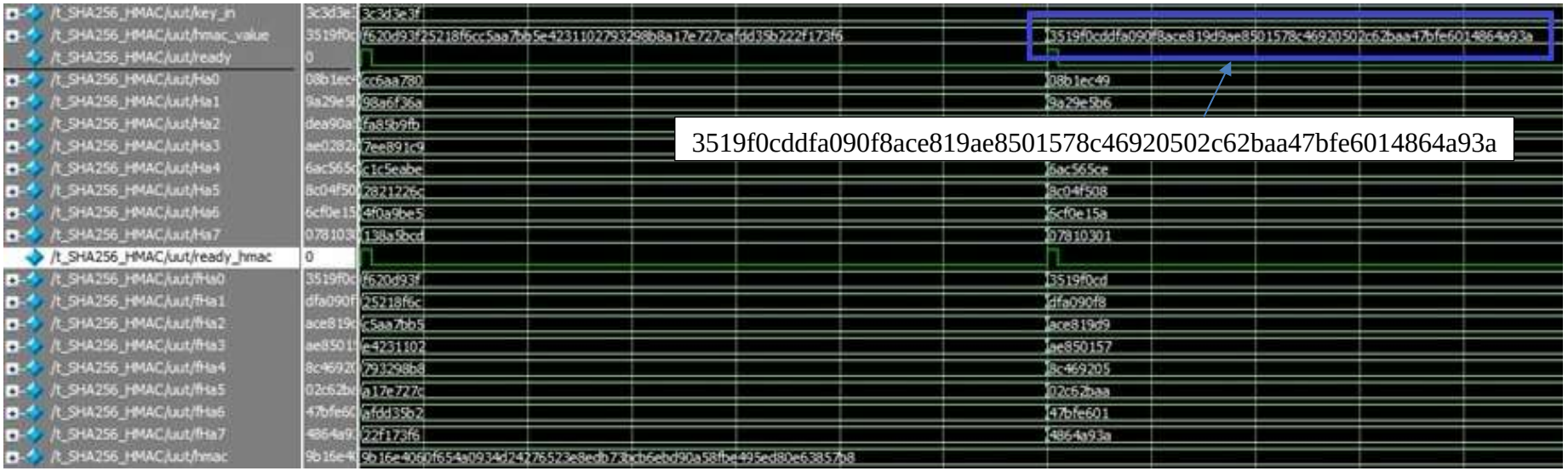


Figure 4.56 Simulation Waveform of HMAC-SHA-256 Design



Figure 4.57 HMAC Calculation of SHA-256 Design with a Specific Key

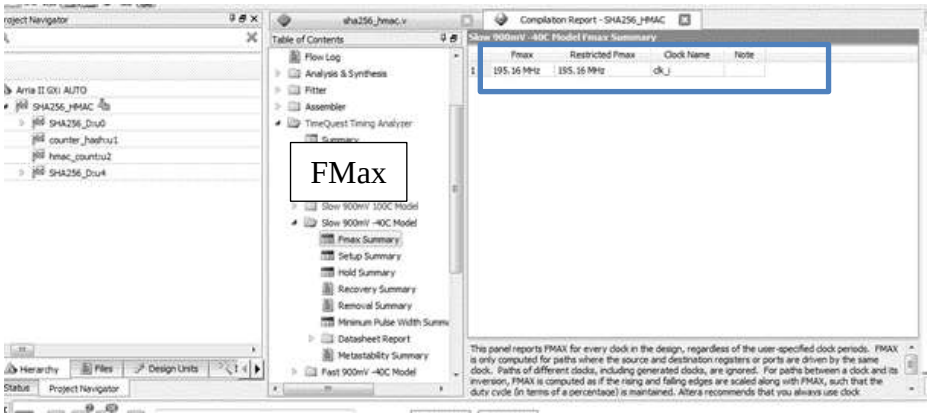


Figure 4.58 Maximum Clock Frequency of HMAC-SHA-256

4.6.3 Discussion of HMAC-SHA1 Design

Table 4.53 shows the proposed HMAC using SHA-1 design on Arria II GX and Cyclone IV GX FPGA family devices. The HMAC-SHA1 algorithm was successfully designed using Altera Quartus II 15.0. Both the functional and timing simulations were evaluated and verified using ModelSim-Altera 10.3d.

Table 4.53 HMAC-SHA-1 Design

FPGA Device	Design	FMax (MHz)	LUTs	Reg
Altera Arria II GX	Proposed HMAC-SHA-1	260.01	2,267	4,062
Altera Cyclone IV GX	Proposed HMAC-SHA-1	121.89	5,843	4,062

Table 4.52 shows that the maximum frequency for the proposed HMAC-SHA1 design increased significantly on Arria II GX, which is 260.01 MHz, with 2,267 ALUTs and a total of 4,062 registers. The synthesis and implementation on Cyclone IV GX give a low speed, which is only 121.89 MHz, with SDC clock constraint of 10 ns. The results of HMAC-SHA-1 design improve with the help of the advisor on Altera Quartus II and TimeQuest Timing Analyzer. Utilising SDC restriction of 4 ns in the HMAC-SHA-1 design on Arria II GX enhances the maximum frequency of the design, provided that the setup and hold times remain consistent. Table 4.54 presents the outcomes of the power analysis for the HMAC-SHA-1 design. To acquire the results of power analysis, the .vcd file produced after gate-level simulation must be integrated with the Verilog input design at the default cooling temperature. From this table, the dynamic thermal power dissipation of HMAC-SHA-1 design is higher than those of other designs as there are lots of switching activities during signal toggling.

Table 4.54 Power Analysis of HMAC-SHA-1 Design

Design	HMAC-SHA-1
Total Thermal Power Dissipation (mW)	320.29
Core Dynamic Thermal Power Dissipation (mW)	82.40
Core Static Thermal Power Dissipation (mW)	118.66
I/O Thermal Power Dissipation (mW)	119.22

Table 4.55 shows the proposed HMAC-SHA-1 design and other previous publications of HMAC using SHA-1 design on various types of FPGA family devices. It includes both Altera and Xilinx implementations. From this result, the synthesis and implementation on Arria II GX provide high speed with a maximum frequency of 260.01 MHz compared with other publications based on various types of FPGA family devices. This study implements HMAC utilising SHA-1 to optimise hardware resources and performance. There are several related studies on IPSec and HMAC-SHA1 to improve the hardware size, performance, and hardware usage. McLoone & McCanny (2002) proposed a single chip hardware IPSec comprising Rijndael and HMAC-SHA1. In a study by Selimis et al. (2003), the maximum frequency increased by the wireless design. HMAC with SHA1/MD5 was first introduced inWang et al. (2004), in which the hardware complexity was reduced, and an efficient structure of hash function was developed to share the hardware. Michail et al. (2004) showed an implementation on a FPGA device, whereas small-size with high-speed HMAC-SHA1 was implemented in Yiakoumis et al. (2005). Khan et al. (2007a) implemented pipelining and parallelism in the design of the HMAC-hash unit, integrating it into a single reconfigurable unit. Although these designs were successfully implemented, their maximum frequencies were limited to several tens of MHz or at most 162 MHz (Yiakoumis et al., 2006). Enhanced performance is necessary to satisfy the demands of modern systems.

Table 4.55 Comparison of FPGA-based Implementation of Previous HMAC Designs

FPGA Device	Design	FMax (MHz)	LUTs/CLBs	Reg
Altera Arria II GX	Proposed HMAC-SHA-1	260.01	2,267	4,062
Altera Cyclone IV GX	Proposed HMAC-SHA-1	121.89	5,843	4,062

Table 4.55 Continued

Xilinx Virtex-II	(Yiakoumis et al., 2006)	162	854	-
Xilinx XC2V4000	(Khan et al., 2007a)	43.47	7,484	-
Xilinx VirtexE-8	(Yiakoumis et al., 2005)	111	686	-
Xilinx V3200efg1156	(Michail et al., 2004)	62.0	6,011	-
EP2OKIOOOEB C652-IX	(Wang et al., 2004)	21.96	-	-
V150bg352	(Selimis et al., 2003)	82	1,018	-
Xilinx XCV1000E	(McLoone & McCanny, 2002)	50	7,247	-

The design and implementation of the high-speed HMAC-SHA-1 were successfully executed. The design's maximum frequency is 260.01 MHz, utilising 2,267 LUTs. The implementation of the SHA-1 design is based on an iterative design approach. Implementation on Arria II GX can provide high speed compared with other FPGA family devices. The design can be improved significantly by doing these two tasks: structure the HDL code of the design and get the assistance of the advisor of Altera Quartus II. The TimeQuest Timing Analyzer can meet the time requirement of the design by giving the appropriate SDC clock constraint to the design.

4.6.4 Discussion of HMAC-SHA-256 Design

Table 4.56 displays the proposed HMAC-SHA-256 design and other publications utilising HMAC on various FPGA family devices. HMAC-SHA-256 was successfully designed using Altera Quartus II 15.0. ModelSim-Altera 10.3d was used for functional and timing simulation to validate the output results. The maximum frequency of the HMAC-SHA-256 design increased dramatically on Arria II GX with 3,953 LUTs and 2,714 total registers, as shown in Table 4.56. Based on these findings, the synthesis and implementation

on Arria II GX give fast speed with a maximum frequency of 195.16 MHz in comparison to other HMAC publications utilising various types of hash functions on FPGA family device types. Furthermore, with the assistance of Altera Quartus II and TimeQuest Timing Analyzer advisors, the results of HMAC-SHA-256 design improve (TimeQuest Timing Analyzer, 2020). Implementing SDC clock constraint of 5.3 on the HMAC-SHA-256 design for Arria II GX ensures that the design's maximum frequency is achievable under stable setup and hold conditions. Thus, this paper proposed high-performance and error-free HMAC-SHA-256 design with an appropriate FPGA device and clock constraint that meet the design requirement.

Table 4.56 Comparison of FPGA-based Implementation Comparison of Previous HMAC-SHA-256 Designs

Authors/year	**FPGA Device**	**Design**	**FMax (MHz)**	**LUTs/ CLBs /LE**	**Reg**
Proposed Design	Altera Arria II GX	Proposed HMAC (SHA-256)	195.16	3,953	2,714
(Kieu-Do-Nguyen et al., 2022)	Virtex 4/Virtex 5	HMAC (SHA-256)	188	1,615	-
(Pham et al., 2022)	Virtex2 XC2VP20	SHA-256	165	3,695	-
(Rubayya & Resmi, 2015)	Xilinx Device (no mention device name)	HMAC (SHA-256)	110.009	6,861	-
(Juliato & Gebotys, 2011)	Altera Apex 20K, EP20K1000EBC652	HMAC (SHA-256)	35.55	9,231	-
(Juliato & Gebotys, 2011)	Xilinx Virtex-E, XCV1600EBG1156	HMAC (SHA-256)	48.12	3,463	-

Table 4.56 Continued

(Juliato & Gebotys, 2011)	Xilinx Virtex-II, XCV2V4000BF 957	HMAC (SHA-256)	59.66	3,608	-
(Khan et al., 2007)	Xilinx XC2V4000	HMAC (MD5, SHA-1, RIPEMD-160)	43.47	7,484	-
(Yiakoumis et al., 2005)	Xilinx VirtexE-8	HMAC (MD5) HMAC (SHA-1)	55 111	686	-
(Michail, 2004)	Xilinx V3200efg1156	HMAC (SHA-1)	62.0	6,011	-
(Wang, 2004)	EP2OKIOOOE BC652-IX	HMAC (SHA-1/MD5)	22.67	-	-
(Selimis, 2003)	V150bg352	HMAC (SHA-1)	82	1,018	-
(McLoone, 2002)	Xilinx XCV1000E	HMAC (SHA-1)	50	7,247	-

A comparative analysis of HMAC-SHA-256 designs implemented on different FPGA platforms is shown in Table 4.55. With an area utilisation of 3,953 LUTs and 2,714 registers, the proposed HMAC-SHA256 design, when implemented on Altera Arria II GX with Quartus II 15.0, reaches a maximum frequency of 195.16 MHz. In contrast to Rubayya & Resmi (2015), whose HMAC-SHA-256 design employed 6,861 LUTs and recorded a maximum frequency of 110.009 MHz, the suggested design achieves a 42.4% reduction in area and a 77.4% rise in speed, demonstrating notable gains in hardware efficiency and performance. Only HMAC-SHA-256 architectures are compared in this section since these designs differ from standalone SHA-256 cores in that they include extra overhead such key padding, inner and outer hashing, and iterative control. Because of their different functional

scopes and resource requirements, direct comparisons between HMAC-based designs and pure SHA-256 architectures would consequently produce inaccurate results.

The design of HMAC-SHA-256 was successfully designed through the use and development of high-speed computing, which possessed a maximum frequency of 195.16 MHz. FPGA implementation on Arria II GX can offer great speed. Moreover, the design can be significantly enhanced with the assistance of the advisor Altera Quartus II. Furthermore, providing the design with proper SDC clock constraints will allow the TimeQuest Timing Analyzer to meet the time requirements.

Table 4.56 compares the proposed HMAC-SHA-256 design with previously published HMAC-SHA-256 implementations, focussing on two critical performance metrics: maximum frequency and area utilisation. All implementations were synthesised and simulated with Quartus II on the Altera Arria II GX FPGA platform. The suggested HMAC-SHA-256 architecture demonstrated superior performance, attaining a maximum frequency of 195.16 MHz and a throughput of 4,196.30 Mbps. The increasing trend in throughput is directly attributed to architectural optimization, particularly through unfolding techniques that reduce the number of processing cycles per block.

4.7 Summary

In summary, this chapter focuses on the results and discussion of hash function with an FPGA implementation design. After the synthesis process, the testbench input file together with the design file were compiled to obtain the simulation waveform in terms of functional and timing simulation. The maximum frequency of the design can be obtained from the Quartus II CAD tool, and the design slack needs to be evaluated whether it meets the timing requirement of the design. This approach avoids violation in the design, especially

in terms of setup and hold time. Then, the design was validated using FPGA Cyclone IV board. This study used unfolding transformation and Gray encoding approaches to optimise SHA-1, MD5, RIPEMD-160, and SHA-256 hash function designs for FPGA implementations. The unfolding approach was used to increase throughput through concurrent execution of internal processes by decreasing latency and enhancing parallelism. This technique is supported by proven research and has demonstrated efficiency in pipelined architectures.

On the other hand, Gay encoding was used to solve dynamic power usage since it reduces signal transitions by changing only one bit between consecutive states. The decrease in switching activity plays a crucial role in reducing overall power consumption, especially in clocked components like state machines and control logic. These methods were chosen not at random, but because of their theoretical advantages, compatibility with previous efforts, and observable influence on area, power, and performance, as illustrated throughout this chapter. The proposed designs thus represent a balanced and efficient architectural solution, optimized for high-speed, low-power cryptographic applications on FPGA platforms.

This study does not examine the cryptographic resistance or security vulnerabilities of hash algorithms; however, it enhances practical security deployment by demonstrating the efficient implementation of these functions on FPGA, optimising for area, speed, and power. The structural integrity of each algorithm is preserved, and no alterations to the cryptographic core logic are made. Thus, the security properties of the algorithms, as outlined in their original specifications, are maintained in this hardware implementation.

CHAPTER 5

CONCLUSION

5.1 Summary of the Research

This work directly addressed the study's central challenge, which is a lack of high-performance and energy-efficient cryptographic hash function architectures designed for HMAC applications on reconfigurable hardware platforms. Existing implementations frequently suffer from low throughput, inefficient area utilisation, and high dynamic power consumption, limiting their usability in resource-constrained embedded systems like IoT and secure routers. This study proposes and validates a systematic design methodology that incorporates unfolding transformation, pipelining, Gray encoding, and clock gating techniques to address these challenges. The findings illustrate the contribution of each technique in addressing the limitations identified in the problem statement. The implementation of unfolding transformation (factors 2 and 4) on SHA-1, MD5, SHA-256, and RIPEMD-160 effectively decreased the clock cycles required for each hash computation, leading to a notable improvement in throughput. The SHA-256 unfolding factor 4 design attained a throughput of 4,196.30 Mbps, indicating a 294.5% enhancement compared to earlier architectures. The incorporation of Gray encoding in state machine design, along with the application of clock gating to inactive logic blocks, resulted in significant reductions in dynamic power consumption.

An important problem found in previous designs was excessive switching activity, which is immediately addressed by these low-power techniques. Additionally, the designs were implemented on the Altera Arria II GX FPGA, selected for its optimal performance-to-area ratio. The proposed SHA-1 implementation utilised 548 ALUTs and 907 registers,

representing reductions of 98.38% and 97.17%, respectively, when compared to Stratix II GX-based implementations, thereby demonstrating substantial area efficiency. Multiple hash functions were also successfully integrated into a single HMAC design. The HMAC-SHA-256 design achieved a maximum frequency of 195.16 MHz, utilising 3,953 LUTs and 2,714 registers, thereby providing an efficient solution for real-time secure communication systems. The significance of HDL coding style and design optimisation techniques was validated during the synthesis and implementation stages, especially when it came to reducing crucial route delays and reaching higher maximum clock frequencies. In conclusion, the research successfully met all four objectives and provided specific solutions to the difficulties raised in the initial problem formulation.

This work provides both methodological and practical contributions to cryptographic hardware design by presenting a validated framework for high-throughput, low-power hash function architectures optimised for HMAC and FPGA platforms.All hash functions and HMAC algorithms, namely MD5, SHA-1, RIPEMD-160, and SHA-256, were successfully designed, synthesised, and implemented using Altera Quartus II on the Arria II GX FPGA with Verilog HDL. The designs were evaluated for performance metrics, including frequency, logic utilisation, and power consumption. To further improve performance, various strategies were employed, including iterative design, state encoding, clock gating, unfolding, and pipelining. These approaches collectively contributed to achieving high performance while balancing trade-offs among speed, area, and power.

Objective 1 aims to formulate unfolding transformation factors 2 and 4 methods into hash function design. The research successfully implemented unfolding transformation with factors of 2 and 4 in the hash function design to optimise throughput. The unfolding

technique minimised the iterations needed for calculation, hence parallelising operations and significantly reduced latency. In the SHA-1 design, the number of cycles decreased from 80 to 40 by employing an unfolding factor of 2. This illustrates that the implementation of unfolding transformation markedly improves the design's throughput, rendering it appropriate for high-speed applications. The technology demonstrated scalability, with possible use to additional hash functions beyond SHA-1, including SHA-256 and RIPEMD-160.

Objective 2 aims to develop low-power design techniques using state encoding and clock gating into hash function design. Low-power optimisation techniques, including Gray encoding and clock gating, were successfully integrated into the hash function design. Gray encoding minimised dynamic power consumption by reducing switching activity in state transitions, while clock gating deactivated unused portions of the circuit to further reduce power dissipation. These techniques were applied across multiple designs, including MD5 and SHA-1, and demonstrated a significant reduction in power consumption without compromising performance. The results underscore the necessity of integrating these technologies in energy-sensitive environments, such as Internet of Things (IoT) devices and portable systems, where power efficiency is paramount.

Objective 3 aims to analyse the cryptographic hash functions in HMAC design. This research thoroughly analysed the integration of cryptographic hash functions into HMAC designs, focusing on their performance in secure communication applications. The research assessed the MD5, SHA-1, SHA-256, and RIPEMD-160 hash algorithms, analysing their appropriateness for HMAC designs for speed, resource utilisation, and security robustness. The results demonstrated that the proposed optimised hash function designs. In addition, the

computational requirements of HMAC were met and significant improvements in throughput and power efficiency were achieved compared to conventional methods. The scalability and usefulness of the proposed designs for real-world applications including blockchain systems, IoT connection, and secure data transmission are underlined by this research.

Objective 4 aims to validate the efficient implementation of hash function algorithms using FPGA. Hash functions were successfully implemented, synthesised, and validated on FPGA platforms, specifically using Altera Quartus II Arria II GX. The performance of the designs was assessed using key criteria, including maximum frequency, logic utilisation, and power consumption. The integration of the TimeQuest Timing Analyzer ensured that precise clock constraints were applied, allowing the designs to meet timing requirements with positive slack. The results confirmed that the combination of unfolding transformation and pipelining achieved high throughput, while low-power techniques optimised energy consumption. These results confirm the effectiveness and scalability of the proposed methodology for FPGA-based implementation.

The TimeQuest Timing Analyzer was utilised to optimise timing performance by applying precise clock constraints, ensuring that the designs met timing requirements with positive slack. This approach improved the maximum operating frequency of the hash function and HMAC designs, resulting in increased throughput. TimeQuest's interactive reporting and compatibility with the SDC format made it an industry-standard tool for efficient timing analysis, enabling streamlined development and performance evaluation of the hash function and HMAC architectures.

The performance of the proposed hash function implementations was extensively assessed using essential hardware parameters such as maximum operating frequency, area utilisation (logic elements and registers), and throughput. An in-depth comparison of four popular cryptographic hash functions such as SHA-256, RIPEMD-160, MD5, and SHA-1. These designs implemented based on Altera Arria II GX FPGA using Verilog HDL. To measure the impact of unfolding transformation, low-power optimisation, and pipelining on each design, they were compared to prior state-of-the-art implementations. The next sections provide an overview of the performance improvements achieved by the proposed designs, demonstrating their efficiency and scalability for use in embedded systems with limited resources and strict security measures.

In comparison, Wang et al. (2010) achieved 66.48 MHz on a Stratix II GX device, whereas Fan et al. (2023) reported 133.6 MHz without specifying the FPGA model. This reflects a 337.1% increase in operating frequency relative to Wang's design and a 123.4% increase compared to Fan’s work. In comparison, Wang et al.'s 32-stage pipelined design obtained a throughput of 32.035 Gbps, resulting in a 330.7% enhancement in throughput relative to the proposed design.

The proposed SHA-1 designs, implemented on the Arria II GX EP2AGX45DF29C4, demonstrate an area of 548 combinational ALUTs and 907 registers. This is a significant reduction compared to the 33,764 ALUTs and 32,048 registers reported by Jiang et al. (2009) on the Stratix II GX. This indicates a 98.38% reduction in ALUTs and a 97.17% reduction in registers, highlighting the significant area efficiency of the proposed design. The proposed design utilises unfolding transformation to achieve a balance between area and performance,

resulting in minimal resource utilisation and a latency of 40 cycles, making it appropriate for compact, high-speed cryptographic applications.

The most efficient design is the pipelined RIPEMD-160 on Arria II GX, which achieves a ratio of 3.5477 Mbps/ALUT, far greater than all other designs. In addition, the unfolding-based designs (factor 2 and factor 4) demonstrate competitive ratios of 0.5109 Mbps/ALUT and 0.5203 Mbps/ALUT, respectively, reflecting effective trade-offs between performance and resource utilisation.

The proposed SHA-256 design with an unfolding factor of 4 had the greatest throughput of 4,196.30 Mbps, indicating a considerable improvement over earlier SHA-256 architectures. In comparison to the iterative SHA-256 design by Kammoun et al. (2021), which attained a throughput of 1,063 Mbps, the proposed design demonstrates a 294.5% enhancement in throughput. In comparison to the SHA-256 design by Kahri et al. (2015), which achieved a throughput of 1,580 Mbps, the proposed architecture exhibits a 165.5% enhancement in throughput. The proposed design attained a maximum frequency (Fmax) of 159.82 MHz. While this is not the highest among the compared designs, it demonstrates an effective balance between speed and logic utilisation. For example, while the SHA-256 design by Pham et al. (2020) attained a maximum frequency (Fmax) of 411.3 MHz, its throughput of 3,290.40 Mbps remains 21.5% lower than that of the proposed design. This underscores the benefit of unfolding transformation in boosting throughput without excessively increasing clock speed.

This research successfully achieved its objectives by proposing and validating innovative methodologies to optimise cryptographic hash functions for HMAC applications. The integration of unfolding transformation, pipelining, state encoding, and clock gating

techniques demonstrated significant improvements in throughput, power efficiency, and resource utilisation. The findings contribute to advancing cryptographic hardware design methodologies and provide scalable solutions for secure data transmission across embedded systems, IoT networks, and financial applications.

5.2 Future Work

In future research, several approaches can be explored to further optimise hash functions and HMAC designs, particularly in achieving high-performance and resource-efficient implementations. Techniques such as resource sharing, retiming, and unfolding design offer significant potential for maximising frequency and improving the overall performance of cryptographic designs.

Resource sharing involves merging similar non-linear functions or operations across multiple hash functions. By employing this approach, area utilisation can be minimised without compromising performance. This technique is particularly effective in reducing redundant computations, which can lead to a more compact and efficient design while achieving a higher maximum operating frequency. Future studies should investigate how resource sharing can be optimised across various cryptographic algorithms to achieve scalable and modular architectures.

Retiming is a promising technique applicable to hash function and HMAC designs, aimed at minimising delays and enhancing the maximum operational frequency. This technique works by repositioning registers within the design to balance critical paths, minimising the delay and improving the overall timing performance. Future work can focus on systematically applying retiming across different hash function architectures to further enhance their performance in high-speed applications.

Unfolding design remains a critical technique for improving throughput and frequency by allowing multiple operations to occur within a single clock cycle. By increasing the unfolding factor, more operations can be parallelised, significantly reducing the number of cycles required for hash computations. Researchers can explore advanced unfolding techniques and their integration with pipelining and other optimisation methods to maximise both throughput and resource utilisation.

As FPGAs become more economically viable for low-end and consumer-grade applications, future work can explore cost-effective implementations of cryptographic designs on emerging FPGA platforms. These include leveraging the latest FPGA technologies to improve energy efficiency, scalability, and performance for applications such as IoT security, blockchain, and real-time data authentication.

Power consumption remains a critical consideration in cryptographic designs, especially for energy-sensitive applications like IoT and portable devices. Future research should focus on accurately estimating and minimising power consumption using tools such as Quartus II PowerPlay Power Analyzer during place-and-route stages. Although EDA tools are essential for power analysis, their limited support for low-power implementations in complex designs poses challenges. To address this, researchers should investigate alternative strategies, such as early-stage register-transfer level power estimation techniques and RTL-level power optimisation methods.

Methods such as clock gating can be optimised to minimise dynamic power consumption in system on chip designs. However, it is crucial to carefully manage conditions in clock gating to ensure that blocks are activated or deactivated appropriately, avoiding unnecessary power dissipation. Future studies can also explore the integration of

advanced dynamic power management strategies, considering real-time workload variations to optimise energy consumption.

Although EDA tools are essential for facilitating the design process, their limitations in supporting low-power implementations highlight the need for innovative design techniques. Future research should explore methodologies that can complement EDA tools, such as manually optimised RTL coding styles or power-aware algorithmic modifications. These approaches could allow designers to achieve low-power consumption without fully relying on high-level circuit design tools.

In conclusion, by adopting advanced techniques such as resource sharing, retiming, and unfolding, as well as addressing the challenges in power optimisation and EDA tool limitations, future research can pave the way for even more efficient and scalable cryptographic designs. These advancements will support the development of secure, high-performance systems for modern applications, including IoT, blockchain, and real-time data transmission, ensuring they meet the demands of an increasingly connected and data-driven world.

REFERENCES

Abbas, A., Voss, R., Wienbrandt, L., & Schimmler, M. (2014). An efficient implementation of PBKDF2 with RIPEMD-160 on multiple FPGAS. In *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS* (Vol. 2015-April, pp. 454–461). IEEE Computer Society.

Ahmad, I., & Das, A. S. (2005). Hardware implementation analysis of SHA-256 and SHA-512 algorithms on FPGAs. *Computers and Electrical Engineering*, *31*(6), 345–360.

Alamgir, N., Nejati, S., & Bright, C. (2024). SHA-256 collision attack with programmatic SAT. Retrieved from http://arxiv.org/abs/2406.20072

Albahadily, H. K., Jabbar, I. A., Altaay, A. A., & Ren, X. (2023). Issuing digital signatures for integrity and authentication of digital documents. *Al-Mustansiriyah Journal of Science*, *34*(3), 50–55.

Ali, A. M., & Farhan, A. K. (2020). A novel improvement with an effective expansion to enhance the MD5 hash function for verification of a secure e-document. *IEEE Access*, *8*, 80290–80304.

Al-Nawashi, M. M., Al-hazaimeh, O. M., Al-Qasrawi, I. S., Abu-Ein, A. A., & Al-Bsool, M. H. (2024). Analysis and evolution of SHA-1 algorithm - Analytical Technique. *International Journal of Computer Networks and Communications*, *16*(3), 89–101.

Altera. (2010). FPGA power management and modeling techniques. *Altera Corporation, 101 Innovation Drive San Jose, CA 95134*.

Anwar, M. R., Apriani, D., & Adianita, I. R. (2021). Hash algorithm in verification of certificate data integrity and security. *APTISI Transactions on Technopreneurship*, *3*(2), 67–74.

Arvind K. Sharma, & Dr.S.K. Mittal. (2019). Cryptography & network security hash function applications, attacks and advances: a review. [IEEE].

Bensalem, H., Blaquière, Y., & Savaria, Y. (2021). Acceleration of the secure hash algorithm-256 (SHA-256) on an fpga-cpu cluster using opencl. *Proceedings - IEEE International Symposium on Circuits and Systems*, *2021-May*, 3–7.

Brazhnikov, S. (2020). A hardware implementation of the sha2 hash algorithms using cmos 28nm technology. *Proceedings of the 2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering, EIConRus 2020*, 1784–1786.

Castellon, C. E., Roy, S., Kreidl, O. P., Dutta, A., & Boloni, L. (2022). Towards an energy-efficient hash-based message authentication code (hmac). In *2022 IEEE 13th International Green and Sustainable Computing Conference, IGSC 2022*. Institute of Electrical and Electronics Engineers Inc.

Chaves, R., Kuzmanov, G., Sousa, L., & Vassiliadis, S. (2006). Improving sha-2 hardware implementations. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol. 4249 LNCS).

Chaves, Ricardo, Kuzmanov, G., Sousa, L., & Vassiliadis, S. (2006). Improving SHA-2 Hardware Implementations. *Cryptographic Hardware and Embedded Systems - CHES 2006 SE - 24*, 298–310.

Chen, F., & Yuan, J. (2012). Enhanced key derivation function of hmac-sha-256 algorithmin lte network. In *Proceedings - 2012 4th International Conference on Multimedia and Security, MINES 2012* (pp. 15–18).

Chen, Yimeng, & Li, S. (2020a). A high-throughput hardware implementation of sha-256 algorithm. In *IEEE International Symposium on Circuit and Systems (ISCAS)*. IEEE.

Cheng, X.-H., & Deng, J.-Z. (2010). Design of sha-1 algorithm based on fpga. In *NSWCTC 2010 - The 2nd International Conference on Networks Security, Wireless Communications and Trusted Computing* (Vol. 1, pp. 532–534).

Choi, H., & Seo, S. C. (2020). Optimization of pbkdf2-hmac-sha256 and pbkdf2-hmac-lsh256 in cpu environments. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol. 12583 LNCS, pp. 321–333). Springer Science and Business Media Deutschland GmbH.

Dai, Z., & Zhou, N. (2003). FPGA implementation of sha-1 algorithm. *ASIC, 2003. Proceedings. 5th International Conference On, 2*, 1321-1324 Vol.2.

Deepakumara, J., Heys, H. M., & Venkatesan, R. (2001b). FPGA implementation of md5 hash algorithm. In *Canadian Conference on Electrical and Computer Engineering* (Vol. 2, pp. 919–924).

Dewi, A. S., Rancang Bangun, T., Sandi, P., Tinggi, S., Negara, S., & Setiawan, H. (2019). Implementation of sha-256 and aes-256 for securing digital al quran verification system. In *2019 Fourth International Conference on Informatics and Computing (ICIC)*.

Díez, J. M., Bojani, S., Lj, S., & Carreras, C. (2002). Hash algorithms for cryptographic protocols : fpga implementations. In *10th Telecommunications forum TELEFOR'2002, Belgrade, Yugoslavia*.

DiNardi, M., & Radhakrishnan, D. (2023). SHA-256 hash function on intel de10 lite fpga. *International Journal for Research in Applied Science and Engineering Technology*, *11*(12), 1175–1184.

Dobbertin, H., Bosselaers, A., & Preneel, B. (1996). RIPEMD-160: A strengthened version of ripemd. *Proceedings of the Third International Workshop on Fast Software Encryption*, 71–82.

Docherty, J., & Koelmans, A. (2011). A flexible hardware implementation of sha-1 and sha-2 hash functions. In *Proceedings - IEEE International Symposium on Circuits and Systems* (pp. 1932–1935).

dominikus, s. (2002). a hardware implementation of md4-family hash algorithms. *Proceedings of the IEEE International Conference on Electronics, Circuits, and Systems*, *3*, 1143–1146.

El Hadj Youssef, W., Abdelli, A., Dridi, F., & Machhout, M. (2020). Hardware implementation of secure lightweight cryptographic designs for iot applications. *Security and Communication Networks*, *2020*.

Fan, W., Liu, Q., Zhang, X., Gao, Y., Qi, X., & Wang, X. (2023). A symmetric and multilayer reconfigurable architecture for hash algorithm. *Electronics (Switzerland)*, *12*(13).

Fauziah, N. A., Rachmawanto, E. H., Setiadi, D. R. I. M., & Sari, C. A. (2018). Design and implementation of aes and sha-256 cryptography for securing multimedia file over

android chat application. In *2018 International Seminar on Research of Information Technology and Intelligent Systems, ISRITI 2018* (pp. 146–151). Institute of Electrical and Electronics Engineers Inc.

Florin, R., & Ionut, R. (2019). FPGA based architecture for securing iot with blockchain. In *2019 International Conference on Speech Technology and Human-Computer Dialogue (SpeD)*. IEEE.

Gad, A. H., Absalazeem, S. E. E., Abdelmegid, O., & Mostafa, H. (2020). Low power and area sha-256 hardware accelerator on virtex-7 fpga. In *2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES)*. IEEE.

Ganesh, T. S., Frederick, M. T., Sudarshan, T. S. B., & Somani, A. K. (2007). Hashchip: A shared-resource multi-hash function processor architecture on fpga. *Integration, the VLSI Journal*, *40*(1), 11–19.

Goos, G., Hartmanis, J., Van, J., Board, L. E., Hutchison, D., Kanade, T., Weikum, G. (2005). Optimizing sha-1 hash function for high througput with a partial unrolling study. In *Lecture Notes in LNCS 3728 Integrated Circuit and System Design. Power and Tming Modeling, Optimization and Simulation* (pp. 591–600).

Goyal, J., Ratnawat, D., Ahmed, M., & Gopalani, D. (2023). Simulation and synthesis of sha-256 using verilog hdl for blockchain applications. In *Lecture Notes in Networks and Systems* (Vol. 653 LNNS, pp. 463–477). Springer Science and Business Media Deutschland GmbH.

He, D., & Xue, Z. (2010). Multi-parallel architecture for md5 implementations on fpga with gigabit-level throughput. *Proceedings - 2010 International Symposium on Intelligence Information Processing and Trusted Computing, IPTC 2010*, 535–538.

He, Z., Liji, W., & Zhang, X. (2018). High-speed pipeline design for hmac of sha-256 with masking scheme. *12th IEEE International Conference on Anti-Counterfeiting, Security, and Identification (ASID)*, 174–178.

Henriguez, F. R., Saqib, N. A., Diaz-Perez, A., & Koc, C. kaya. (2006). Cryptographic algorithms on reconfigurable hardware. *Springer Series on Signal and Communication*.

Horpenyuk, A., Luzhetska, N., & Horpenyuk, M. (2024). Research and improvement of computing algorithms for calculating the trigonometrical coefficients of the hashing algorithm md5. *Computer Systems and Network*, *6*(1), 26–35.

Jarvinen, K. (2004). Design and implementation of a sha-1 hash module on fpgas, (Otakaari 5A, Espoo FIN-02150, Finland).

Jarvinen, K., Tommiska, M., & Skytta, J. (2005). Hardware implementation analysis of the md5 hash algorithm. *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, *00*(C), 298a–298a. Retrieved from http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1385853

Jasim, A. H., Hammood, D. A., & Al-Askery, A. (2023). Design and implementation of aes-sha security hardware using fpga. In *2023 1st International Conference on Advances in Electrical, Electronics and Computational Intelligence, ICAEECI 2023*. Institute of Electrical and Electronics Engineers Inc.

Jayapandian, N., Menagadevi, R., Abinaya, S., & Sri Sampoorani, O. (2018a). To enhance consumer privacy and security for online shopping using md5 algorithm. *Proceedings of 2017 International Conference on Innovations in Information, Embedded and Communication Systems, ICIIECS 2017*, *2018-January*, 1–4.

Jiang, L., Wang, Y., Zhao, Q., Shao, Y., & Zhao, X. (2009). Ultra-high throughput architectures for sha-1 hash algorithm on fpga. *Proceedings - 2009 International Conference on Computational Intelligence and Software Engineering, CiSE 2009*, 1–4.

John, A., & Jose, J. (2023). Hash function design based on hybrid five-neighborhood cellular automata and sponge functions. *Complex Systems*, *32*(2), 171–188.

Juliato, M., & Gebotys, C. (2011). FPGA Implementation of an HMAC Processor based on the SHA-2 Family of Hash Functions.

Kakarountas, A. P., Michail, H., Milidonis, A., & Goutis, C. E. (2006). High-speed fpga implementation of secure hash algorithm for ipsec and vpn applications. *The Journal of Supercomputing*, *37*, 179–195.

Kammoun, M., Elleuchi, M., Abid, M., & Obeid, A. M. (2021). HW/SW architecture exploration for an efficient implementation of the secure hash algorithm sha-256. *Journal of Communications Software and Systems*, *17*(2), 87–96.

Kammoun, M., Elleuchi, M., Abid, M., & S. BenSaleh, M. (2020). FPGA-based implementation of the sha_256 hash algorithm. In *2020 IEEE International Conference on Design & test of Integrated Micro & Nan-Systems (DTS)*. IEEE.

Kang, Y. K., Kim, D. W., Kwon, T. W., & Choi, J. R. (2002). An efficient implementation of hash function processor for ipsec. *IEEE Asia-Pacific Conference on ASIC , 2002*, 93–96.

Khan, B. U. I., Olanrewaju, R. F., Morshidi, M. A., Mir, R. N., Kiah, M. L. B. M., & Khan, A. M. (2022). Evolution and analysis of secured hash algrithm (sha) family. *Malaysian Journal of Computer Science*, *35*(3), 179–200.

Khan, E., El-Kharashi, M. W., Gebali, F., & Abd-El-Barr, M. (2005a). A reconfigurable hardware unit for the hmac algorithm. In *ITI 3rd International Conference on Information and Communications Technology, ICICT 2005 - Enabling Technologies for the New Knowledge Society* (Vol. 2005, pp. 861–874).

Khan, E., El-Kharashi, M. W., Gebali, F., & Abd-El-Barr, M. (2005b). An fpga design of a unified hash engine for ipsec authentication. In *Proceedings - Fifth International Workshop on System-on-Chip for Real-Time Applications, IWSOC 2005* (Vol. 2005, pp. 450–453).

Khan, E., El-Kharashi, M. W., Gebali, F., & Abd-El-Barr, M. (2007). Design and performance analysis of a unified, reconfigurable hmac-hash unit. *IEEE Transactions on Circuits and Systems I: Regular Papers*, *54*(12), 2683–2695.

Kieu-Do-Nguyen, B., Hoang, T. T., Pham, C. K., & Pham-Quoc, C. (2021). A power-efficient implementation of sha-256 hash function for embedded applications. In *International Conference on Advanced Technologies for Communications* (Vol. 2021-October, pp. 39–44). IEEE Computer Society.

Kieu-Do-Nguyen, B., Hoang, T. T., Tsukamoto, A., Suzaki, K., & Pham, C. K. (2022). High-performance multi-function hmac-sha2 fpga implementation. In *20th IEEE International Interregional NEWCAS Conference, NEWCAS 2022 - Proceedings* (pp. 30–34). Institute of Electrical and Electronics Engineers Inc.

Kim, J. W., Lee, H. U., & Won, Y. (2012). Design for high throughput sha-1 hash function on fpga. *ICUFN 2012 - 4th International Conference on Ubiquitous and Future Networks, Final Program*, 403–404.

Knežević, M., Sakiyama, K., Lee, Y. K., & Verbauwhede, I. (2008). On the high-throughput implementation of ripemd-160 hash algorithm. *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*, (1), 85–90.

Kumar, K. K., Ramaraj, E., Srikanth, B., Rao, A. S., & Prasad, P. B. V. N. (2022). Role of md5 message-digest algorithm for providing security to low-power devices. In *Proceedings - 2022 6th International Conference on Intelligent Computing and Control Systems, ICICCS 2022* (pp. 352–358). Institute of Electrical and Electronics Engineers Inc.

Kuznetsov, A., Shekhanin, K., Kolhatin, A., Kovalchuk, D., Babenko, V., & Perevozova, I. (2019). Performance of hash algorithms on gpus for use in blockchain. *2019 IEEE International Conference on Advanced Trends in Information Theory, ATIT 2019 - Proceedings*, 166–170.

Landge, I. A., & Mishra, B. K. (2016). Hardware Based MD5 Implementation using vhdl for secured embedded and vlsi based designs. In *International Conference on Communication and Electronics Systems (ICCES)* (pp. 1–6).

Lee, E. H., Lee, J. H., Park, I. H., & Cho, K. R. (2009). Implementation of high-speed sha-1 architecture. *IEICE Electronics Express*, *6*(16), 1174–1179.

Lee, E.-H., Kim, S.-M., Lee, J.-H., & Cho, K. (2009). Design of a high speed sha-1 architecture using unfolded pipeline for biomedical applications. In *Proceedings of the 13th World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI2009)* (pp. 1–4).

Lee, H.-U., Seongjing, L., Kim, J.-W., & Youjip, W. (2015). Parallelizing sha-1. *IEICE Electronics Express*, *12*(12), 1–12.

Lee, J.-H., Kim, S.-C., & Song, Y.-J. (2011). High-speed fpga implementation of the sha-1 hash function. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, *E94-A*(9), 1873–1876.

Lee, S. H., & Shin, K. W. (2018). An efficient implementation of sha processor including three hash algorithms (sha-512, sha-512/224, sha-512/256). In *International Conference on Electronics, Information and Communication, ICEIC 2018* (Vol. 2018-January, pp. 1–4). Institute of Electrical and Electronics Engineers Inc.

Lee, Y. K., Chan, H., & Verbauwhede, I. (2006). Throughput optimized sha-1 architecture using unfolding transformation. *Application Specific Systems, Architectures and Processors (ASAP'06)*, 354–359.

Lee, Y. K., Chan, H., & Verbauwhede, I. (2008). Design methodology for throughput optimum architectures of hash algorithms of the md4-class. *Journal of Signal Processing Systems*, *53*(1–2), 89–102.

Li, J., He, Z., & Qin, Y. (2019). Design of asynchronous high throughput sha-256 hardware accelerator in 40nm cmos. *Proceedings of International Conference on ASIC*, 0–3.

Li, W., Zhu, Y., Tian, L. I., Nan, T., & Chen, X. (2020). FPGA-based Hardware Acceleration for Image Copyright Protection Syetem Based on Blockchain. *Proceedings - 2020 7th IEEE International Conference on Cyber Security and Cloud Computing and 2020 6th IEEE International Conference on Edge Computing and Scalable Cloud, CSCloud-EdgeCom 2020*, 234–239.

Li, Y., Liu, F., & Wang, G. (2023). Automating collision attacks on ripemd-160. *IACR Transactions on Symmetric Cryptology*, *2023*(4), 112–142.

Lin, L., Chen, K., & Zhong, S. (2017). Enhancing the session security of zen cart based on hmac-sha256. *KSII Transactions on Internet and Information Systems*, *11*(1), 466–483.

Liu, Y., Wu, L., Niu, Y., Zhang, X., & Gao, Z. (2012). A high-speed sha-1 ip core for 10 gbps ethernet security processor. In *Proceedings of the 2012 8th International Conference on Computational Intelligence and Security, CIS 2012* (pp. 237–241).

Luan, P. H., Duong, T. S., Duong Le, V. T., Tran, T. H., & Nakashima, Y. (2023). Energy-efficient unified multi-hash coprocessor for securing iot systems integrating blockchain. In *Midwest Symposium on Circuits and Systems* (pp. 355–359). Institute of Electrical and Electronics Engineers Inc.

Ma, K. M., Le, D. H., Pham, C. K., & Hoang, T. T. (2023). Design of an soc based on 32-bit risc-v processor with low-latency lightweight cryptographic cores in fpga. *Future Internet*, *15*(5).

Makkad, R. K., & Sahu, A. K. (2017). Novel design of fast and compact sha-1 algorithm for security applications. In *2016 IEEE International Conference on Recent Trends in Electronics, Information and Communication Technology, RTEICT 2016 - Proceedings* (pp. 921–925). Institute of Electrical and Electronics Engineers Inc.

Martino, R., & Cilardo, A. (2020). Designing a sha-256 processor for blockchain-based iot applications. *Internet of Things (Netherlands)*, *11*.

McEvoy, R. P., Crowe, F. M., Murphy, C. C., & Marnane, W. P. (2006a). Optimisation of the sha-2 family of hash functions on fpgas. In *Proceedings - IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures 2006* , Vol. 2006, pp. 317–322.

McLoone, M., & McCanny, J. V. (2002). A single-chip ipsec cryptographic processor. *IEEE Workshop on Signal Processing Systems, SiPS: Design and Implementation*, *2002-Janua*, 133–138.

Mestiri, H., Kahri, F., Bouallegue, B., & Machhout, M. (2015). Efficient fpga hardware implementation of secure hash function sha-2. *International Journal of Computer Network and Information Security*, *7*(1), 9–15.

Miao, L., Jinfu, X., Xiaohui, Y., & Zhifeng, Y. (2009). Design and implementation of reconfigurable security hash algorithms based on fpga. *2009 WASE International Conference on Information Engineering*, 381–384.

Michail, H. E., Thanasoulis, V. N., Panagiotakopoulos, G. A., Kakarountas, A. P., & Goutis, C. E. (2008). Efficient pipelined hardware implementation of ripemd-160 hash function. *Worls Academy of Science, Engineering and Technology, International Journal of Electrical, Computer, Energetic, Electronic and Communication Engineering*, *2*(2), 184–186.

Michail, H. E., Thnanasoulis, V. N., Schinianakis, D. M., Panagiotakopoulos, G. A., & Goutis, C. E. (2008). Application of novel technique in ripemd-160 aiming at high-throughput. *IEEE International Symposium on Industrial Electronics*, 1937–1940.

Michail, H. E., Gregoriades, A., Kelefouras, V., Athanasiou, G., Kritikakou, A., & Goutis, C. (2010). Authentication with ripemd-160 and other alternatives: a hardware design perspective. *New Advanced Technologies*, (March).

Michail, H. E., Milidonis, A., Kakarountas, A., & Goutis, C. (2005). Novel high throughput implementation of sha-256 hash function through pre-computation technique.

Proceedings of the IEEE International Conference on Electronics, Circuits, and Systems.

Michail, H. E., Athanasiou, G., Kritikakou, A., Goutis, C., Gregoriades, A., & Papadopoulou, V. (2010). Ultra-high speed sha-256 hashing cryptographic module for ipsec hardware/software codesign. *SECRYPT 2010 - Proceedings of the International Conference on Security and Cryptography*, 309–313.

Michail, H. E., Kotsiolis, A., Kakarountas, A., Athanasiou, G., & Goutis, C. (2015). Hardware implementation of the totally self-checking sha-256 hash core. In *Proceedings - EUROCON 2015*. Institute of Electrical and Electronics Engineers Inc.

Michail, H. E., Kakarountas, A., Milidonis, A., & Goutis, C. (2009). A top-down design methodology for ultrahigh-performance hashing cores. *IEEE Transactions on Dependable and Secure Computing*, *6*(4), 255–268.

Michail, H. E., Kakarountas, A. P., Koufopavlou, O., & Goutis, C. E. (2005). A low-power and high-throughput implementation of the sha-1 hash function. In *Proceedings - IEEE International Symposium on Circuits and Systems* (pp. 4086–4089).

Michail, H.E., Kakarountas, a. P., Milidonis, a., & Goutis, C. E. (2004). Efficient implementation of the keyed-hash message authentication code (hmac) using the sha-1 hash function. *Proceedings of the 2004 11th IEEE International Conference on Electronics, Circuits and Systems, 2004. ICECS 2004.*, 567–570.

Mohamed, A., & Nadjia, A. (2015). SHA-2 hardware core for virtex-5 fpga. In *12th International Multi-Conference on Systems, Signals and Devices, SSD 2015*.

Mohammed, S. D., Rahma, A. M. S., & Hasan, T. M. (2020). Maintaining the integrity of encrypted data by using the improving hash function based on GF (28). *TEM Journal*, *9*(3), 1277–1284.

Mousavi, S. M., & Shakour, Dr. M. H. (2019). Increasing cryptographic security using message authentication code (mac). *International Journal of Engineering and Technology*, *11*(4), 912–917.

Mulder, V., Mermoud, A., Lenders, V., & Tellenbach, B. (2023). Trends in data protection and encryption technologies. *Trends in Data Protection and Encryption Technologies*. Springer Nature.

Nasution, R. M. (2022). Implementasi metode secure hash algorithm (sha-1) untuk mendeteksi orisinalitas file audio. *Bulletin of Computer Science Research*, *2*(3), 73–84.

Navabi, Z. (2006). Verilog digital system design, register transfer level synthesis, testbench, and verification, (Second Edi). McGraw-Hill, Electronic Engineering.

Ng, C., Ng, T., Yip, K., Road, P., & Kong, H. (2004). A unified architecture of md5 and ripemd-160 hash. *Proceedings of the 2004 International Symposium on Circuits and Systems*, *2*, 889–892.

Nichols, R. K. (1999). *ISCA guide to Cryptography*. McGraw-Hill.

NIST FIPS. (2002). 198: The keyed-hash message authentication code (hmac). *National Institute of Standards and Technology, Federal Information Processing Standards*, 29.

Nistrina, K., Sri Rahayu, N., & Puja Permana, R. (2024). MD5 in a web-based population data management application to improve account security. *Mantik Journal*, *8*(1), 45–54.

Nkouankou, A., Clarice, F., Abel, W., & Ndoundam, R. (2022). Pre-image attack of the md5 hash function by proportional logic. *International Journal of Research and Innovation in Applied Science*, 2454–6194. Retrieved from www.rsisinternational.org

Noaman, M. A. (2013). A vhdl model for implementation of md5 hash algorithm. *Eng.& Tech. Journal*, *31*, *Part(A*(6), 1107–1116.

Oku, D., Yanagisawa, M., & Togawa, N. (2018). Scan-based side-channel attack against HMAC-SHA-256 circuits based on isolating bit-transition groups using scan signatures. *IPSJ Transactions on System LSI Design Methodology*, *11*, 16–28.

Padhi, M., & Chaudhari, R. (2018a). An optimized pipelined architecture of sha-256 hash function. In *2017 7th International Symposium on Embedded Computing and System Design, ISED 2017* (Vol. 2018-January, pp. 1–4).

Palnitkar, S. (2003). Verilog hdl, a guide to digital design and synthesis, *IEEE 1364-2001 Compliant* (Second Edi). SunSoft Press, A Prentice Hall Title.

Parhi, K. K. (1999). VLSI digital signal processing systems: design and implementation. John Wiley & Sons.

Peroumal, V., R, R., Kurkarni, A., & Thakur, P. (2022). FPGA implementation of secure block creation algorithm for blockchain technology. *ECS Transactions, The Electrochemical Society*, *107*(1).

Pham, H. L., Tran, T. H., Duong Le, V. T., & Nakashima, Y. (2022). A high-efficiency fpga-based multimode sha-2 accelerator. *IEEE Access*, *10*, 11830–11845.

Pham, H. L., Tran, T. H., Phan, T. D., Duong Le, V. T., Lam, D. K., & Nakashima, Y. (2020). Double sha-256 hardware architecture with compact message expander for bitcoin mining. *IEEE Access*, *8*, 139634–139646.

Qiong, T., & Jianwu, Y. (2012). Implementation and optimization of the high-performance sha-1 model based on fpga. In *Proceedings - 2012 International Conference on Computer Science and Service System, CSSS 2012* (pp. 687–690).

R. Saeed, T. (2016). FPGA design and implementation of data covering based on md5 algorithm. *Engineering and Technology Journal*, *34*(14), 2621–2630.

Ravilla, D., & Putta, C. S. R. (2015a). Implementation of hmac-sha256 algorithm for hybrid routing protocols in manets. In *2015 International Conference on Electronic Design, Computer Networks and Automated Verification, EDCAV 2015* (pp. 154–159). Institute of Electrical and Electronics Engineers Inc.

Ravilla, D., & Putta, C. S. R. (2015b). Routing using trust-based system with sha-2 authentication. In *Procedia Computer Science* (Vol. 46, pp. 1108–1115). Elsevier B.V.

RHenriquez, F., Saqib, N. A., Perez, A. D., & Koc, C. K. (2006). *Cryptographic Algorithms on Reconfigurable Hardware*. Springer.

Rivest, R. L. (1992). The md5 message-digest algorithm. rfc 1321, MIT Laboratory for Computer Science and RSA Data Security.

Rote, M. D., Vijendran, N., & Selvakumar, D. (2016). High performance sha-2 core using the round pipelined technique. In *2015 IEEE International Conference on Electronics, Computing and Communication Technologies, CONECCT 2015*. Institute of Electrical and Electronics Engineers Inc.

Rubayya, R. S., & Resmi, R. (2015). Memory optimization of hmac/sha-2 encryption. In *2014 1st International Conference on Computational Systems and Communications, ICCSC 2014* (pp. 282–287). Institute of Electrical and Electronics Engineers Inc.

Saju, N. S., & Sreehari, K. N. (2021). Design and execution of highly adaptable elliptic curve cryptographic processor and algorithm on fpga using verilog hdl. In *ICCISc 2021 - 2021 International Conference on Communication, Control and Information Sciences, Proceedings*. Institute of Electrical and Electronics Engineers Inc.

Sandeep Kumar, K. (2016). Implementation of sha-1 algorithm on fpga. *International Refereed Journal of Engineering and Science* (Vol. 5). Retrieved from www.irjes.com

Santos, C. E. B., Silva, L. M. D. da, Torquato, M. F., Silva, S. N., & Fernandes, M. A. C. (2024). SHA-256 hardware proposal for iot devices in the blockchain context. *Sensors*, *24*(12).

Satoh, A., & Inoue, T. (2001). ASIC-hardware-focused comparison for hash functions, 1–6.

Sediyono, E., Santoso, K. I., & Suhartono. (2013). Secure login by using one-time password authentication based on md5 hash encrypted sms. In *Proceedings of the 2013 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2013* (pp. 1604–1608).

Selimis, G., Sklavos, N., & Koufopavlou, O. (2003). VLSI implementation of the keyed-hash message authentication code for the wireless application protocol. *Proceedings of the IEEE International Conference on Electronics, Circuits, and Systems*, *1*, 24–27.

SemiEngineering. (n.d.). Clock gating. SemiEngineering. Retrieved August 5, 2025, from https://semiengineering.com/knowledge_centers/low-power/techniques/clock-gating-2/

Sharif, M. U., Rashid, R., Rogawski, M., & Gaj, K. (2011). Use of embedded fpga resources in implementations of five round three sha-3 candidates. *International Conference on Field-Programmable Logic and Application 2011*, 1–13.

Sharma, A. K., & Mittal, Dr. S. K. (2019). Cryptography & network security hash function applications, attacks and advances: a review. In *3rd International Conference on Inventive Systems and Control (ICISC 2019)* (pp. 177–188). [IEEE].

Shen, Y., & Wang, G. (2018). Improved preimage attacks on ripemd-160 and has-160. *KSII Transactions on Internet and Information Systems*, *12*(2), 727–746.

Sideris, A., Sanida, T., & Dasygenis, M. (2024). Hardware acceleration design of the sha-3 for high throughput and low area on fpga. *Journal of Cryptographic Engineering*, *14*(2), 193–205.

Sklavos, N., Dimitroulakos, G., & Koufopavlou, O. (2003). An ultra-high speed architecture for vlsi implementation of hash functions. In *Proceedings of the IEEE International Conference on Electronics, Circuits, and Systems* (Vol. 3, pp. 990–993).

Sklavos, N., & Koufopavlou, O. (2003). On the hardware implementations of the sha-2(256, 384, 512) hash algorithms. *Circuits and Systems International Symposium*, *5*, 153–156.

Sklavos, N., & Koufopavlou, O. (2005a). Implementation of the sha-2 hash family standard using fpgas. *Journal of Supercomputing*, *31*(3), 227–248.

Sklavos, N., & Koufopavlou, O. (2005b). On the hardware implementation of ripemd processor: networking high speed hashing, up to 2 gbps. *Computers and Electrical Engineering*, *31*(6), 361–379.

Sklavos, Nicolas, Alexopoulos, E., & Koufopavlou, O. (2003). Networking data integrity : high speed architectures and hardware implementations, *1*(0), 54–59.

Sondakh, C. G., Ardiansyah, R., Joefrie, Y. Y., Angreni, D. S., & Pusadan, M. Y. (2024). Implementation of data layer in blockchain network using sha256 hashing algorithm. *Advance Sustainable Science, Engineering and Technology*, *6*(2).

Stallings, W. (1996). Data & computer communications. Prentice Hall.

Sun, W., Guo, H., He, H., & Dai, Z. (2007). Design and optimized implementation of the sha-2(256, 384, 512) hash algorithms, 858–861.

Surya Prashanth, B., Upendhar, M., Tanmay Sharma, T., & Prakash Kumar, R. (2021). Secure hash algorithm using fpga. *International Research Journal of Modernization in Engineering Technology and Science Www.Irjmets.Com @International Research Journal of Modernization in Engineering*, *3*(5), 2158–2165. Retrieved from www.irjmets.com

Sy, R. C. M., Sy, M. P. M., Abucejo, J. N., Aquino, K. A., Bongalos, P. B., & Limboc, L. N. C. (2023). A qr identification system with sms notification implementing secure hash algorithm 256 (sha-256). In *2023 International Conference on Computational Intelligence, Networks and Security, ICCINS 2023*. Institute of Electrical and Electronics Engineers Inc.

Thakur, Ms. D. (2018). Low power implementation of secure hashing algorithm (sha-2) using vhdl on fpga of sha-256. *International Journal for Research in Applied Science and Engineering Technology*, *6*(5), 2298–2303.

Thiruneelakandan, A., & Thirumurugan, T. (2011). An approach towards improved cyber security by hardware acceleration of openssl cryptographic functions. In *Proceedings of International Conference on Electronics Communication and Computing Technologies 2011, ICECCT'11* (pp. 13–16).

Thomas, A., & Bhakthavatchalu, R. (2021). Implementation of sha 256 using matlab and on fpga by the application of block chain concepts. In *ICCISc 2021 - 2021 International*

Conference on Communication, Control and Information Sciences, Proceedings. Institute of Electrical and Electronics Engineers Inc.

Tian, Y., Zhang, K., Wang, P., Zhang, Y., & Yang, J. (2018). Add “salt” md5 algorithm’s fpga implementation. In *Procedia Computer Science* (Vol. 131, pp. 255–260). Elsevier B.V.

Toruan, D. L. (2021). Implementasi metode secure hash algorithm-1 untuk mendeteksi keaslian file dokumen. *Journal of Information System Research* (Vol. 2).

Uriawan, W., Ramadita, R., Putra, R. D., Siregar, R. I., & Addiva, R. (2024, July 1). Authenticate and verification source files using sha256 and hmac algorithms. *Preprints 2024, 2024070075*.

Wagner, U., & Lugrin, T. (2023). Hash functions, book chapter in trends in data protection and encryption technologies. trends in data protection and encryption technologies. Springer Nature.

Wang, M., Su, C., & Huang, C. (2004). An hmac processor with integrated sha-1 and md5 algorithms. *Proceedings of the ASP-DAC*, 456–458.

Wang, Y., Zhao, Q., Jiang, L., & Shao, Y. (2010). Ultra-high throughput implementations for md5 hash algorithm on fpga. In *High performance computing and applications, Lecture notes in Computer Science.* (pp. 433–441).

Wong, M. M., Pudi, V., & Chattopadhyay, A. (2018). Lightweight and high performance sha-256 using architectural folding and 4-2 adder compressor. In *Proceesdings of the 2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE.

Wu, R., Zhang, X., Wang, M., & Wang, L. (2020). A high-performance parallel hardware architecture of sha-256 hash in asic. *International Conference on Advanced Communication Technology, ICACT*, *2020*(5), 1242–1247.

Yiakoumis, I. I., Papadonikolakis, M. E., Michail, H. E., Kakarountas, A. P., & Goutis, C. E. (2006). Maximizing the hash function of authentication codes. *IEEE Potentials*, *25*(2), 9–12.

Yiakoumis, I., Papadonikolakis, M., Michail, H., Kakarountas, a. P., & Goutis, C. E. (2005). Efficient small-sized implementation of the keyed-hash message authentication code. *EUROCON 2005 - The International Conference on Computer as a Tool"*, *2*(1), 1875–1878.

Zhang, Y., Kim, J., Choi, K., & Shon, T. (2014). High performance and low power hardware implementation for cryptographic hash functions. *International Journal of Distributed Sensor Networks*, *2014*.

Zhong, L., Wan, W., & Kong, D. (2016). Javaweb login authentication based on improved md5 algorithm. *ICALIP 2016 - 2016 International Conference on Audio, Language and Image Processing - Proceedings*, 131–135.

Zhou, H., & Liu, Q. (2011). Hardware design for sha-1 based on fpga. In *2011 International Conference on Electronics, Communications and Control, ICECC 2011 - Proceedings* (pp. 2076–2078).

Zhuoyu, H., & Yongzhen, L. (2022). Design and implementation of efficient hash functions. In *2022 IEEE 2nd International Conference on Power, Electronics and Computer Applications, ICPECA 2022* (pp. 1240–1243). Institute of Electrical and Electronics Engineers Inc.

APPENDICES

APPENDIX A: Journal Publications/ Conference Proceedings/Other Articles

List of Publications

Scopus-Indexed Journals Publications

1. Shamsiah Suhaili, Norhuzaimin Julai, Rohana Sapawi, Nordiana Rajaee, **Towards Maximising Hardware Resources and Design Efficiency via High-Speed Implementation of HMAC based on SHA-256 Design**, Pertanika Journal Science and Technology (PJST), ISSN/ISBN 0128-7702, January 2024.

2. Suhaili S., Watanabe T. and Julai N., **Throughput Improvement of RIPEMD-160 Design using Unfolding Transformation Technique.** Journal of Optimization in Industrial Engineering, 15(1), 207-16, March 2022.

3. Shamsiah bt. Suhaili, Norhuzaimin bin Julai, **FPGA-based Implementation of SHA-256 with Improvement of Throughput using Unfolding Transformation,** Pertanika Journal Science & Technology, 30 (1), 581-603, January 2022.

4. BEST PAPER AWARD: Shamsiah binti Suhaili, Takahiro Watanabe and Norhuzaimin Julai, **High Speed and Throughput Evaluation of SHA-1 Hash Function Design with Pipelining and Unfolding Transformation Techniques**, Journal of Telecommunication, Electronics and Computer Engineering (JTEC), Vol. 9, No.3-10, pp. 19-22, 2017.

Scopus-Indexed Conference Proceedings

1. Shamsiah Suhaili, Norhuzaimin Julai, Asrani Lit, Maimun Huja Husin, **Power Efficiency in Unfolding RIPEMD-160: Dynamic Power Analysis Using Gray Encoding in FPGA Design,** IOP Conference Series: Journal of Physics, Vol. 3020, 012006, ISMED 2024, 16 June 2025

2. Shamsiah binti Suhaili, Norhuzaimin bin Julai, Asrani bin Lit, Maimun binti Huja Husin, Mohamad Faizrizwan bin Mohd Sabri, **Simulation-Based Power Estimation for High Throughput SHA-256 Design on Unfolding Transformation.** AIP Conference Proceedings, Volume 3056, Issue 1, 7 April 2025.

3. Shamsiah Suhaili, Cleopatra Chundang Anak Niam, Zainah Md. Zain and Norhuzaimin Julai, **Design and Implementation of MD5 Hash Function Algorithm using Verilog HDL,** Proceedings of the 12th National Technical Seminar on Unmanned System Technology 2020, Springer Lect. Notes Electrical Engineering (LNEE), Vol. 770 (Chapter 38) – pp 499-510.

4. Shamsiah Suhaili, Rene Brooke Fredrick, Zainah Md. Zain, Norhuzaimin Julai, **Design and Implementation of Advanced Encryption Standard Using Verilog HDL**, Proceedings of the 12th National Technical Seminar on Unmanned System Technology 2020, Springer Lect. Notes in Electrical Engineering (LNEE), Vol. 770 (Chapter 37) pp 483-498.

5. Shamsiah bt. Suhaili, Kayle Jacqueline anak Kumar, Norhuzaimin bin Julai, Maimun binti Huja Husin, Mohamad Faizrizwan bin Mohd Sabri, and Asrani bin Lit (2020), **Implementation of Verilog HDL in Calculator Design with FPGA Simulation**, 2020 13th International UNIMAS Engineering Conference (EnCon), 1(1), 1–6.

Other Articles

1. Shamsiah binti Suhaili, Joyce Huong Shing Yii, Asrani bin Lit, Maimun Huja Husin, Mohamad Faizrizwan bin Mohd Sabri, Norhuzaimin Julai, **Development of Digital Image Processing Algorithms via FPGA Implementation**, Semarak International Journal of Electronic System Engineering, August 2024.

2. Shamsiah bt. Suhaili, Takahiro Watanabe, **High-Throughput of SHA- 256 Hash Function with Unfolding Transformation**, Global Journal of Engineering & Technology Review (GJETR), Vol 4(4), pp 73 -81, 2019.

APPENDIX B: Copyright and Awards

MyIPO Copyright Registration Number:

Intellectual Property:

MyIPO Copyright LY2022 Q02690

Title of Work : **LP HiT SHA-256**: Low-Power and High-Throughput of the SHA-256 Hash Function Design.

Date of Approval : 10 June 2022

Awards:

1. InTEX22 - **Bronze Award : Simulation-based Power Estimation of Low-Power SHA-256 Cryptographic Hash Function Design Techniques**, The Innovation Technology Exposition and Conference 2022 (InTEX22), “Inspiring Innovation through Digitalisation” (Poster Presentation), 15–16 June 2022.

2. IIDEX2022 - **Silver Award : LP HiT SHA-256: Low-Power and High-Throughput of the SHA-256 Hash Function Design**, UiTM Invention, Innovation & Design Exposition (IIDEX2022),(Video Presentation), 21–25 November 2022.

3. InTEX23 - **Bronze Award : FPGA-based IoT Home Automation, The Innovation Technology Exposition and Conference 2023 (InTEX23)**, (Poster Presentation), 23 - 24 August 2023.

4. SIRAC II 2024 - **Silver Award : Development of Digital Image Processing via FPGA Implementation,** Semarak International Research Article Competition 2024 (SIRAC II 2024), (Paper Submission), July 2024.

5. MTE2025 – **Bronze Award: FPGA DomoSync: A Smart & Efficient FPGA-based IoT Home Automation**, Malaysia Technology Expo 2025 (MTE2025), (Poster Presentation), 20 -22 February 2025.

APPENDIX C: Top Level RTL viewer

This appendix presents the top-level RTL viewer outputs for all hash function design variants synthesized using Quartus II such as MD5, SHA-1, RIPEMD-160 and SHA-256 hash function. These visualizations demonstrate how iterative, unfolding, pipelining, and Gray encoding techniques affect the overall hardware architecture.

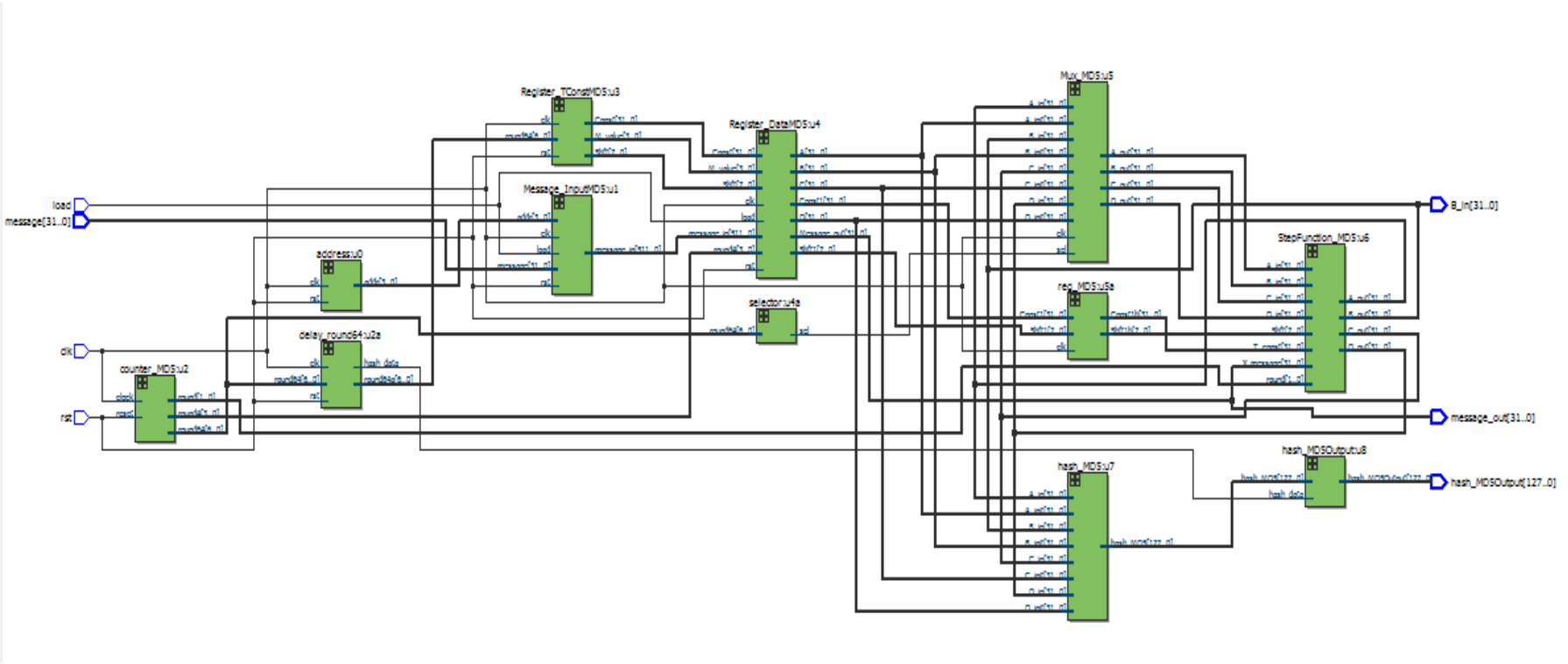


Figure 1: Top-Level Architecture of MD5 Binary Design

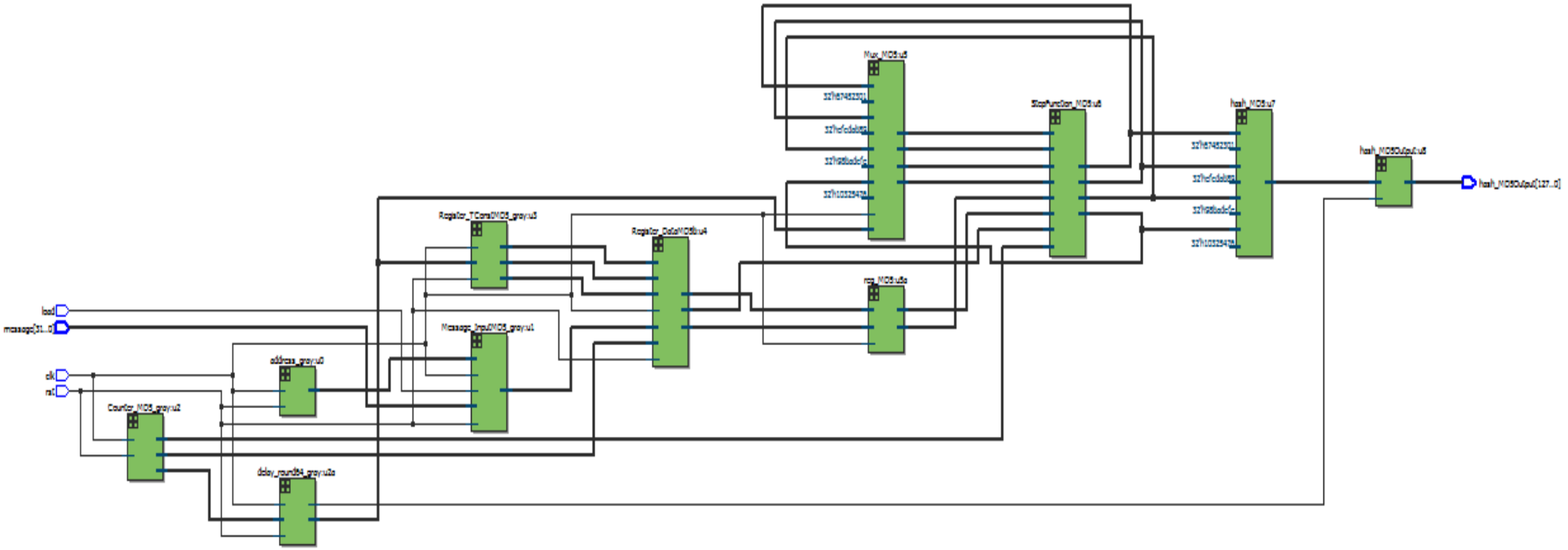


Figure 2: Top-Level Architecture of MD5 Gray Design

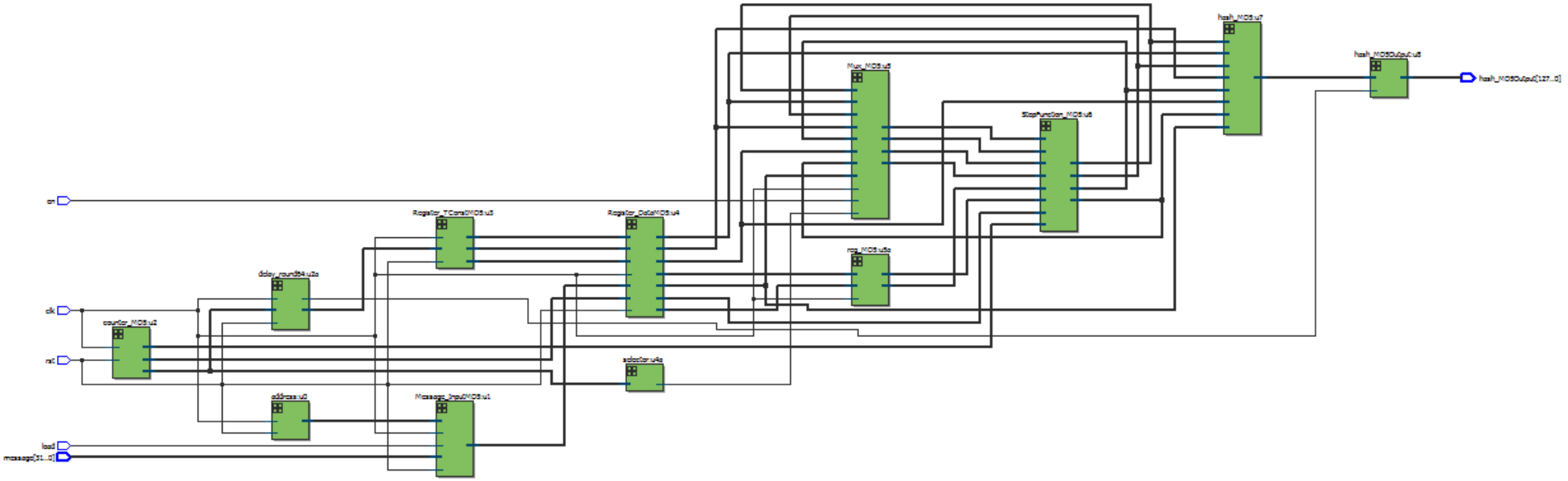


Figure 3: Top-Level Architecture of MD5 Binary with Gating Design

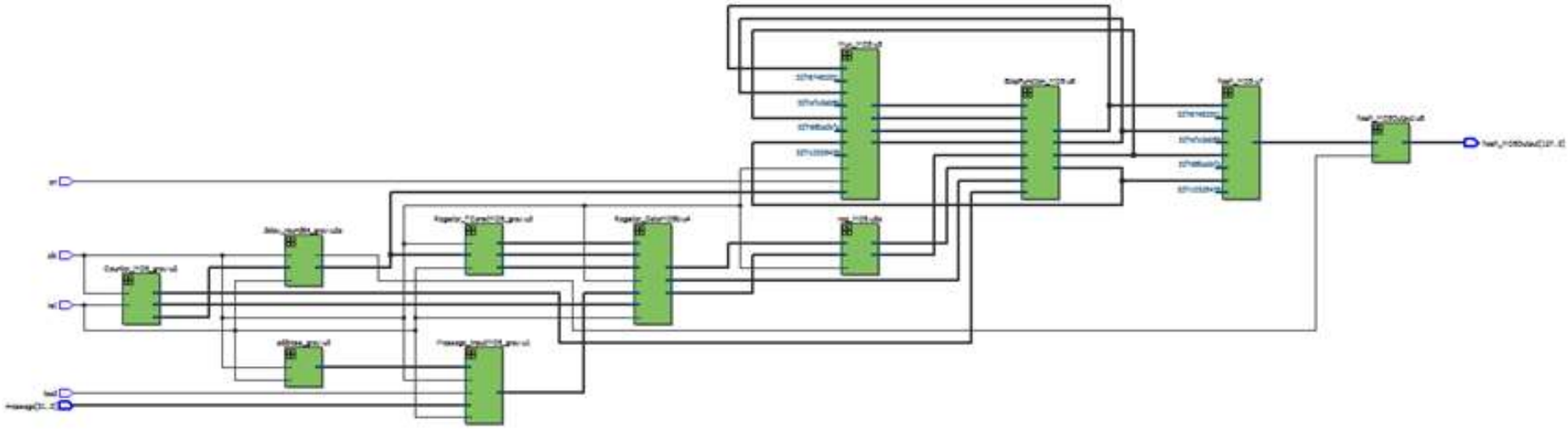


Figure 4: Top-Level Architecture of MD5a Gray Gating Design

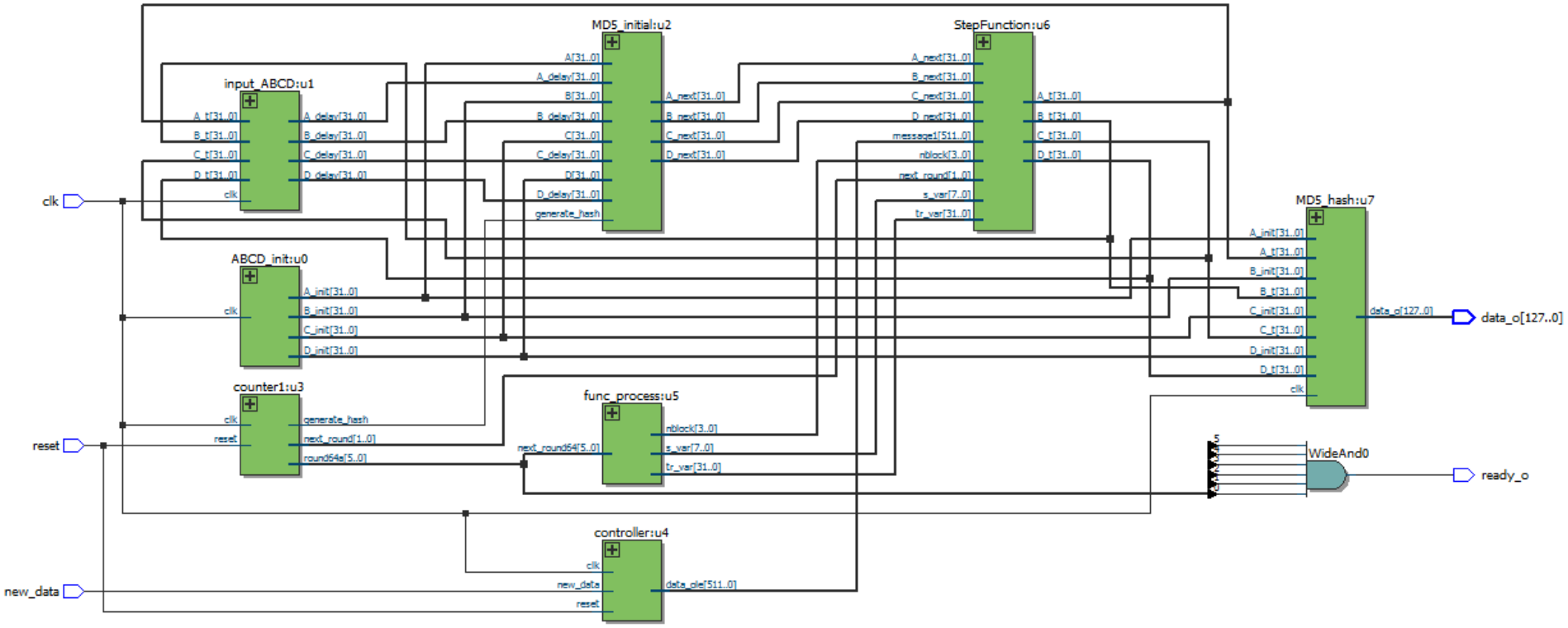


Figure 5: Top-Level Architecture of MD5 with Controller Design

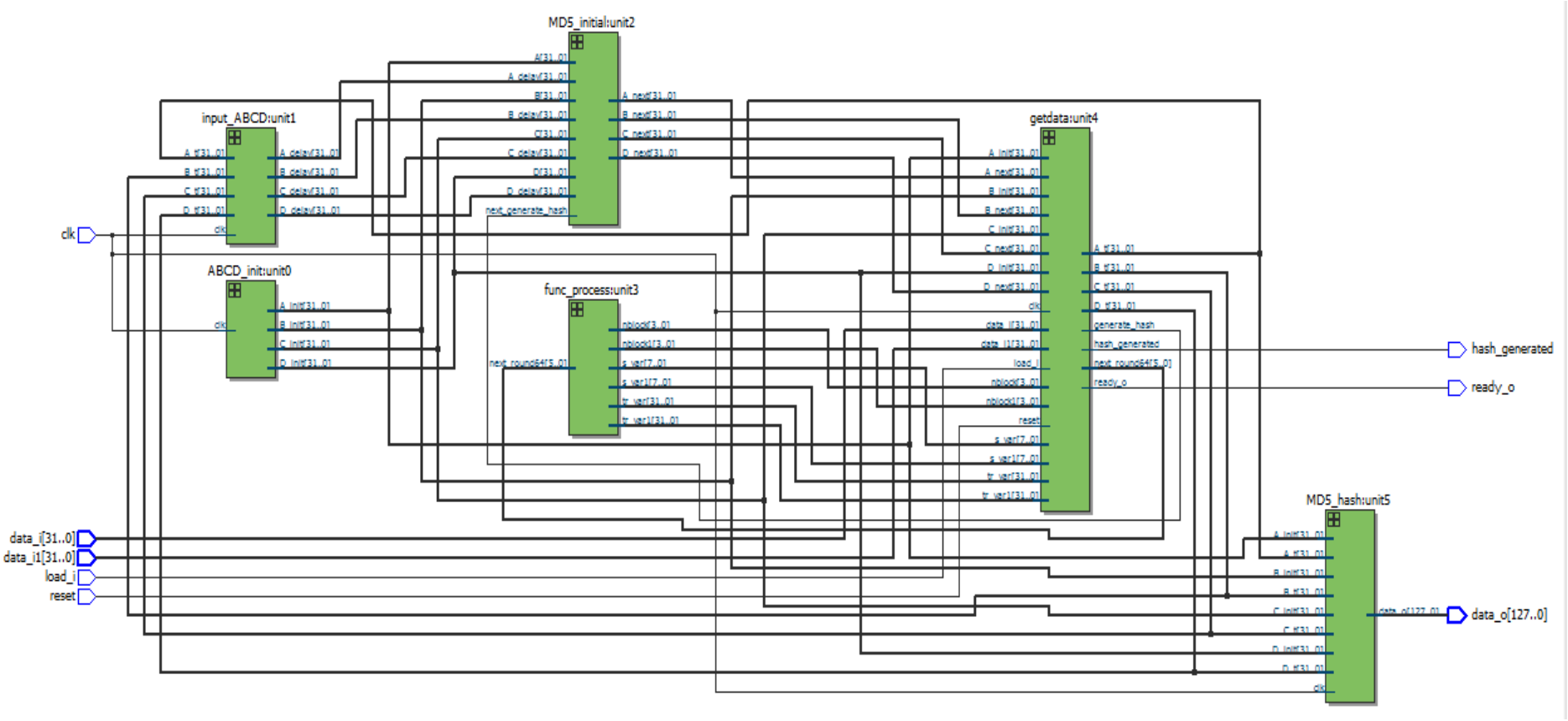


Figure 6: Top-Level Architecture of MD5 Unfolding Design

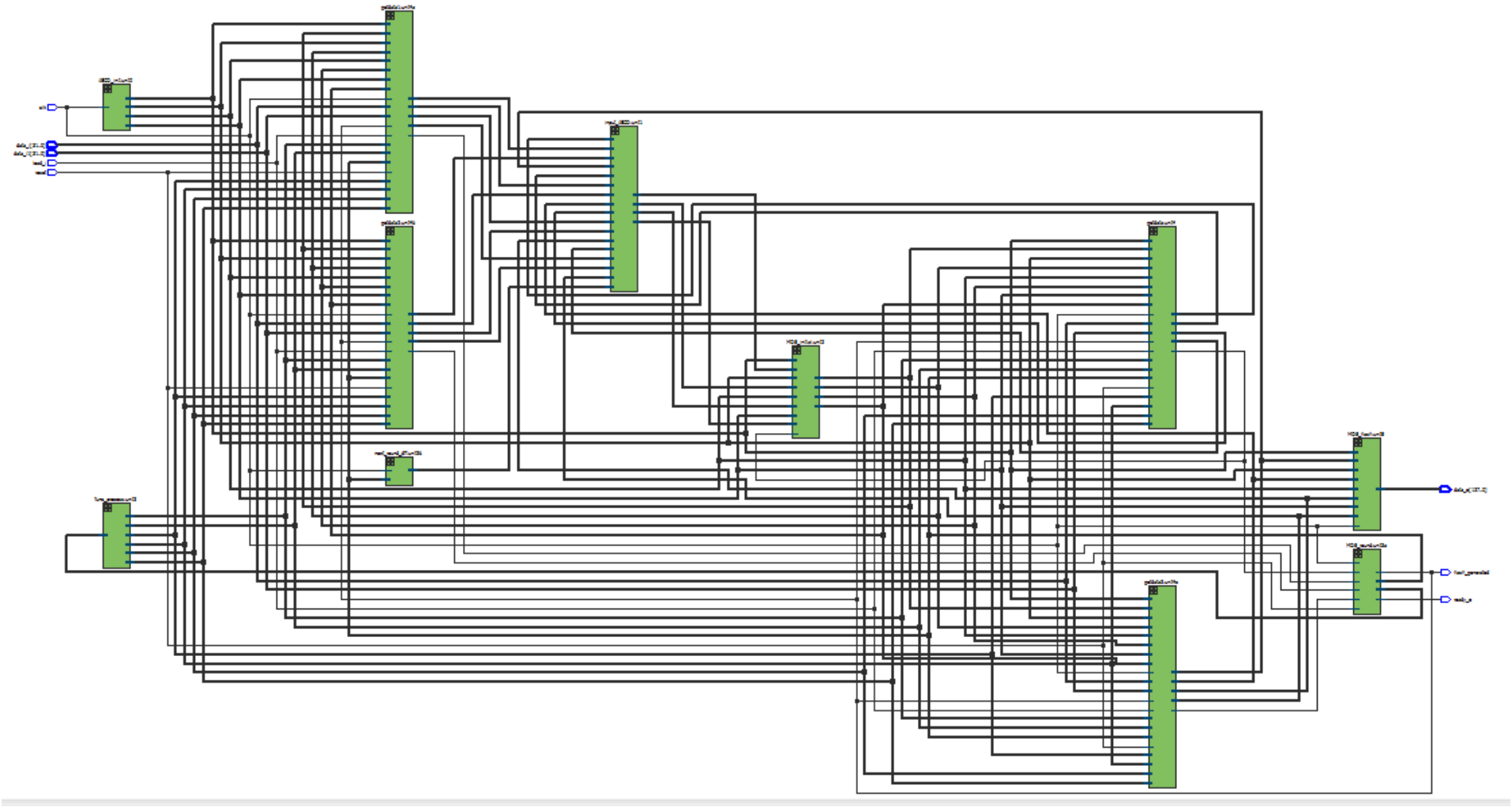


Figure 7: Top-Level Architecture of MD5 Unfolding with Four-stage Pipelining Design

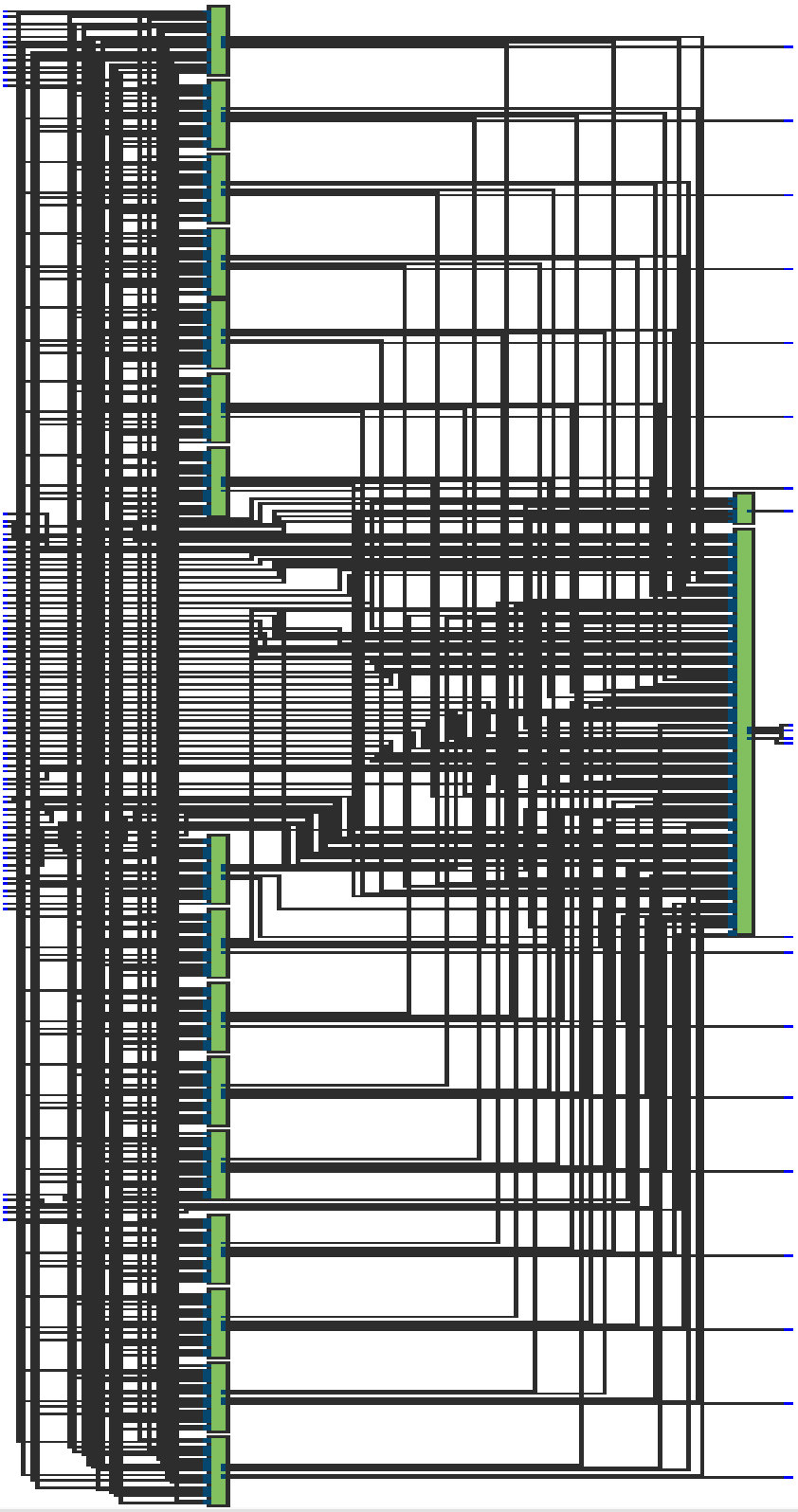


Figure 8: Top-Level Architecture of MD5 Unfolding with 32-stage Pipelining Design

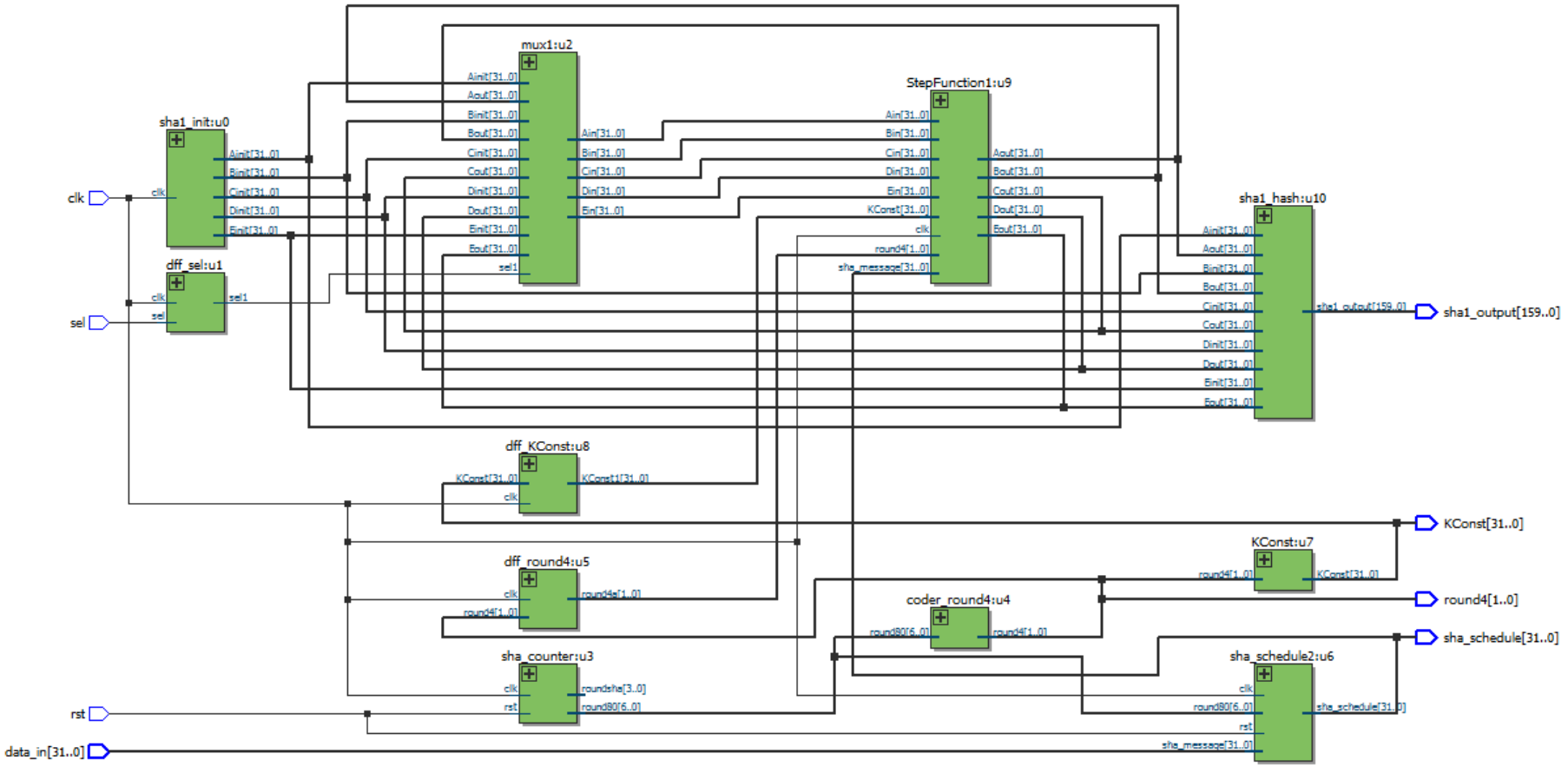


Figure 9: Top-Level Architecture of SHA-1 Design

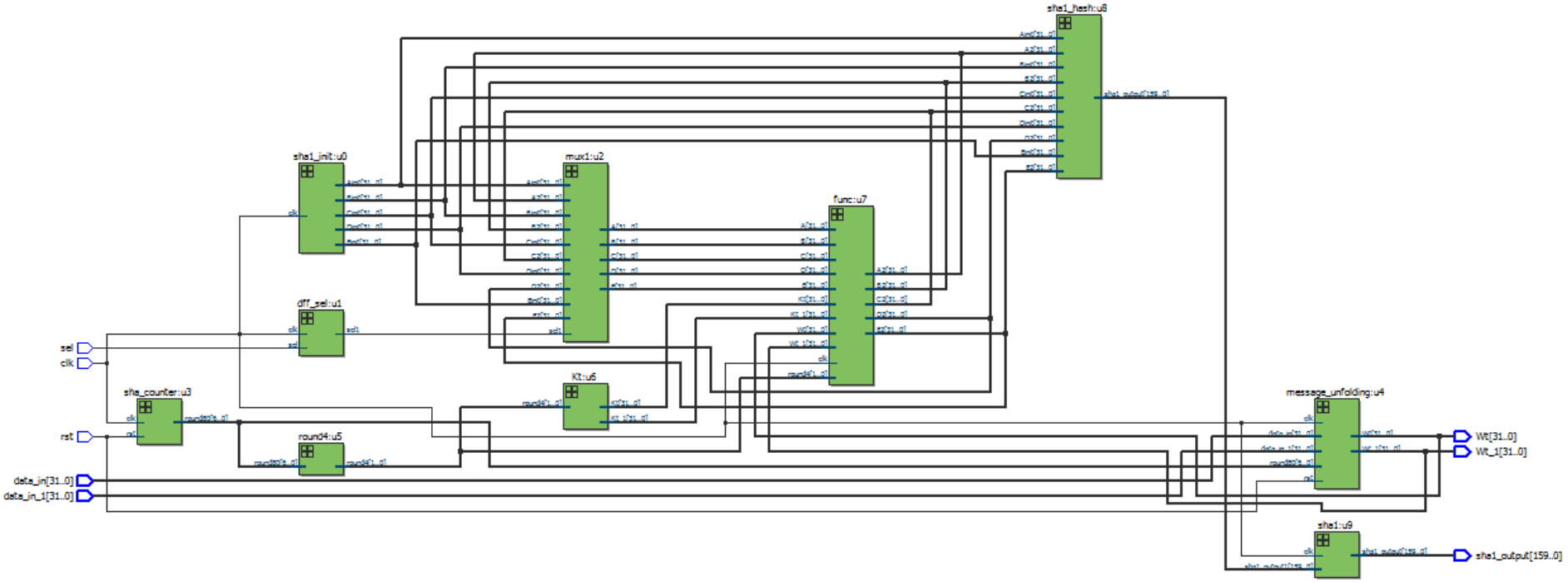


Figure 10: Top-Level Architecture of SHA-1 Unfolding Design

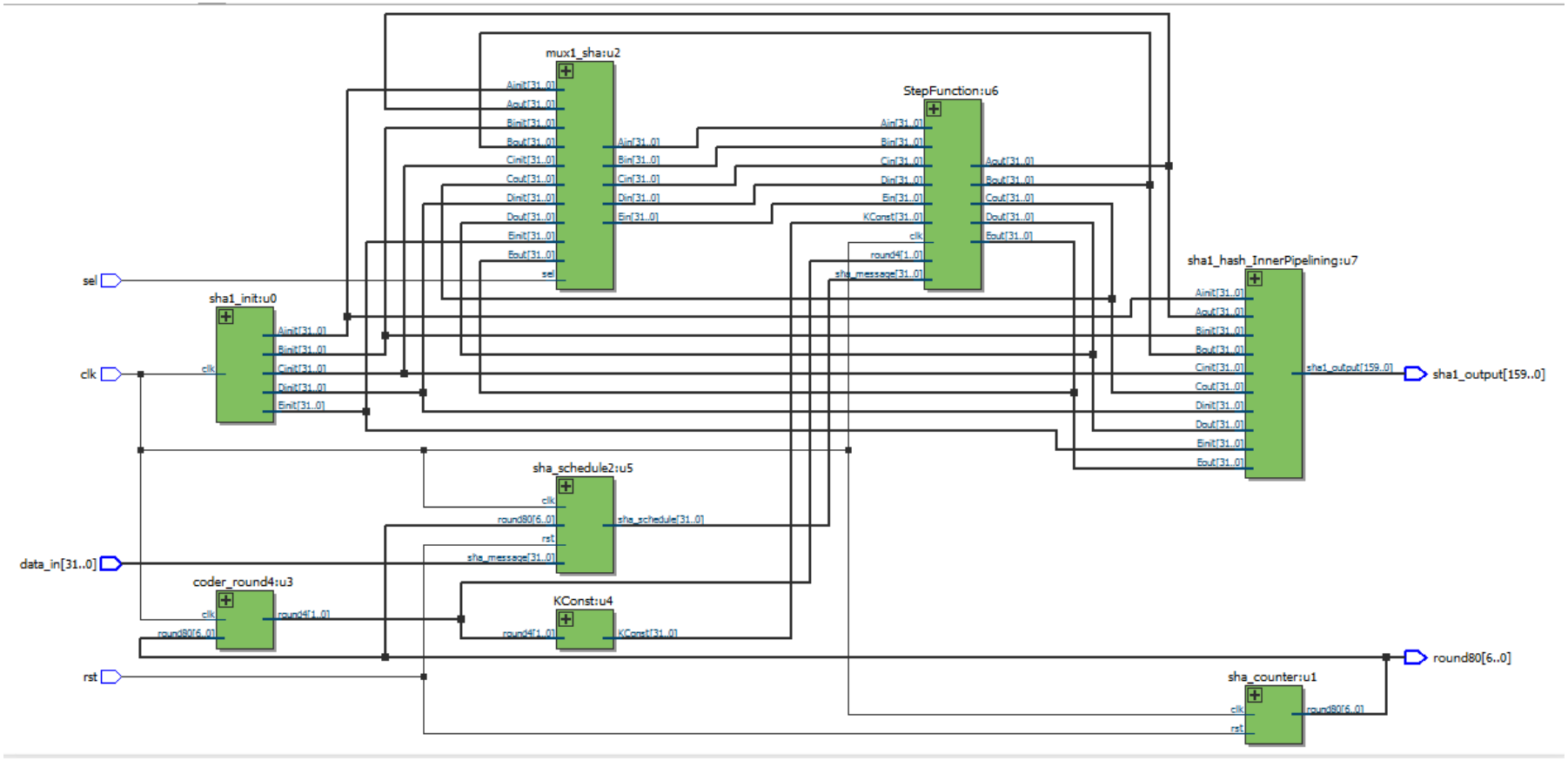


Figure 11: Top-Level Architecture of SHA-1 Inner Pipelining Design

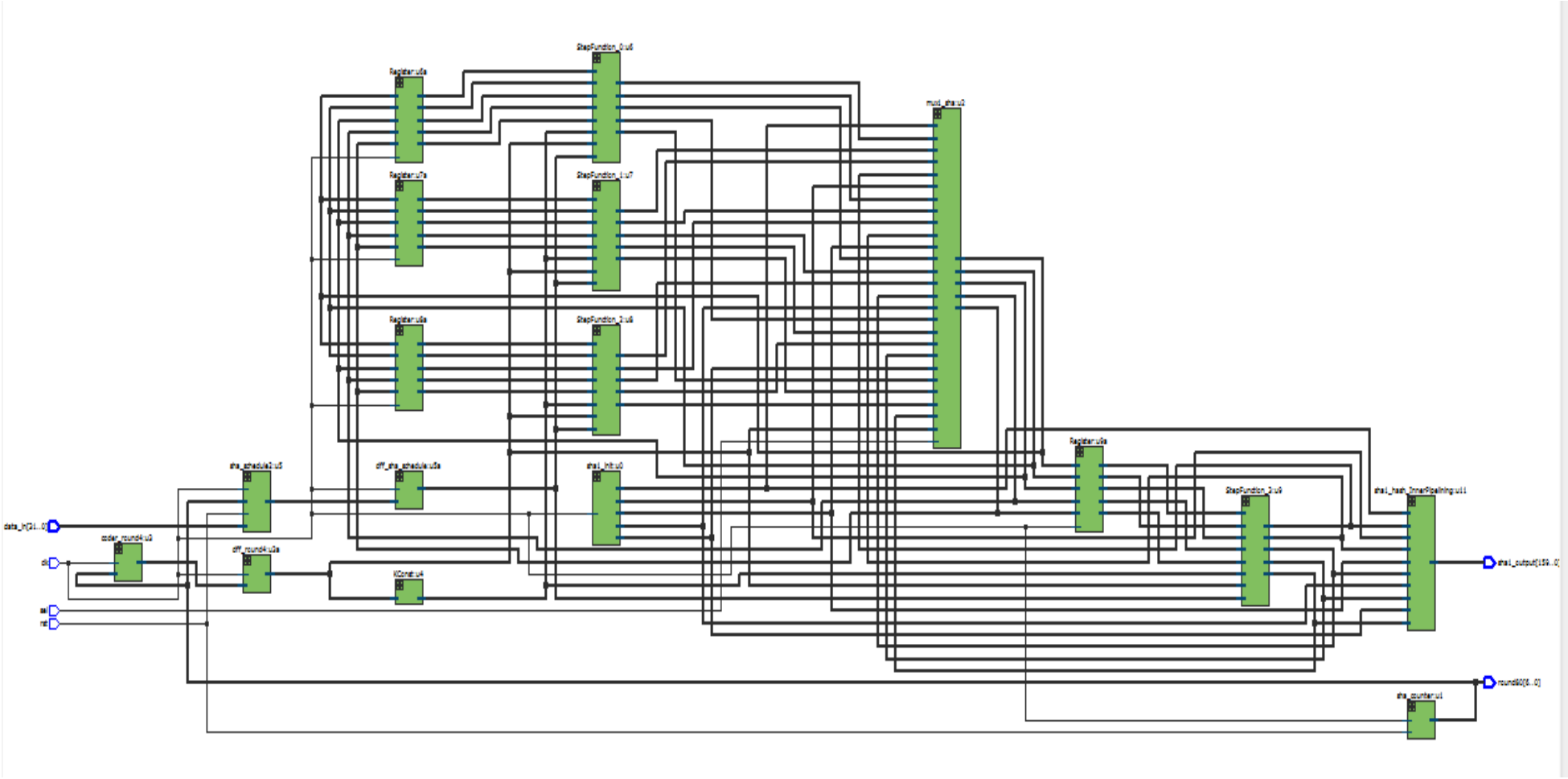


Figure 12: Top-Level Architecture of SHA-1 with Four-stage Pipelining Design

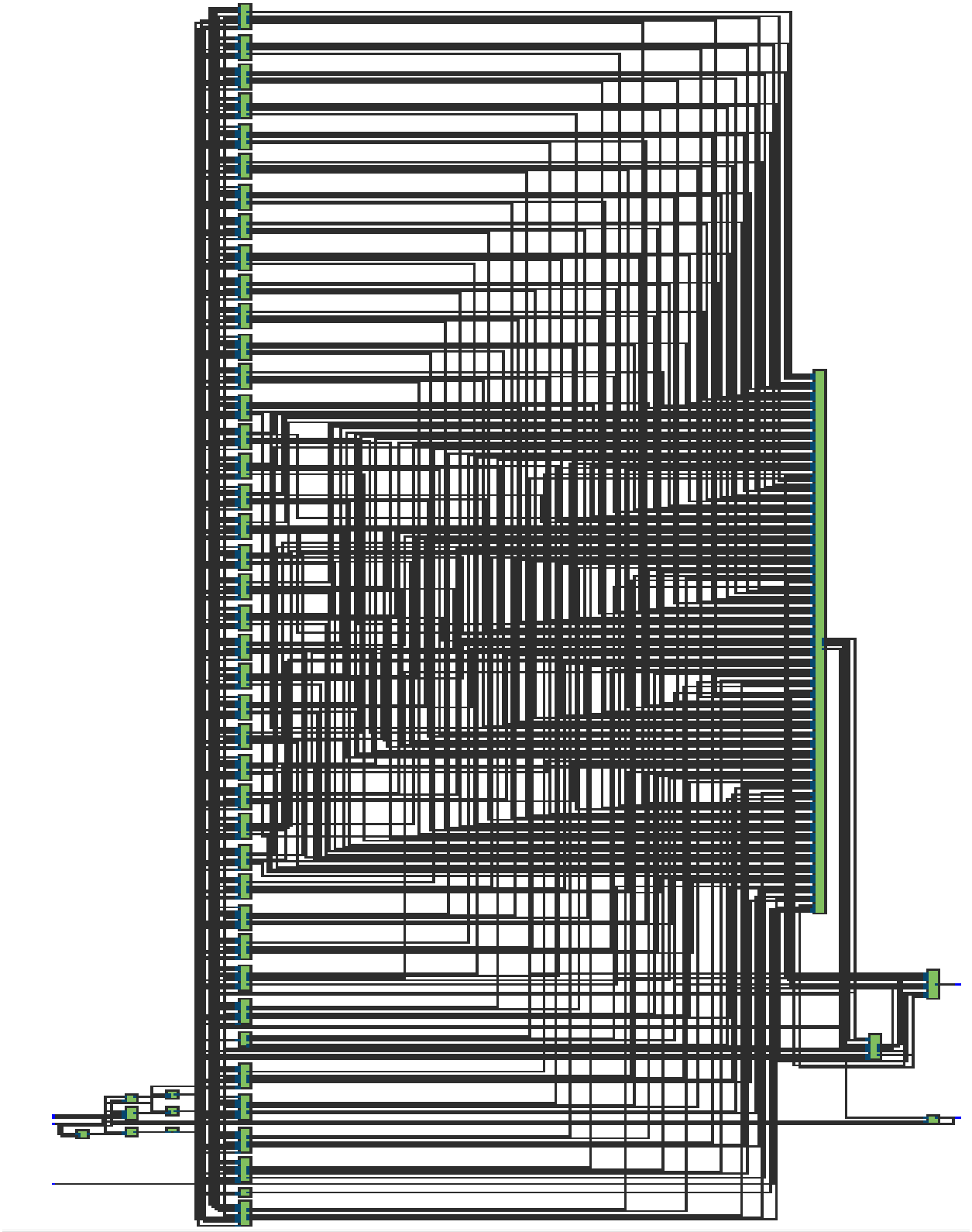


Figure 13 : Top-Level Architecture of SHA-1 with 40-stage Pipelining Design

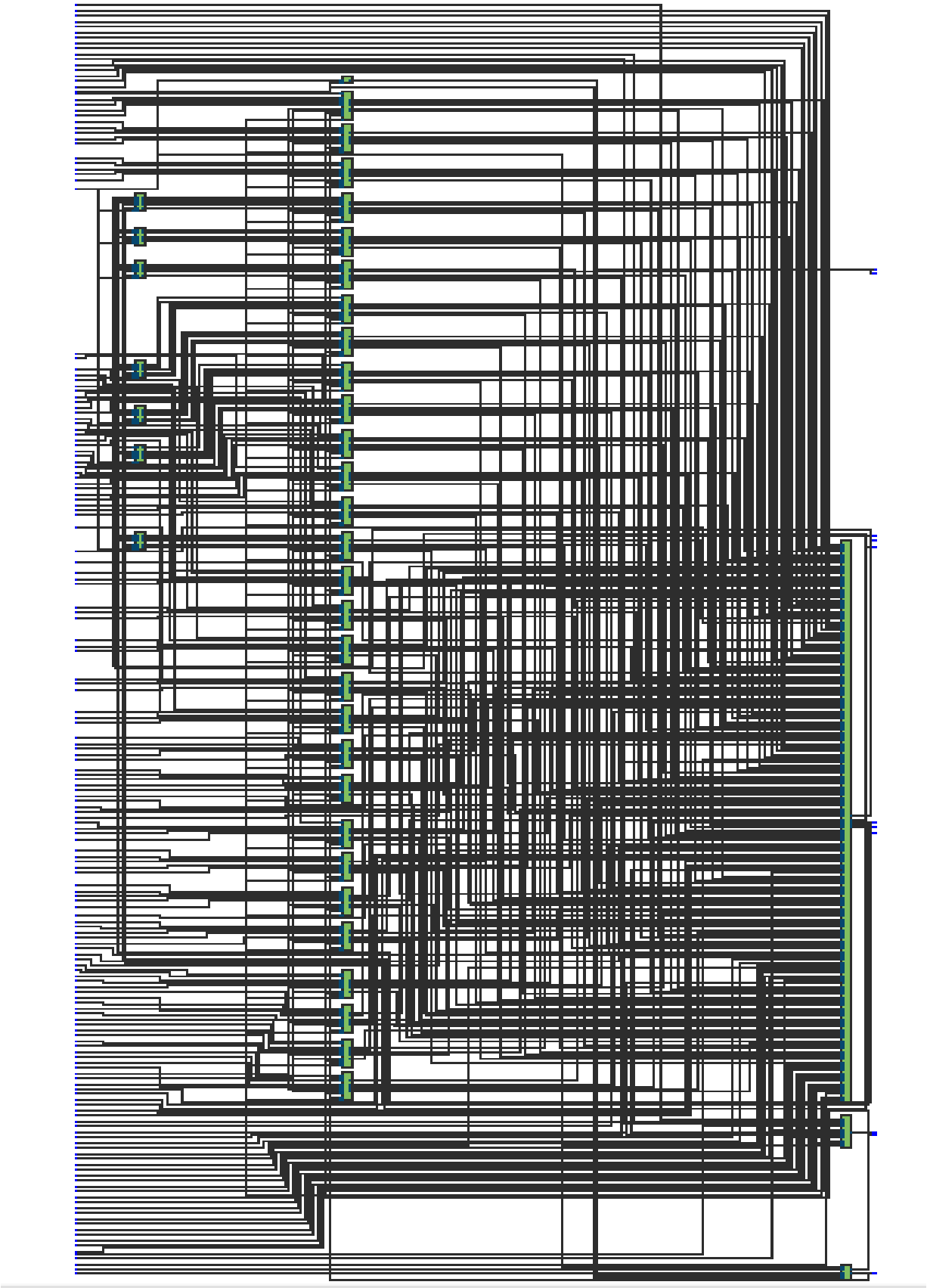


Figure 14: Top-Level Architecture of SHA-1 Unfolding with 40-stage Pipelining Design

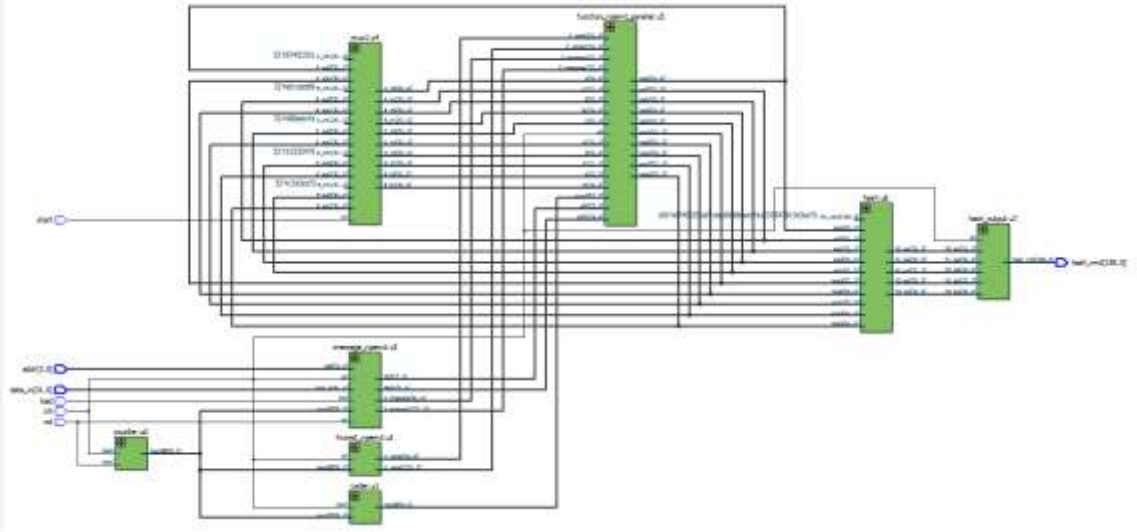


Figure 15: Top-Level Architecture of RIPEMD-160 Design

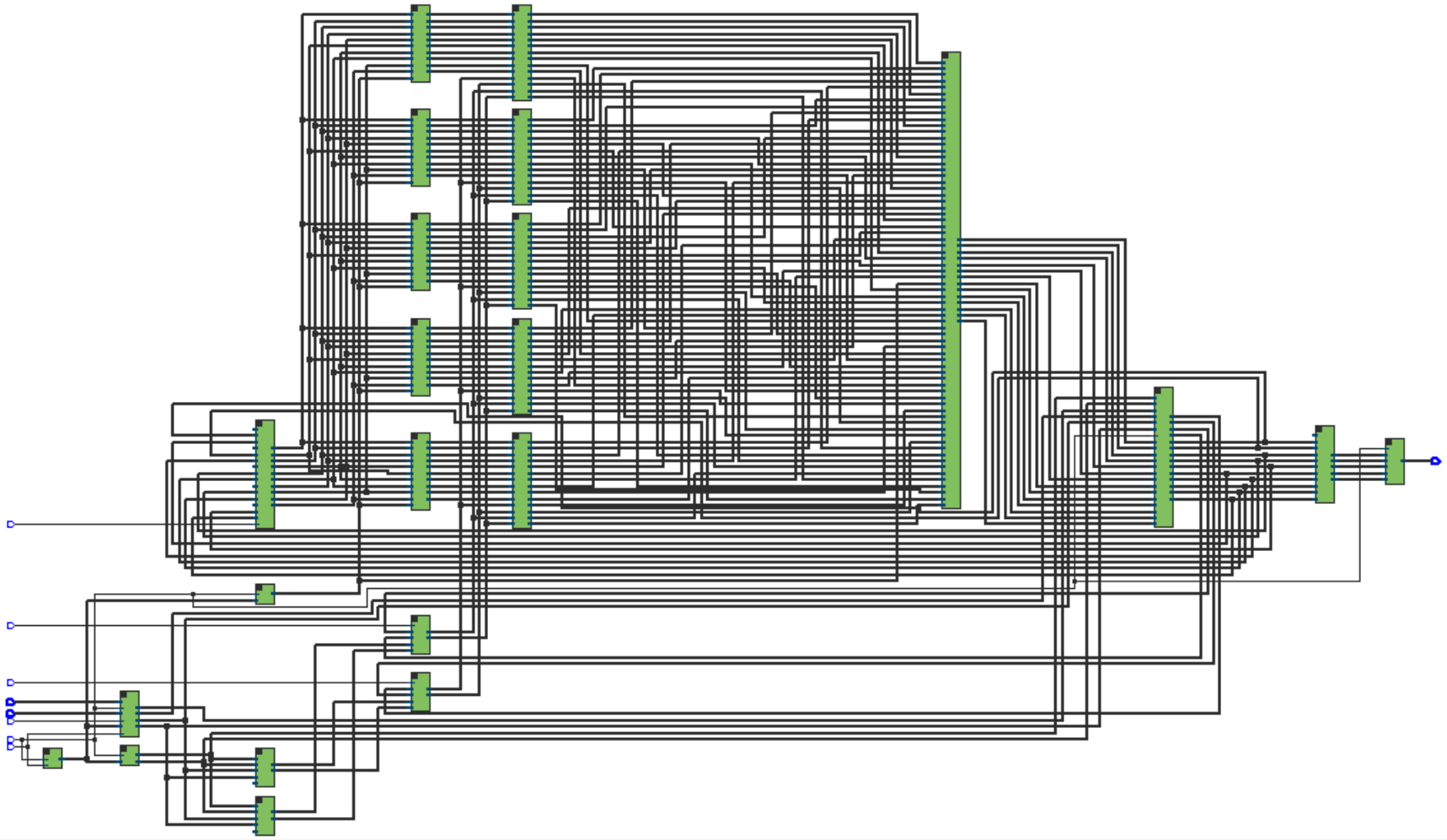


Figure 16: Top-Level Architecture of Pipelined RIPEMD-160 Design

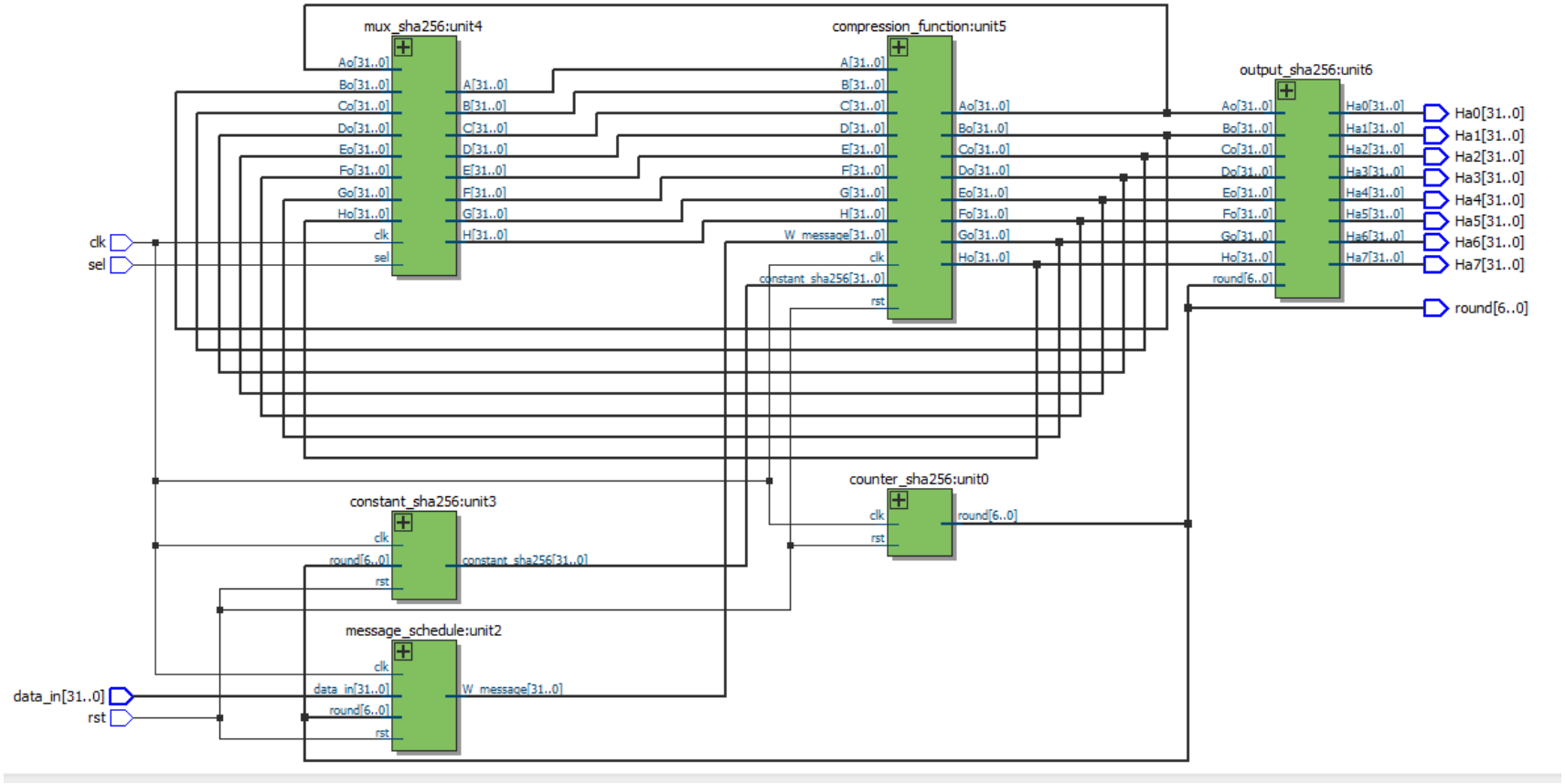


Figure 17: Top-Level Architecture of SHA-256 Design

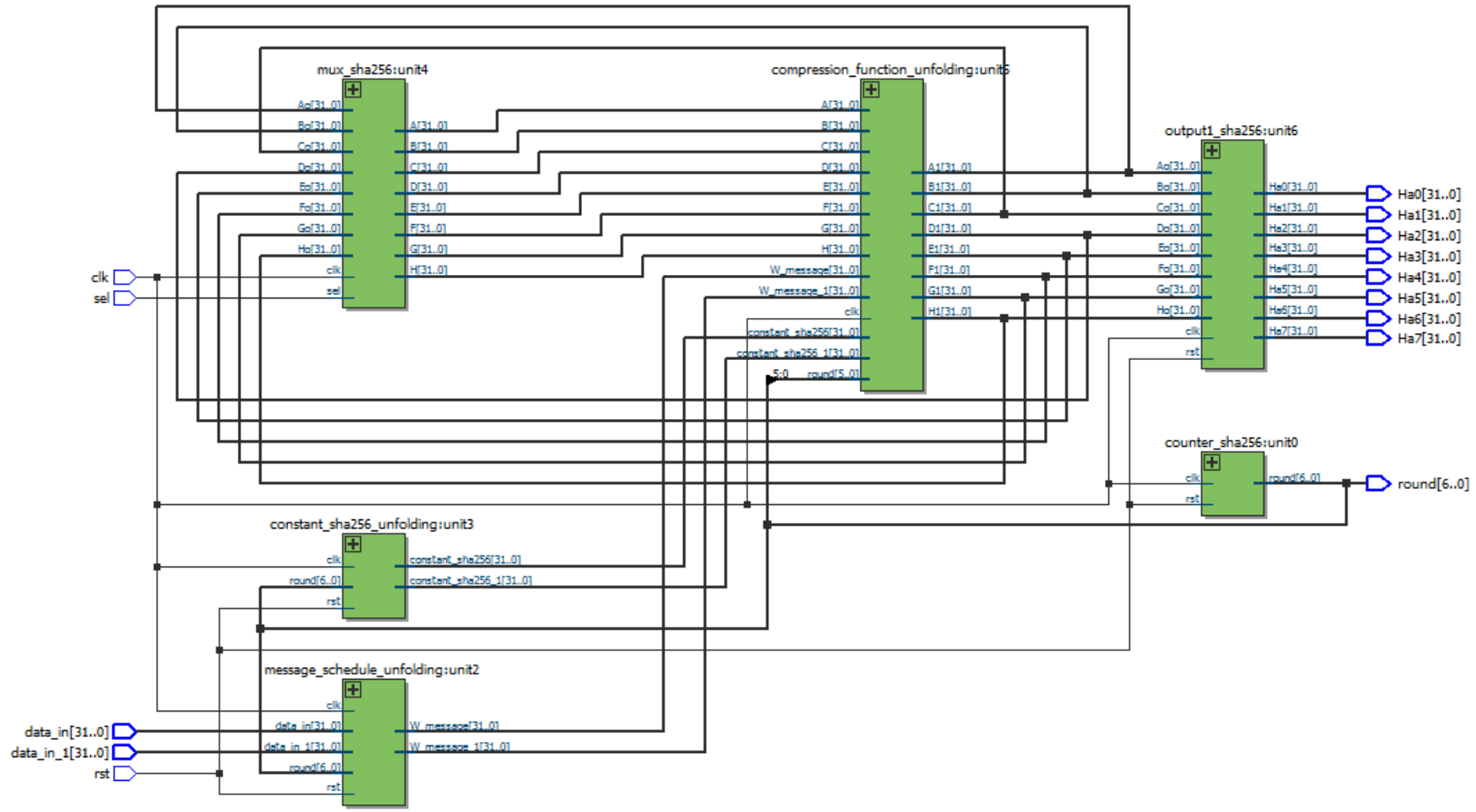


Figure 18: Top-Level Architecture of SHA-256 Unfolding Design

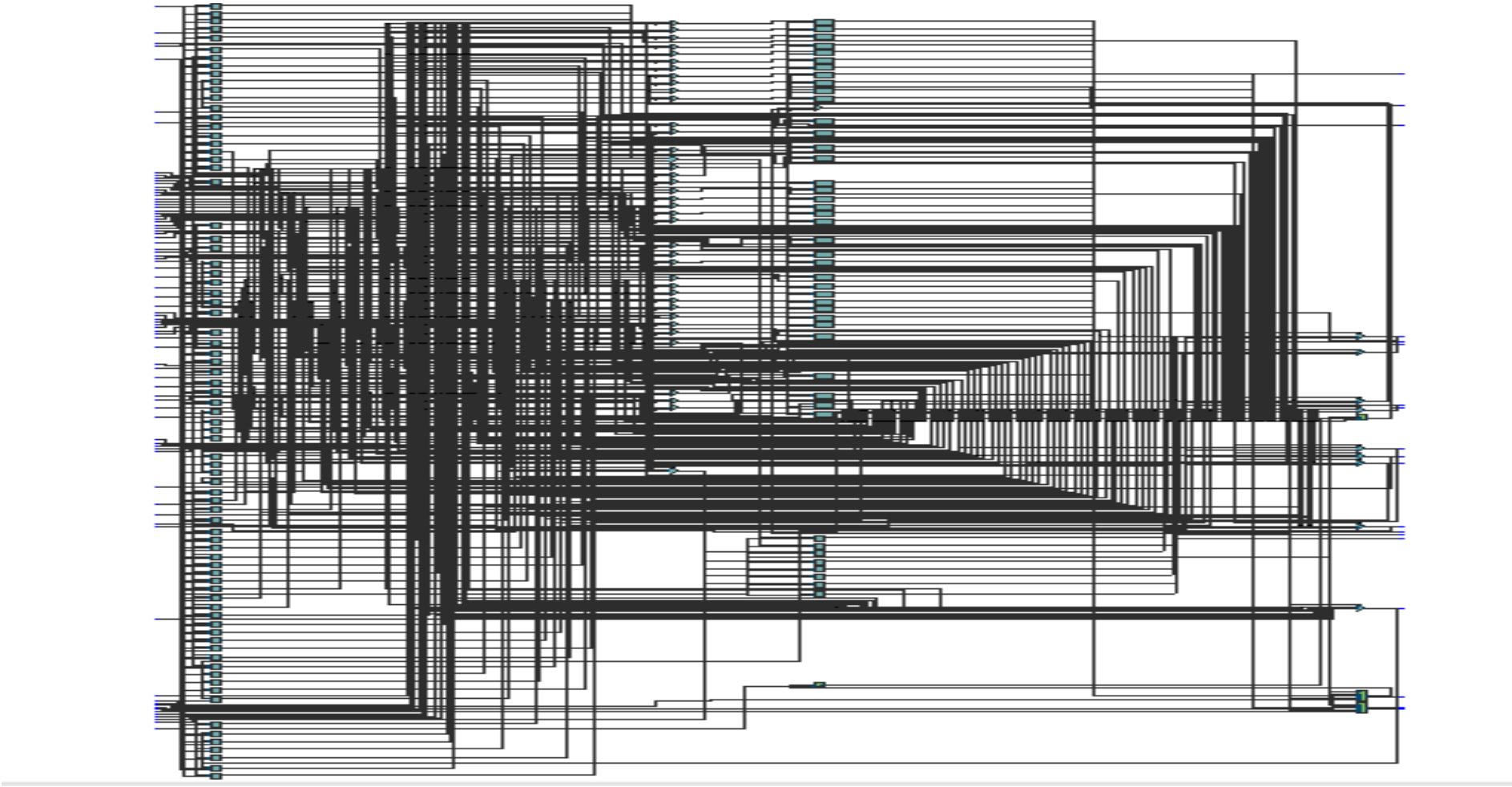


Figure 19: Top-Level Architecture of HMAC-SHA-1 Design