VLSI ARCHITECTURE DESIGN

FOR

MD5 HASHING ALGORITHM

Submitted in partial fulfillment of the Degree of

Bachelor of Technology

May – 2015

BY

Abhinav Soni	(111090)
Anirudh Mehrotra	(111075)
Kushagra Goyal	(111054)

Name of Supervisor – Mr. Akhil Ranjan

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT

CERTIFICATE

This is to certify that project report entitled "**VLSI ARCHITECTURE DESIGN FOR MD5 HASHING ALGORITHM**", submitted by Anirudh Mehrotra (111075), Abhinav Soni (111090), Kushagra Goyal (111054), in partial fulfillment for the award of degree of Bachelor of Technology in Electronics and Communication Engineering to Jaypee University of Information Technology, Waknaghat, Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

Date: **Supervisor's Name: Mr. Akhil Ranjan**

Designation: Assistant Professor (Grade II)

ACKNOWLEDGEMENT

We, the undersigned, warmly acknowledge the continuous encouragement and invaluable suggestions offered by our guide Mr. Akhil Ranjan, Assistant Professor (Grade II), Department of Electronics and Communication, Jaypee University of Information and Technology (Waknaghat) in bringing this project to a successful completion. We would like to express him our sincere appreciation and deepest gratitude for his advice and endless support.

Finally we are grateful to all who are directly or indirectly involved in the successful completion of the project.

Date:

Abhinav Soni

Kushagra Goyal

Anirudh Mehrotra

ABSTRACT

With the increase of the amount of data and users in the information systems, the requirement of data integrity has increased and needs to be improved. One important element in the information system is a key of authentication schemes, which is used as a message authentication code (MAC). One technique to produce a MAC is based on using a hash function and is referred to as a HMAC.

MD5 is one of the most important hash algorithms today. It has been designed by R. Rivest in 1992 and it is considered as a standard in hash function design. Among the many reasons behind the use of MAC is that cryptographic hash functions such as MD5 and SHA-1generally execute faster in software than symmetric block ciphers.

MD5 was designed as a strengthened version of MD4 by increasing number of process rounds after partial attacks on MD4. In 1993 Bert Den Boer and Antoon Bossselaers found pseudo collisions in MD5. In 1996 Dobertin found collisions in compression function of MD5. But these attacks do not affect practical applications.

In this project, the hardware implementation of the MD5 algorithm on reconfigurable devices, is investigated.

Signature of Students	Signature of Supervisor
Kushagra Goyal	Mr. Akhil Ranjan
Anirudh Mehrotra	Date :
Abhinav Soni	
Date :	

TABLE OF CONTENTS

Topics		Page No.
Acknowledgement		ii
Abstract		iii
List of Figures		vi
Listof tables		vii
Chapter 1	Introduction	1
Chapter 2	Hardware Description Language	2
	2.1 VHDL	2
	2.2 VHDL vs. Verilog	2
Chapter 3	Xilinx ISE	7
	3.1 Introduction	7
	3.2 Simulation	7
	3.3 Synthesis	7
	3.4 Spartan Family	8
Chapter 4	Synthesis In VHDL	9
	4.1 Introduction	9
	4.2 Steps involved in Synthesis Process	9
	4.3 VHDL Based Synthesis	12
	4.4 Levels of VHDL based Synthesis	14
	4.5 FPGA Implementation Tools	15
Chapter 5	Simulators	18
	5.1 Introduction	18
	5.2 HDL Simulators	18
Chapter 6	Hash Functions	21
	6.1 Introduction	21
	6.2 Properties	21
	6.3 Encoding vs. Encryption vs. Hashing	23

	6.4	Summary	25
Chapter 7	Message Digest		26
	7.1	Introduction	26
	7.2	Types of Message Digests	26
Chapter 8	Message Digest (MD 5) Algorithm		28
	8.1	Introduction	28
	8.2	Steps involved in MD5 Algorithm	28
Results and Outputs			33
Future Scope			35
Conclusion			38
Applications			39
Appendix			40
References			42

LIST OF FIGURES

FIGURE NO.	FIGURE NAME	PAGE NO.
4.1	VHDL based synthesis	12
4.2	Levels of VHDL based synthesis	14
7.1	Encoding Diagram	26
7.2	Encryption Diagram	26
7.3	Hashing Diagram	27
R.1	512 Bit Padding Output	36
R.2	Final Output	36
R.3	RTL Schematic	37
A.1	Applications	39

LIST OF TABLES

TABLE NO.	TABLE NAME	PAGE NO.
4.1	Translate	16
4.2	Map	16
4.3	Place & Route	16
4.4	Generate Programming File	17

CHAPTER 1: INTRODUCTION

Data integrity assurance and data origin authentication are essential security services in financial transactions, electronic commerce, electronic mail, software distribution, data storage and so on. The broadest definition of authentication within computing systems encompasses identity verification, message origin authentication and message content authentication. In IPSEC, the technique of cryptographic hash functions is utilized to achieve these security services.

A **cryptographic hash function** is a hash function which is considered practically impossible to invert, that is, to recreate the input data from its hash value alone.

The ideal cryptographic hash function has four main properties:

- It is easy to compute the hash value for any given message
- It is infeasible to generate a message that has a given hash
- It is infeasible to modify a message without changing the hash
- It is infeasible to find two different messages with the same hash.

A cryptographic hash function must be able to withstand all known types of cryptanalytic attack. At a minimum, it must have the following properties:

- Pre-image resistance

  Given a hash h it should be difficult to find any message m such that $h = hash(m)$. This concept is related to that of one-way function. Functions that lack this property are vulnerable to pre-image attacks.

- Second pre-image resistance

  Given an input m_1 it should be difficult to find another input m_2 such that $m_1 \neq m_2$ and $hash(m_1) = hash(m_2)$. Functions that lack this property are vulnerable to second-pre-image attacks.

- Collision resistance

  It should be difficult to find two different messages m_1 and m_2 such that $hash(m_1) = hash(m_2)$. Such a pair is called a cryptographic hash collision.

CHAPTER 2: HARDWARE DESCRIPTION LANGUAGE

2.1 VHDL

(**VHSIC Hardware Description Language**) is a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate array and integrated circuits. VHDL can also be used as a general purpose parallel programming language.

Advantages

- The key advantage of VHDL, when used for systems design, is that it allows the behavior of the required system to be described (modeled) and verified (simulated) before synthesis tools translate the design into real hardware (gates and wires).
- Another benefit is that VHDL allows the description of a concurrent system. VHDL is a dataflow language, unlike procedural computing languages such as BASIC, C, and assembly code, which all run sequentially, one instruction at a time.
- A VHDL project is multipurpose. Being created once, a calculation block can be used in many other projects. However, many formational and functional block parameters can be tuned (capacity parameters, memory size, element base, block composition and interconnection structure).
- A VHDL project is portable. Being created for one element base, a computing device project can be ported on another element base, for example VLSI with various technologies.

2.2 VHDL vs VERILOG

- ***Capability***

When modeling abstract hardware, the capability of VHDL can sometimes only be achieved in Verilog when using the PLI. The choice of which to use is not therefore based solely on technical capability but on:

- personal preferences
- EDA tool availability

- commercial, business and marketing issues

The modeling constructs of VHDL and Verilog cover a slightly different spectrum across the levels of behavioral abstraction.

- ***Compilation***

  VHDL. Multiple design-units (entity/architecture pairs), that reside in the same system file, may be separately compiled if so desired. However, it is good design practice to keep each design unit in its own system file in which case separate compilation should not be an issue.

  Verilog. The Verilog language is still rooted in it's native interpretative mode. Compilation is a means of speeding up simulation, but has not changed the original nature of the language. As a result care must be taken with both the compilation order of code written in a single file and the compilation order of multiple files. Simulation results can change by simply changing the order of compilation.

- ***Data types***

  VHDL. A variety of language or user defined data types can be used. Dedicated conversion functions are needed to convert objects from one type to another. The choice of which data types to use should be considered wisely, especially enumerated (abstract) data types. It makes models easier to write, clearer to read and avoid unnecessary conversion functions that can clutter the code.

  Verilog. Compared to VHDL, Verilog data types are very simple, easy to use and very much geared towards modeling hardware structure as opposed to abstract hardware modeling. Unlike VHDL, all data types used in a Verilog model are defined by the Verilog language and not by the user. There are net data types, for example wire, and a register data type called reg. A model with a signal whose type is one of the net data types has a corresponding electrical wire in the implied modeled circuit. Objects that are signals, of type reg hold their value over simulation delta cycles and should not be confused with the modeling of a hardware register. Verilog may be preferred because of its simplicity.

- ***Design reusability***

  VHDL. Procedures and functions may be placed in a package so that they are available to any design-unit that wishes to use them.

  Verilog. There is no concept of packages in Verilog. Functions and procedures used within a model must be defined in the module. To make functions and procedures generally accessible from different module statements the functions and procedures must be placed in a separate system file and included using the `include compiler directive.

- ***Easiest to Learn***

  Starting with zero knowledge of either language, Verilog is probably the easiest to grasp and understand. This assumes the Verilog compiler directive language for simulation and the PLI language is not included. If these languages are included they can be looked upon as two additional languages that need to be learned. VHDL may seem less intuitive at first for two primary reasons. First, it is very strongly typed; a feature that makes it robust and powerful for the advanced user after a longer learning phase. Second, there are many ways to model the same circuit, especially those with large hierarchical structures.

- ***Forward and back annotation***

  A spin-off from Verilog is the Standard Delay Format (SDF). This is a general purpose format used to define the timing delays in a circuit. The format provides a bidirectional link between, chip layout tools, and either synthesis or simulation tools, in order to provide more accurate timing representations. The SDF format is now an industry standard in it's own right.

- ***High level constructs***

  VHDL. There are more constructs and features for high-level modeling in VHDL than there are in Verilog. Abstract data types can be used along with the following statements:

  * Package statements for model reuse,
  * Configuration statements for configuring design structure,
  * Generate statements for replicating structure,
  * Generic statements for generic models that can be individually characterized, for example, bit width.

Verilog. Except for being able to parameterize models by overloading parameter constants, there is no equivalent to the high-level VHDL modeling statements in Verilog.

- ***Language Extensions***

  The use of language extensions will make a model non standard and most likely not portable across other design tools. However, sometimes they are necessary in order to achieve the desired results.

  VHDL. Has an attribute called 'foreign that allows architectures and subprograms to be modeled in another language.

  Verilog. The Programming Language Interface (PLI) is an interface mechanism between Verilog models and Verilog software tools. For example, a designer, or more likely, a Verilog tool vendor, can specify user defined tasks or functions in the C programming language, and then call them from the Verilog source description. Use of such tasks or functions make a Verilog model nonstandard and so may not be usable by other Verilog tools. Their use is not recommended.

- ***Libraries***

  VHDL. A library is a store for compiled entities, architectures, packages and configurations. Useful for managing multiple design projects.

  Verilog. There is no concept of a library in Verilog. This is due to it's origins as an interpretive language.

- ***Low Level Constructs***

  VHDL. Simple two input logical operators are built into the language, they are: NOT, AND, OR, NAND, NOR, XOR and XNOR. Any timing must be separately specified using the after clause. Separate constructs defined under the VITAL language must be used to define the cell primitives of ASIC and FPGA libraries.

  Verilog. The Verilog language was originally developed with gate level modeling in mind, and so has very good constructs for modeling at this level and for modeling the cell primitives of ASIC and FPGA libraries. Examples include User Defined Primitive s (UDP), truth tables and the specify block for specifying timing delays across a module.

- ***Managing large designs***

  VHDL. Configuration, generate, generic and package statements all help manage large design structures.

  Verilog. There are no statements in Verilog that help manage large designs.

- ***Operators***

  The majority of operators are the same between the two languages. Verilog does have very useful unary reduction operators that are not in VHDL. A loop statement can be used in VHDL to perform the same operation as a Verilog unary reduction operator. VHDL has the mod operator that is not found in Verilog.

- ***Parameterizable models***

  VHDL. A specific bit width model can be instantiated from a generic n-bit model using the generic statement. The generic model will not synthesize until it is instantiated and the value of the generic given.

  Verilog. A specific width model can be instantiated from a generic n-bit model using overloaded parameter values. The generic model must have a default parameter value defined. This means two things. In the absence of an overloaded value being specified, it will still synthesize, but will use the specified default parameter value. Also, it does not need to be instantiated with an overloaded parameter value specified, before it will synthesize.

- ***Procedures and tasks***

  VHDL allows concurrent procedure calls; Verilog does not allow concurrent task calls.

- ***Readability***

  This is more a matter of coding style and experience than language feature. VHDL is a concise and verbose language; its roots are based on Ada. Verilog is more like C because it's constructs are based approximately 50% on C and 50% on Ada. For this reason an existing C programmer may prefer Verilog over VHDL.

- ***Structural replication***

  VHDL. The *generate* statement replicates a number of instances of the same design-unit or some sub part of a design, and connects it appropriately.

  Verilog. There is no equivalent to the *generate* statement in Verilog.

CHAPTER 3: XILINX ISE

3.1 Introduction

Xilinx designs, develops and markets programmable logic products, including integrated circuits (ICs), software design tools, predefined system functions delivered as intellectual property (IP) cores, design services, customer training, field engineering and technical support. Xilinx sells both FPGAs and CPLDs for electronic equipment manufacturers in end markets such as communications, industrial, consumer, automotive and data processing.R1oss Freeman, Bernard Vonderschmitt, and James V Barnett II, who all had worked for integrated circuit and solid-state device manufacturer Zilog Corp, founded Xilinx in 1984.

The Xilinx ISE is a design environment for FPGA products from Xilinx, and is tightly-coupled to the architecture of such chips, and cannot be used with FPGA products from other vendors. The Xilinx ISE is primarily used for circuit synthesis and design, while the ModelSim logic simulator is used for system-level testing.

3.2 Simulation

System-level testing may be performed with the ModelSim logic simulator, and such test programs must also be written in HDL languages. Test bench programs may include simulated input signal waveforms, or monitors which observe and verify the outputs of the device under test.

ModelSim may be used to perform the following types of simulations:

- Logical verification, to ensure the module produces expected results
- Behavioural verification, to verify logical and timing issues
- Post-place & route simulation, to verify behaviour after placement of the module within the reconfigurable logic of the FPGA

3.3 Synthesis

Xilinx's patented algorithms for synthesis allow designs to run upto 30% faster than competing programs, and allows greater logic density which reduces project costs.

Also, due to the increasing complexity of FPGA fabric, including memory blocks and I/O blocks, more complex synthesis algorithms were developed that separate unrelated modules into *slices*, reducing post-placement errors.

IP Cores are offered by Xilinx and other third-party vendors, to implement system-level functions such as digital signal processing (DSP), bus interfaces, networking protocols, image processing, embedded processors, and peripherals. Xilinx has been instrumental in shifting designs from ASIC-based implementation to FPGA-based implementation.

3.4 Spartan family

The Spartan series targets applications with a low-power footprint, extreme cost sensitivity and high-volume; e.g. displays, set-top boxes, wireless routers and other applications.

The Spartan-6 family is built on a 45-nanometer [nm], 9-metal layer, dual-oxide process technology. The Spartan-6 was marketed in 2009 as a low-cost solution for automotive, wireless communications, flat-panel display and video surveillance applications.

CHAPTER 4: SYNTHESIS IN VHDL

4.1 Introduction

Synthesis is a general term that describes the process of transformation of the model of a design, usually described in a hardware description language (HDL), from one level of behavioral abstraction to a lower, more detailed behavioral level.

These transformations try to improve upon a set of objective metrics (e.g., area, speed, power dissipation) of a design, while satisfying a set of constraints (e.g., I/O rates, MIPS, sample period) imposed on it.

Synthesis in its most generic form simply refers to the incorporation of additional lower level implementation details into a digital design, however, most current tools expect the output to be a net list of gates that are optimized for area, power, or latency

4.2 Steps Involved in Synthesis Process

Synthesis to the gate-level of abstraction consists of three major steps –

- Design specification (in a machine-readable form)
- Design implementation
- Design validation and verification

Design specification implies a description of the design where functional, performance, and cost constraints (area, power, speed) are captured in a form that facilitates processing (I.e., executable) in a CAD environment.

4.2.1 Methods for capturing the specification are:

- Graphical methods
- Language-based methods

Many consider graphical methods and language methods to be indistinguishable, in that one can be quickly converted to another. It is often the designer’s choice as to which mechanism they find convenient when specifying a complex digital design.

4.2.1.1 Graphical Methods

These include:

- Block diagrams, state diagrams, schematics, data flow graphs, timing diagrams, truth tables, etc.

Advantages:

- Faster learning curve
- Intuitive documentation and
- Reusability of design.

Disadvantages:

- Limited support for functional and timing hierarchy in complex digital systems
- Limited support for multiple views (e.g., simulation versus hardware generation)

4.2.1.2 Language-based Methods

These methods include:

- Hardware description languages (HDLs), high level algorithmic macro descriptors (such as Matlab), or application-specific languages (such as Silage and DSP/C for signal processing).

Advantages:

- Ability to serve as a standard medium of communication between the algorithm developer and the synthesis expert
- Robustness through well-defined semantics
- Ability for representing complex behavior and data structures
- Support for synthesis and its verification in the same environment,
- Their system-independent nature that facilitates design documentation.

Disadvantages:

- Specification is harder to visualize, and requires a longer learning curve.

4.2.2 Intermediate Steps involved in Design Implementation

- Algorithm design:
  - Evaluates the performance of the algorithm being implemented in terms of:
    - The precision of the word length,
    - The number of iterations and
    - The quality of the results obtained with respect to the design requirements.
- Behavioral simulations:
  - Done after the algorithm design is complete to verify the detailed functional behavior of the system.
- Data and control flow graph generation:
  - Derives an intermediate representation of the behavioral specification where the data flow, input/output, control dependencies and synchronization signals are documented.
- The result of the previous steps in synthesis is sometimes called a behavioral-level model of the circuit.
- The next steps involve a knowledge of the hardware and software modules available in the design library. At this stage of the synthesis process, based on the control/data flow graph, the steps are:
  - Module selection (determining which modules are used in design)
  - Estimation and allocation of the number of modules to be used
  - Transformations that optimize the control/data flow graph without changing the input/output functional properties, but improve on its synthesis metrics (such as area, power, or speed).
- Scheduling determines when a certain operation will be executed, and assignment determines the module on which the operand will be executed (e.g., an addition on an adder module).
- At the completion of this stage of the synthesis the result is called a register-transfer level (RTL) model of the circuit
- The RTL description or model is then converted to a logic level implementation through a process called logic synthesis, and after further technology level optimization a gate-level model of the circuit results, which must be verified for both timing and function.

Validation at each stage of the design is a feature of the top-down structured design process, and can proceed in a bottom-up or topdown manner

- E.g., in a bottom-up process each selected module is tested to ensure correct functional behavior. After all the constituent modules are validated the entire design is validated at the RTL level.
- After completion of the logic synthesis, the verification process is carried out at the gate level.

4.3 VHDL-Based Synthesis

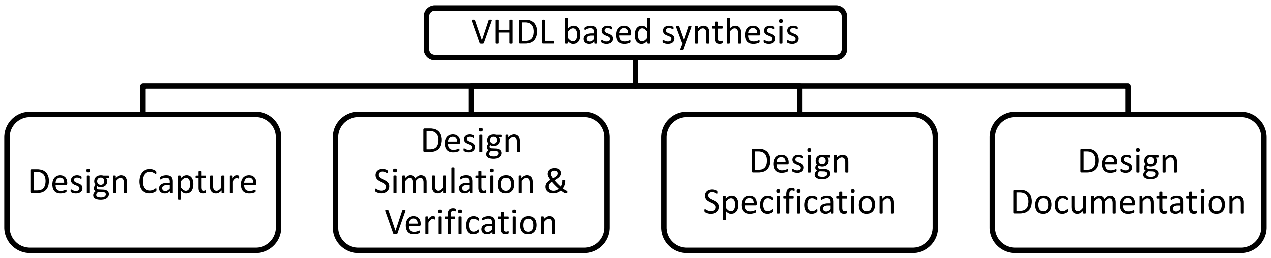


FIGURE: 4.1

Synthesis is a process that is independent of VHDL. However, VHDL assists synthesis through its four major roles listed below.

- In Design Capture, one already has a digital design in mind, and this design is translated to VHDL so that it can be processed directly by a synthesis tools.
- In Design Simulation & Verification, a test bench and a captured design are simulated to ensure that the design works correctly.
- In Design Specification, arguably the most powerful use of VHDL, the design can be specified at a very high level of abstraction, and the synthesis tool can then take this description and translate it to lower levels of design abstraction subject to the enforced constraints.
- In Design Documentation one can use VHDL as an executable version of the system that has been designed or is under design.

4.3.1 Design Capture

- VHDL allows designer to capture the details of a digital design in a form that is machine-readable, that is, the design can be entered into a CAD system.
- VHDL can be used as an alternative to, or in conjunction, with schematic-based design entry methods.
- Typical logic synthesis and RTL synthesis tools are primarily design capture environments. Over 90% of designers use VHDL for this purpose (EE Times, 1996).

4.3.2 Design Specification

- Within a structured design flow, VHDL can be used to capture the behavioral, interface and performance-related aspects of the design.
- The VHDL representation can then be synthesized to an implementation through several consecutive stages, each incrementally adding more detail to it.
- Behavioral synthesis and advanced RTL synthesis tools is representative of environments performing this function.

4.3.3 Design Simulation and Verification

- VHDL used to simulate key properties of a design prior to and after synthesis.
- VHDL can verify those properties at various levels of the synthesis process.
- Various levels of functional and timing simulation can include both technology dependent and independent aspects of the design under synthesis

4.3.4 Design Documentation

- A design represented in VHDL is a well documented design, and is accepted as such by the US Department of Defense.
- VHDL, an IEEE standard, allows designs to be represented in a non-proprietary, technology -independent manner.
- VHDL is capable of representing both the design and its test environment in a tool-environment, technology-neutral manner, adding greatly to design productivity, reuse, and capability for its rapid upgrade.

4.4 Levels of VHDL-based Synthesis

Commonly used synthesis tools support

- Behavioral Synthesis
- RTL Synthesis
- Logic Synthesis

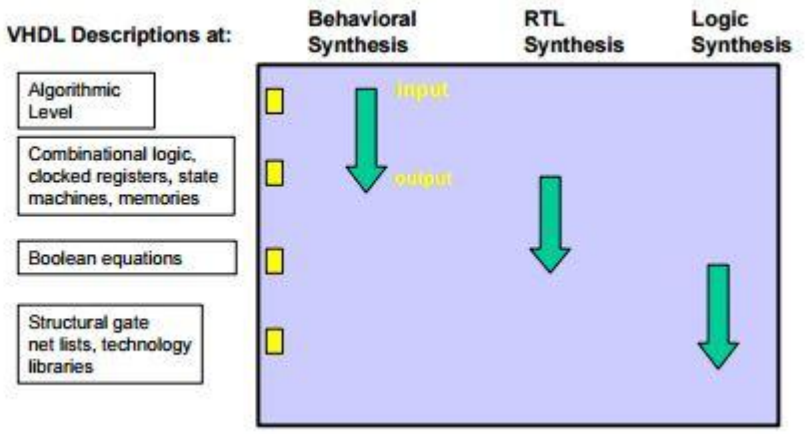


FIGURE: 4.2

Characteristics of Behavioral Synthesis

- Pros
    - User inputs behavioral-level VHDL code that is short, easy to write, and verify, leading to increased productivity
    - The synthesis tools perform the tasks of resource allocation, assignment, scheduling, and optimize area, latency, and power dissipation based on user input
    - Output from a behavioral synthesis tool can be starting point for RTL synthesis tools for further optimization.
- Cons
    - Supports a very small subset of VHDL (e.g., single process architectures).
    - Non-standard implementations depending on tool vendors
    - Currently supports application specific areas (such as DSP) with primarily loop-dominated computational flow graphs
    - Ability to handle larger graphs (more than a few dozen operations) limited ° Needs supporting high level libraries of components

Characteristics of RTL Synthesis

- Pros
  - Allows capture of a digital design at the RTL level in VHDL - improving productivity over logic synthesis tools
  - Allows manual mapping to libraries of high-level components (multipliers, adders)
  - More control over the synthesis process in terms of final architecture
  - Provide several templates for VHDL semantics for state machine optimization
  - IEEE RTL VHDL standard, 1997
  - Supports a large subset of VHDL
- Cons
  - Up until 1997, each vendor supported a different RTL subset of VHDL
  - Requires specification of the datapath, registers, controller, and cycle-bycycle behavior
  - Resource sharing, resource allocation, scheduling, and mapping tasks have to be carried out by the designer prior to coding at the RTL level, limiting architectural exploration.
  - Allows no architectural exploration, and the synthesizer optimizes at the level of the components and states

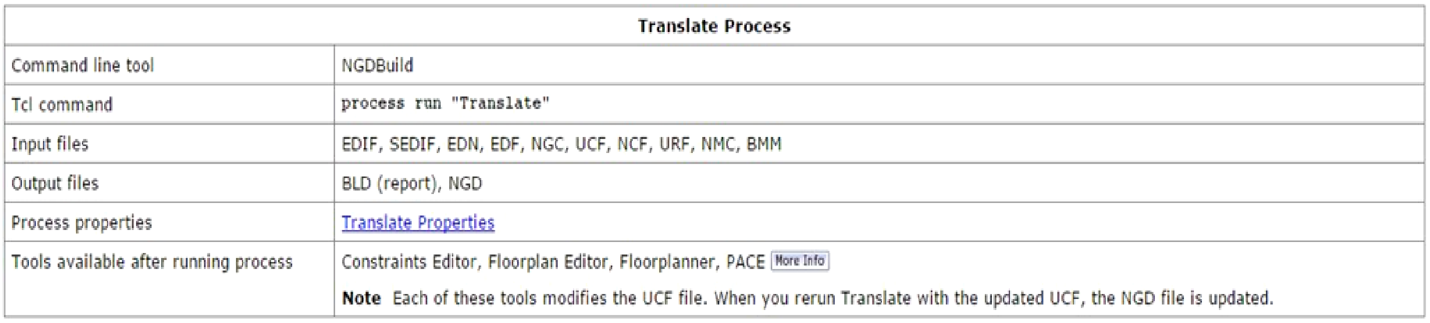
4.5 FPGA Implementation Tools

After synthesis, run design implementation, which comprises the following steps:

1. Translate, which merges the incoming netlists and constraints into a Xilinx® design file
2. Map, which fits the design into the available resources on the target device
3. Place and Route, which places and routes the design to the timing constraints
4. Programming file generation, which creates a bitstream file that can be downloaded to the device

- **Translate**

  The Translate process merges all of the input netlists and design constraints and outputs a xilinx native generic database (NGD) file, which describes the logical design reduced to Xilinx primitives.

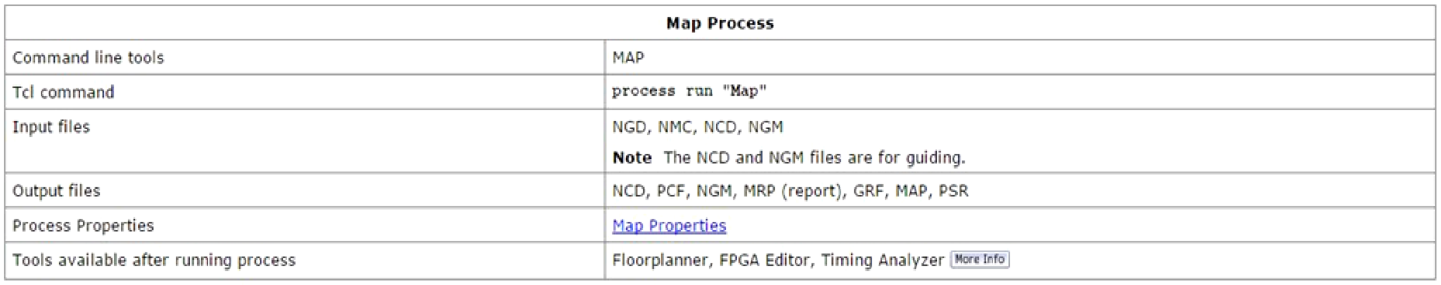


Translate Process	
Command line tool	NGDBuild
Tcl command	`process run "Translate"`
Input files	EDIF, SEDIF, EDN, EDF, NGC, UCF, NCF, URF, NMC, BMM
Output files	BLD (report), NGD
Process properties	Translate Properties
Tools available after running process	Constraints Editor, Floorplan Editor, Floorplanner, PACE More Info **Note** Each of these tools modifies the UCF file. When you rerun Translate with the updated UCF, the NGD file is updated.

TABLE: 4.1

- **Map**

The Map process maps the logic defined by an NGD file into FPGA elements, such as CLBs and IOBs. The output design is a native circuit description (NCD) file that physically represents the design mapped to the components in the Xilinx FPGA.

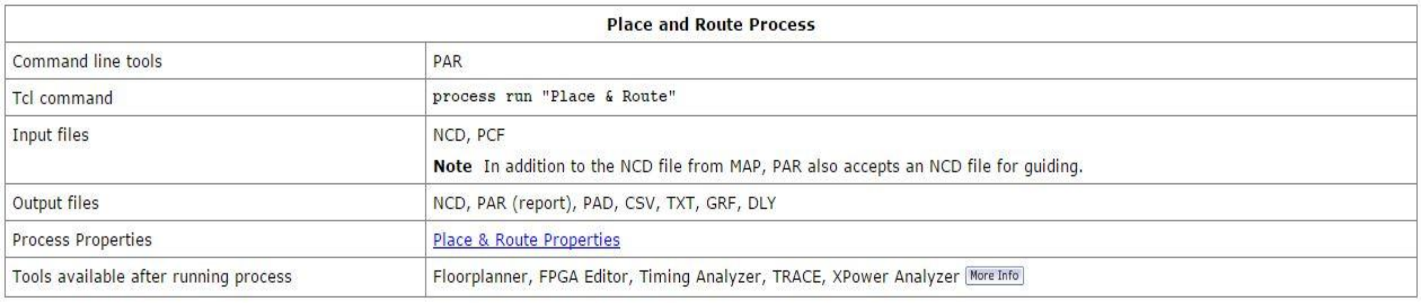


Map Process	
Command line tools	MAP
Tcl command	`process run "Map"`
Input files	NGD, NMC, NCD, NGM **Note** The NCD and NGM files are for guiding.
Output files	NCD, PCF, NGM, MRP (report), GRF, MAP, PSR
Process Properties	Map Properties
Tools available after running process	Floorplanner, FPGA Editor, Timing Analyzer More Info

TABLE: 4.2

- **Place and Route**

The Place and Route process takes a mapped NCD file, places and routes the design, and produces an NCD file that is used as input for bitstream generation.

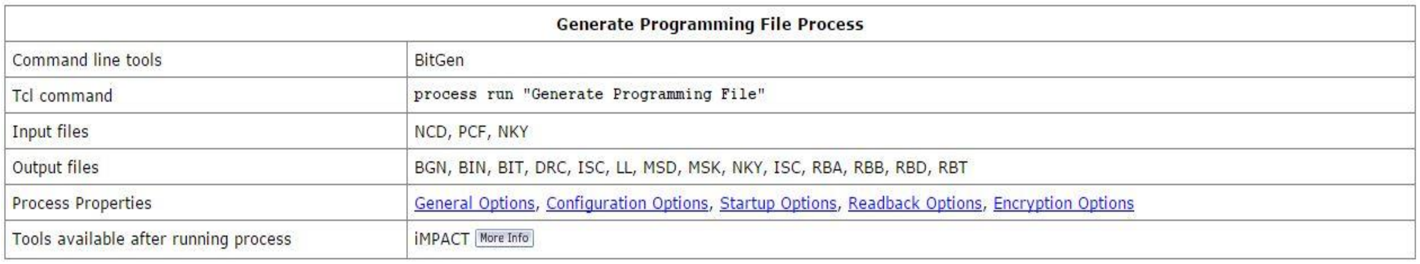


Place and Route Process	
Command line tools	PAR
Tcl command	`process run "Place & Route"`
Input files	NCD, PCF **Note** In addition to the NCD file from MAP, PAR also accepts an NCD file for guiding.
Output files	NCD, PAR (report), PAD, CSV, TXT, GRF, DLY
Process Properties	Place & Route Properties
Tools available after running process	Floorplanner, FPGA Editor, Timing Analyzer, TRACE, XPower Analyzer More Info

TABLE: 4.3

- **Programming File Generation**

  The Generate Programming File process produces a bitstream for Xilinx device configuration. After the design is completely routed, you must configure the device so it can execute the desired function.



Generate Programming File Process	
Command line tools	BitGen
Tcl command	`process run "Generate Programming File"`
Input files	NCD, PCF, NKY
Output files	BGN, BIN, BIT, DRC, ISC, LL, MSD, MSK, NKY, ISC, RBA, RBB, RBD, RBT
Process Properties	General Options, Configuration Options, Startup Options, Readback Options, Encryption Options
Tools available after running process	iMPACT More Info

TABLE: 4.4

Chapter 5: SIMUILATORS

5.1 Introduction

Simulation is the imitation of the operation of a real-world process or system over time. The act of simulating something first requires that a model be developed; this model represents the key characteristics or behaviors/functions of the selected physical or abstract system or process. The model represents the system itself, whereas the simulation represents the operation of the system over time.

Simulation is used in many contexts, such as simulation of technology for performance optimization, safety engineering, testing, training, education, and video games. Often, computer experiments are used to study simulation models. Simulation is also used with scientific modeling of natural systems or human systems to gain insight into their functioning. Simulation can be used to show the eventual real effects of alternative conditions and courses of action. Simulation is also used when the real system cannot be engaged, because it may not be accessible, or it may be dangerous or unacceptable to engage, or it is being designed but not yet built, or it may simply not exist.

5.2 HDL Simulators

HDL simulators are software packages that emulate any hardware description language. HDL simulation software has come a long way since its early origin as a single proprietary product offered by one company. Today, Simulators are available from many vendors, at all price points. For desktop/personal use, Aldec, Mentor, LogicSim, SynaptiCAD,TarangEDA and others offer <$5000 USD tool-suites for the Windows 2000/XP platform. The suites bundle the simulator engine with a complete development environment: text editor, waveform viewer, and RTL-level browser. Additionally, limited-functionality editions of the Aldec and ModelSim simulator are downloadable free of charge, from their respective OEM partners (Actel, Altera, Lattice Semiconductor, Xilinx, etc.) For those desiring open-source software, there is Icarus Verilog,GHDL among others. Here, we have used Modelsim SE/EE 5.4a simulator by Mentor Graphics to simulate our project.

5.2.1 ModelSim

ModelSim is a multi-language HDL simulation environment by Mentor Graphics, for simulation of hardware description languages such as VHDL, Verilog and SystemC, and includes a built-in C debugger. ModelSim can be used independently, or in conjunction with Altera Quartus or Xilinx ISE. Simulation is performed using the graphical user interface (GUI), or automatically using scripts.

ModelSim is offered in multiple editions, such as ModelSim PE, ModelSim SE, and ModelSim XE.

- ModelSim SE offers high-performance and advanced debugging capabilities, while ModelSim PE is the entry-level simulator for hobbyists and students. ModelSim SE is used in large multi-million gate designs, and is supported on Microsoft Windows and Linux, in 32-bit and 64-bit architectures.
- ModelSim XE stands for Xilinx Edition, and is specially designed for integration with Xilinx ISE. ModelSim XE enables testing of HDL programs written for Xilinx Virtex/Spartan series FPGA's without needed physical hardware.

ModelSim can also be used with MATLAB/Simulink, using Link for ModelSim. Link for ModelSim is a fast bidirectional co-simulation interface between Simulink and ModelSim. For such designs, MATLAB provides a numerical simulation toolset, while ModelSim provides tools to verify the hardware implementation & timing characteristics of the design. ModelSim uses a unified kernel for simulation of all supported languages, and the method of debugging embedded C code is the same as VHDL or Verilog.

ModelSim enables simulation, verification and debugging for the following languages:

- VHDL
- Verilog
- Verilog 2001
- SystemVerilog
- PSL
- SystemC

5.2.2 ISE Simulator (ISim)

ISim provides a complete, full-featured HDL simulator integrated within ISE. HDL simulation now can be an even more fundamental step within your design flow with the tight integration of the ISim within your design environment.

ISim Key Features:

- Mixed language support
- Supports VHDL-93 and Verilog 2001
- Native support for all HardIP blocks
- PPC, MGT, PCIe, etc.
- No special license requirements
- Supports AXI Bus Functional Model (BFM)
- Multi-Threaded compilation
- Post-Processing capabilities
- Tcl scriptable GUI and batch mode simulation run
- Standalone Waveform viewing capabilities
- Debug capabilities
- Waveform tracing, waveform viewing, HDL source debugging
- Power Analysis and optimization using SAIF
- Memory Editor for viewing and debugging memory elements
- Single click re-compile and re-launch of simulation
- Integrated with ISE Design Suite and Plan Ahead application
- Easy to use - One-click compilation and simulation
- Hardware Co-simulation capability
- Offload a design or a portion of the design to hardware
- Accelerate RTL simulation by up to 50x
- Xilinx simulation libraries “built-in”
- Additional mapping or compilation not required

CHAPTER 6: HASH FUNCTION

6.1 Introduction

Hash functions compress a string of arbitrary length to a string of fixed length. They provide a unique relationship between the input and the hash value and hence replace the authenticity of a large amount of information (message) by the authenticity of a much smaller hash value (authenticator).

In recent years there has been an increased interest in developing a Message Authentication Code (MAC) derived from a hash code. Among the many reasons behind this are that cryptographic hash functions such as MD5 and SHA-1 generally execute faster in software than symmetric block ciphers such as DES.

6.2 PROPERTIES

Good hash functions, in the original sense of the term, are usually required to satisfy certain properties listed below. The exact requirements are dependent on the application

- **Determinism**

A hash procedure must be deterministic—meaning that for a given input value it must always generate the same hash value. In other words, it must be a function of the data to be hashed, in the mathematical sense of the term.

- **Uniformity**

A good hash function should map the expected inputs as evenly as possible over its output range. That is, every hash value in the output range should be generated with roughly the same probability.

The reason for this last requirement is that the cost of hashing-based methods goes up sharply as the number of collisions—pairs of inputs that are mapped to the same hash value—increases. Basically, if some hash values are more likely to occur than others, a larger fraction of the lookup operations will have to search through a larger set of colliding table entries.

- **Defined range**

It is often desirable that the output of a hash function have fixed size.

Producing fixed-length output from variable length input can be accomplished by breaking the input data into chunks of specific size. Hash functions used for data searches use some arithmetic expression which iteratively processes chunks of the input (such as the characters in a string) to produce the hash value.

In cryptographic hash functions, these chunks are processed by a one-way compression function, with the last chunk being padded if necessary. In this case, their size, which is called *block size*, is much bigger than the size of the hash value.

- **Variable range**

In many applications, the range of hash values may be different for each run of the program, or may change along the same run. In those situations, one needs a hash function which takes two parameters—the input data *z*, and the number *n* of allowed hash values.

- **Data normalization**

In some applications, the input data may contain features that are irrelevant for comparison purposes. For example, when looking up a personal name, it may be desirable to ignore the distinction between upper and lower case letters. For such data, one must use a hash function that is compatible with the data equivalence criterion being used: that is, any two inputs that are considered equivalent must yield the same hash value. This can be accomplished by normalizing the input before hashing it, as by upper-casing all letters.

- **Non-invertible**

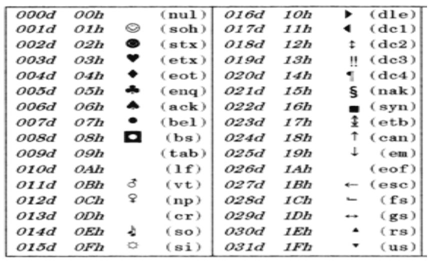
In cryptographic applications, hash functions are typically expected to be non-invertible, meaning that it is not possible to reconstruct the input datum x from its hash value $h(x)$ alone without spending great amounts of computing time

6.3 Encoding vs. Encryption vs. Hashing

Hashing is often confused with encryption and encoding. They are not the same. But before going into the differences, it's mentioned how they are related:

1. All three transform data into another format.
2. Both encoding and encryption are reversible, and hashing is not.

Encoding



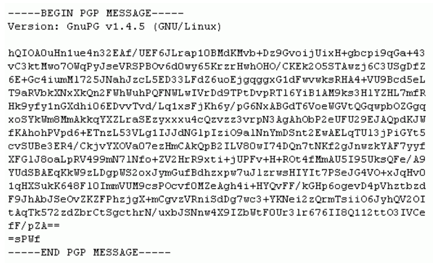
000d	00h		(nul)	016d	10h	►	(dle)
001d	01h	☺	(soh)	017d	11h	◄	(dc1)
002d	02h	☻	(stx)	018d	12h	↕	(dc2)
003d	03h	♥	(etx)	019d	13h	‼	(dc3)
004d	04h	♦	(eot)	020d	14h	¶	(dc4)
005d	05h	♣	(enq)	021d	15h	§	(nak)
006d	06h	♠	(ack)	022d	16h	▬	(syn)
007d	07h	•	(bel)	023d	17h	↨	(etb)
008d	08h	◘	(bs)	024d	18h	↑	(can)
009d	09h		(tab)	025d	19h	↓	(em)
010d	0Ah		(lf)	026d	1Ah		(eof)
011d	0Bh	♂	(vt)	027d	1Bh	←	(esc)
012d	0Ch	♀	(np)	028d	1Ch	∟	(fs)
013d	0Dh		(cr)	029d	1Dh	↔	(gs)
014d	0Eh	♫	(so)	030d	1Eh	▲	(rs)
015d	0Fh	☼	(si)	031d	1Fh	▼	(us)

Figure 7.1

The purpose of encoding is to transform data so that it can be properly (and safely) consumed by a different type of system, e.g. binary data being sent over email, or viewing special characters on a web page. The goal is **not** to keep information secret, but rather to ensure that it's able to be properly consumed.

Encoding transforms data into another format using a scheme that is publicly available so that it can easily be reversed. It does not require a key as the only thing required to decode it is the algorithm that was used to encode it.

Encryption



```
-----BEGIN PGP MESSAGE-----
Version: GnuPG v1.4.5 (GNU/Linux)

hQIOAOuHn1ue4n32EAf/UEF6JLrap1OBMdKMvb+Dz9GvoijUixH+gbcpi9qGa+43
vC3ktMwo7OWqPyJseVRSPBOv6dOwy65KrzrHwhOHO/CKEk2O5STAwzj6C3USgDfZ
6E+Gc4iumMl725JNahJzcL5ED33LFdZ6uoEjgqggxG1dFwvwksRHA4+VU9Bcd5eL
T9aRVbkXNxXkQn2FWhWuhPQFNWLwIVrDd9TPtDvpRTl6YiB1AM9ks3HlYZHL7mfR
Hk9yfy1nGXdhiO6EDvvTvd/Lq1xsFjKh6y/pG6NxABGdT6VoeWGVtQGqwpbOZGgq
xoSYkWm8MmAkkqYXZLraSEzyxxxu4cQzvzz3vrpN3AgAhObP2eUFU29EJAQpdKJW
fKAhohPVpd6+ETnzL53VLg1IJJdNGlpIziO9alNnYmDSnt2EwAELqTUl3jPiGYt5
cvSUBe3ER4/CkjvYXOVaO7ezHmCAkQpB2ILV8OwI74DQn7tNKf2gJnwzkYAF7yyf
XFGlJ8oaLpRV499mN7lNfo+ZV2HrR9xti+jUPFv+H+ROt4fMmAU5I95UksQFe/A9
YUdSBAEqKkW9zLDgpWS2oxJymGufBdhzxpw7uJlzrwsHIYIt7PSeJG4VO+xJqHvO
1qHXSukK648FlOImmVUM9csPOcvfOMZeAgh4i+HYQvFF/kGHp6ogevD4pVhztbzd
F9JhAbJSeOvZKZFPhzjgX+mCgvzVRniSdDg7wc3+YKNei2zQrmTsiiO6JyhQV2OI
tAqTk572zdZbrCtSgcthrN/uxbJSNnw4X9IZbWtFOUr3lr676II8Q112ttO3IVCe
fF/pZA==
=sPWf
-----END PGP MESSAGE-----
```

Figure 7.2

The purpose of encryption is to transform data in order to keep it secret from others, e.g. sending someone a secret letter that only they should be able to read, or securely sending a password over the Internet. Rather than focusing on usability, the goal is to ensure the data cannot be consumed by anyone other than the intended recipient(s).

Encryption transforms data into another format in such a way that only specific individual(s) can reverse the transformation. It uses a key, which is kept secret, in conjunction with the plaintext and the algorithm, in order to perform the encryption operation. As such, the cipher text, algorithm, and key are all required to return to the plaintext.

Hashing

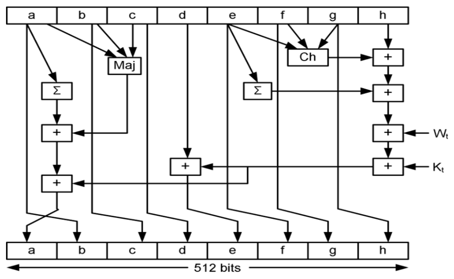


Figure 7.3

Hashing serves the purpose of ensuring integrity, i.e. making it so that if something is changed you can know that it’s changed. Technically, hashing takes arbitrary input and produces a fixed-length string that has the following attributes:

The same input will always produce the same output.

1. Multiple disparate inputs should not produce the same output.
2. It should not be possible to go from the output to the input.
3. Any modification of a given input should result in drastic change to the hash.

Hashing is used in conjunction with authentication to produce strong evidence that a given message has not been modified. This is accomplished by taking a given input, encrypting it with a given key, hashing it, and then encrypting the key with with the recipient’s public key and signing the hash with the sender’s private key.

When the recipient opens the message, they can then decrypt the key with their private key, which allows them to decrypt the message. They then hash the message themselves and compare it to the hash that was signed by the sender. If they match it is an unmodified message, sent by the correct person.

6.4 Summary

- **Encoding** is for maintaining data *usability* and can be reversed by employing the same algorithm that encoded the content, i.e. no key is used.
- **Encryption** is for maintaining data *confidentiality* and requires the use of a key (kept secret) in order to return to plaintext.
- **Hashing** is for validating the integrity of content by detecting all modification thereof via obvious changes to the hash output.

CHAPTER 7: MESSAGE DIGEST

7.1 Introduction

The Message Digest (MD) series of algorithms come under Hash functions used for data security and integrity purpose.

Message digest functions distill the information contained in a file, small or large, into a single large number, typically between 128 and 256 bits in length. The best message digest functions combine these mathematical properties:

- Every bit of the digest function's output is potentially influenced by every bit of the function's input
- If any given bit of the function's input is changed, every output bit has a 50 percent chance of changing.
- Given an input file and its corresponding message digest, it should be computationally infeasible to find another file with the same message digest value 0

Message digests are also called one-way hash functions because they produce values that are difficult to invert, resistant to attack, effectively unique, and widely distributed.

7.2 Types of Message Digests

Ronald Rivest developed 3 algorithms in the MD series.
The following are the algorithms:

- **MD2** –
    1. It was developed in 1989.
    2. MD2 is specified in RFC 1319. Although MD2 is no longer considered secure, even as of 2014, it remains in use in public key infrastructures as part of certificates generated with MD2 and RSA.
    3. The 128-bit (16-byte) MD2 hashes (also termed *message digests*) are typically represented as 32-digit hexadecimal numbers.

- **MD4** –

1. It was 1developed in 1990.
2. The message is padded to ensure that its length in bits plus 64 is divisible by 512. A 64-bit binary representation of the original length of the message is then concatenated to the message. The message is processed in 512-bit blocks and each block is processed in three distinct rounds.
3. The security of MD4 has been severely compromised. The first full collision attack against MD4 was published in 1995 and several newer attacks have been published since then. As of 2007, an attack can generate collisions in less than 2 MD4 hash operations.

- **MD5** –
    1. It was developed in 1991.
    2. It is basically MD4 with "safety-belts" and while it is slightly slower than MD4, it is more secure. The algorithm consists of four distinct rounds, which has a slightly different design from that of MD4. Message-digest size, as well as padding requirements, remain the same.
    3. In 1996 a flaw was found in the design of MD5. While it was not deemed a fatal weakness at the time, cryptographers began recommending the use of other algorithms, such as SHA-1

CHAPTER 8: MESSAGE DIGEST 5 (MD5) ALGORITHM

8.1 Introduction

MD5 is a hash algorithm that is used to verify data integrity through the creation of a 128-bit message digest from data input which may be a message of any length that is claimed to be as unique to that specific data as a fingerprint is to the specific individual.

MD5, which was developed by Professor Ronald L. Rivest of MIT, is intended for use with digital signature applications, which require that large files must be compressed by a secure method before being encrypted with a secret key, under a public key cryptosystem.

MD5 is currently a standard, Internet Engineering Task Force (IETF) Request for Comments (RFC) 1321. According to the standard, it is "computationally infeasible" that any two messages that have been input to the MD5 algorithm could have as the output the same message digest, or that a false message could be created through apprehension of the message digest.

MD5 is the third message digest algorithm created by Rivest. All three (the others are MD2 and MD4) have similar structures, but MD2 was optimized for 8-bit machines, in comparison with the two later formulas, which are optimized for 32-bit machines. The MD5 algorithm is an extension of MD4, which the critical review found to be fast, but possibly not absolutely secure. In comparison, MD5 is not quite as fast as the MD4 algorithm, but offers much more assurance of data security.

8.2 Steps involved in MD5 Algorithm

1. **Append Padding Bits**
    - The message is "padded" (extended) so that its length (in bits) is congruent to 448, modulo 512. That is, the message is extended so that it is just 64 bits shy of being a multiple of 512 bits long. Padding is always performed, even if the length of the message is already congruent to 448, modulo 512.
    - Padding is performed as follows: a single "1" bit is appended to the message, and then "0" bits are appended so that the length in bits of the padded message becomes congruent to 448, modulo 512. In all, at least one bit and at most 512 bits are appended.

2. **Append Length**
   - A 64-bit representation of b (the length of the message before the padding bits were added) is appended to the result of the previous step with low order byte first.
   - At this point the resulting message (after padding with bits and with b) has a length that is an exact multiple of 512 bits
   - The whole bit stream of 512 bits is divided into 16 blocks of 32. Let M[0 ... N-1] denote the words of the resulting blocks of message, where N is a multiple of 16. Each vector M is of 32 bits with low order byte first.

3. **Initialize MD Buffer**
   - A four-word buffer (A,B,C,D) is used to compute the message digest which are predefined.
   - Each of A, B, C, D is a 32-bit register. These registers are initialized to the following values in hexadecimal, low-order bytes first):

word A: 67 45 23 01

word B: ef cd ab 89

word C: 98 ba dc fe

word D: 10 32 54 76

4. **Process Message in 16-Word Blocks**
   - We first define four auxiliary functions that each take as input three 32-bit words and produce as output one 32-bit word.

F(X,Y,Z) = XY v not(X) Z

G(X,Y,Z) = XZ v Y not(Z)

H(X,Y,Z) = X xor Y xor Z

I(X,Y,Z) = Y xor (X v not(Z))

- In each bit position F acts as a conditional: if X then Y else Z. The function F could have been defined using + instead of v since XY and not(X)Z will never have 1's in the same bit position.) It is interesting to note that if the bits of X, Y, and Z are independent and unbiased, the each bit of F(X,Y,Z) will be independent and unbiased.

- The functions G, H, and I are similar to the function F, in that they act in "bitwise parallel" to produce their output from the bits of X, Y, and Z, in such a manner that if the corresponding bits of X, Y,and Z are independent and unbiased, then each bit of G(X,Y,Z), H(X,Y,Z), and I(X,Y,Z) will be independent and unbiased. Note that the function H is the bit-wise "xor" or "parity" function of its inputs.

- This step uses a 64-element table T[1 ... 64] constructed from the sine function. Let T[i] denote the i-th element of the table, which is equal to the integer part of 4294967296 times abs(sin(i)), where i is in radians. The elements of the table are given in the appendix.

5. Four Round Process

- **Four Round Loop**

```
/* Process each 16-word block. */
For i = 0 to N/16-1 do

/* Copy block i into X. */
For j = 0 to 15 do
Set X[j] to M[i*16+j].
end /* of loop on j */

/* Save A as AA, B as BB, C as CC, and D as DD. */
AA = A
BB = B
CC = C
DD = D
```

- **Round 1**

Let [abcd k s i] denote the operation

$a = b + ((a + F(b,c,d) + X[k] + T[i]) \lll s)$.

Do the following 16 operations.

[ABCD 0 7 1] [DABC 1 12 2] [CDAB 2 17 3] [BCDA 3 22 4]

[ABCD 4 7 5] [DABC 5 12 6] [CDAB 6 17 7] [BCDA 7 22 8]

[ABCD 8 7 9] [DABC 9 12 10] [CDAB 10 17 11] [BCDA 11 22 12]

[ABCD 12 7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]

- **Round 2**

Let [abcd k s i] denote the operation

$a = b + ((a + G(b,c,d) + X[k] + T[i]) \lll s)$.

Do the following 16 operations.

[ABCD 1 5 17] [DABC 6 9 18] [CDAB 11 14 19] [BCDA 0 20 20]

[ABCD 5 5 21] [DABC 10 9 22] [CDAB 15 14 23] [BCDA 4 20 24]

[ABCD 9 5 25] [DABC 14 9 26] [CDAB 3 14 27] [BCDA 8 20 28]

[ABCD 13 5 29] [DABC 2 9 30] [CDAB 7 14 31] [BCDA 12 20 32]

- **Round 3**

Let [abcd k s i] denote the operation

$a = b + ((a + H(b,c,d) + X[k] + T[i]) \lll s)$.

Do the following 16 operations.

[ABCD 5 4 33] [DABC 8 11 34] [CDAB 11 16 35] [BCDA 14 23 36]

[ABCD 1 4 37] [DABC 4 11 38] [CDAB 7 16 39] [BCDA 10 23 40]

[ABCD 13 4 41] [DABC 0 11 42] [CDAB 3 16 43] [BCDA 6 23 44]

[ABCD 9 4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA 2 23 48]

- **Round 4**

Let [abcd k s i] denote the operation

a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s).

Do the following 16 operations.

```
[ABCD  0  6 49]   [DABC  7 10 50]   [CDAB 14 15 51]  [BCDA  5 21 52]
[ABCD 12  6 53]   [DABC  3 10 54]   [CDAB 10 15 55]  [BCDA  1 21 56]
[ABCD  8  6 57]   [DABC 15 10 58]  [CDAB  6 15 59]  [BCDA 13 21 60]
[ABCD  4  6 61]   [DABC 11 10 62]  [CDAB  2 15 63]  [BCDA  9 21 64]
```

Then perform the following additions. (That is increment each of the four registers by the value it had before this block was started.)

A = A + AA

B = B + BB

C = C + CC

D = D + DD

end /* of loop on i */

6. Concatenate Low order byte first

AF<=A(7 DOWNTO 0)&A(15 DOWNTO 8)&A(23 DOWNTO 16)&A(31 DOWNTO 24);

BF<=B(7 DOWNTO 0)&B(15 DOWNTO 8)&B(23 DOWNTO 16)&B(31 DOWNTO 24);

CF<=C(7 DOWNTO 0)&C(15 DOWNTO 8)&C(23 DOWNTO 16)&C(31 DOWNTO 24);

DF<=D(7 DOWNTO 0)&D(15 DOWNTO 8)&D(23 DOWNTO 16)&D(31 DOWNTO 24);

7. Output

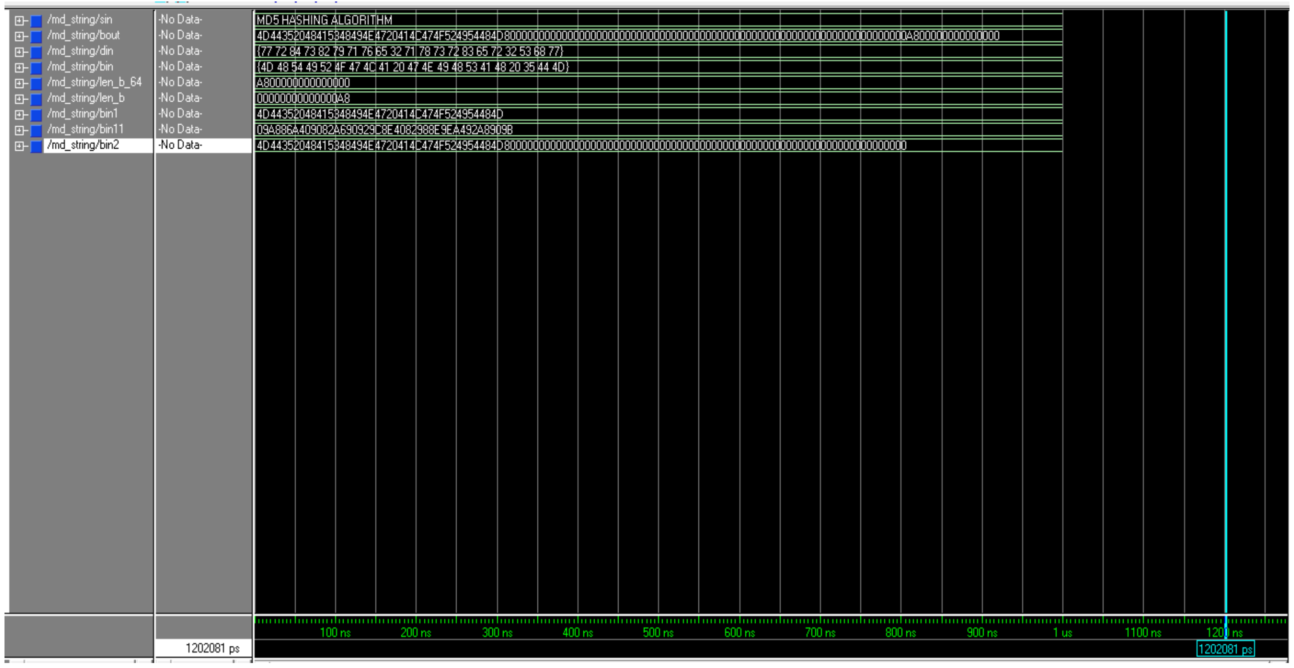
The message digest produced as output is A, B, C, D. That is,it begins with the low-order byte of A, and end with the low-order byte of D.

HASH= AF&BF&CF&DF

RESULTS AND OUTPUTS

Simulations Results

- Output of Module md_string

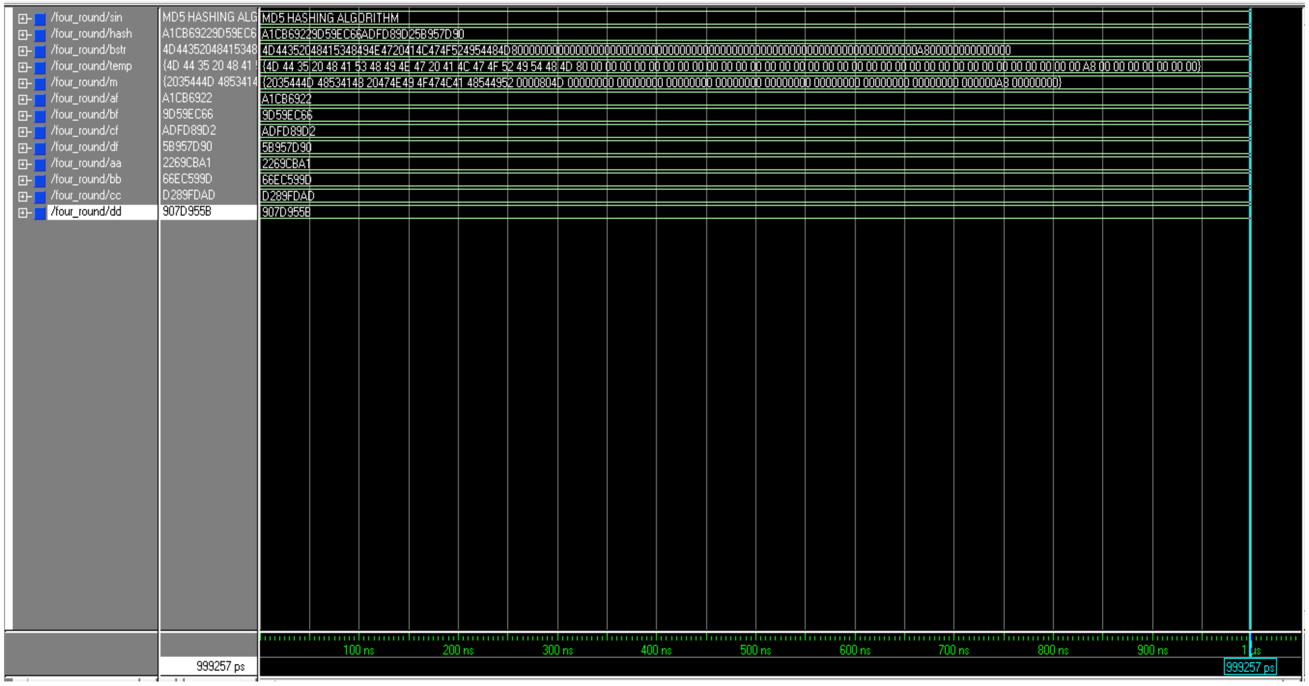


```
/md_string/sin       -No Data-  MD5 HASHING ALGORITHM
/md_string/bout      -No Data-  4D44352048415348494E4720414C474F524954484D800000000000000000000000000000000000000000000000000000000000000000000000A800000000000000
/md_string/din       -No Data-  {77 72 84 73 82 79 71 76 65 32 71 78 73 72 83 65 72 32 53 68 77}
/md_string/bin       -No Data-  {4D 48 54 49 52 4F 47 4C 41 20 47 4E 49 48 53 41 48 20 35 44 4D}
/md_string/len_b_64  -No Data-  A800000000000000
/md_string/len_b     -No Data-  00000000000000A8
/md_string/bin1      -No Data-  4D44352048415348494E4720414C474F524954484D
/md_string/bin11     -No Data-  09A886A409082A690929C8E4082988E9EA492A8909B
/md_string/bin2      -No Data-  4D44352048415348494E4720414C474F524954484D8000000000000000000000000000000000000000000000000000000000000000000000

                     1202081 ps   100 ns  200 ns  300 ns  400 ns  500 ns  600 ns  700 ns  800 ns  900 ns  1 us  1100 ns  1200 ns
                                                                                                                    1202081 ps
```

FIGURE R.1

- Output of Four Round Process



```
/four_round/sin   MD5 HASHING ALG   MD5 HASHING ALGORITHM
/four_round/hash  A1CB69229D59EC6   A1CB69229D59EC66ADFD89D25B957D90
/four_round/bstr  4D44352048415348  4D44352048415348494E4720414C474F524954484D800000000000000000000000000000000000000000000000000000000000000000000000A800000000000000
/four_round/temp  {4D 44 35 20 48 41 {4D 44 35 20 48 41 53 48 49 4E 47 20 41 4C 47 4F 52 49 54 48 4D 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 A8 00 00 00 00 00 00 00}
/four_round/m     {2035444D 4853414 {2035444D 48534148 20474E49 4F474C41 48544952 0000804D 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 000000A8 00000000}
/four_round/af    A1CB6922          A1CB6922
/four_round/bf    9D59EC66          9D59EC66
/four_round/cf    ADFD89D2          ADFD89D2
/four_round/df    5B957D90          5B957D90
/four_round/aa    2269CBA1          2269CBA1
/four_round/bb    66EC599D          66EC599D
/four_round/cc    D289FDAD          D289FDAD
/four_round/dd    907D955B          907D955B

                  999257 ps   100 ns  200 ns  300 ns  400 ns  500 ns  600 ns  700 ns  800 ns  900 ns  1 us
                                                                                                999257 ps
```

FIGURE R.2

RTL Schematic

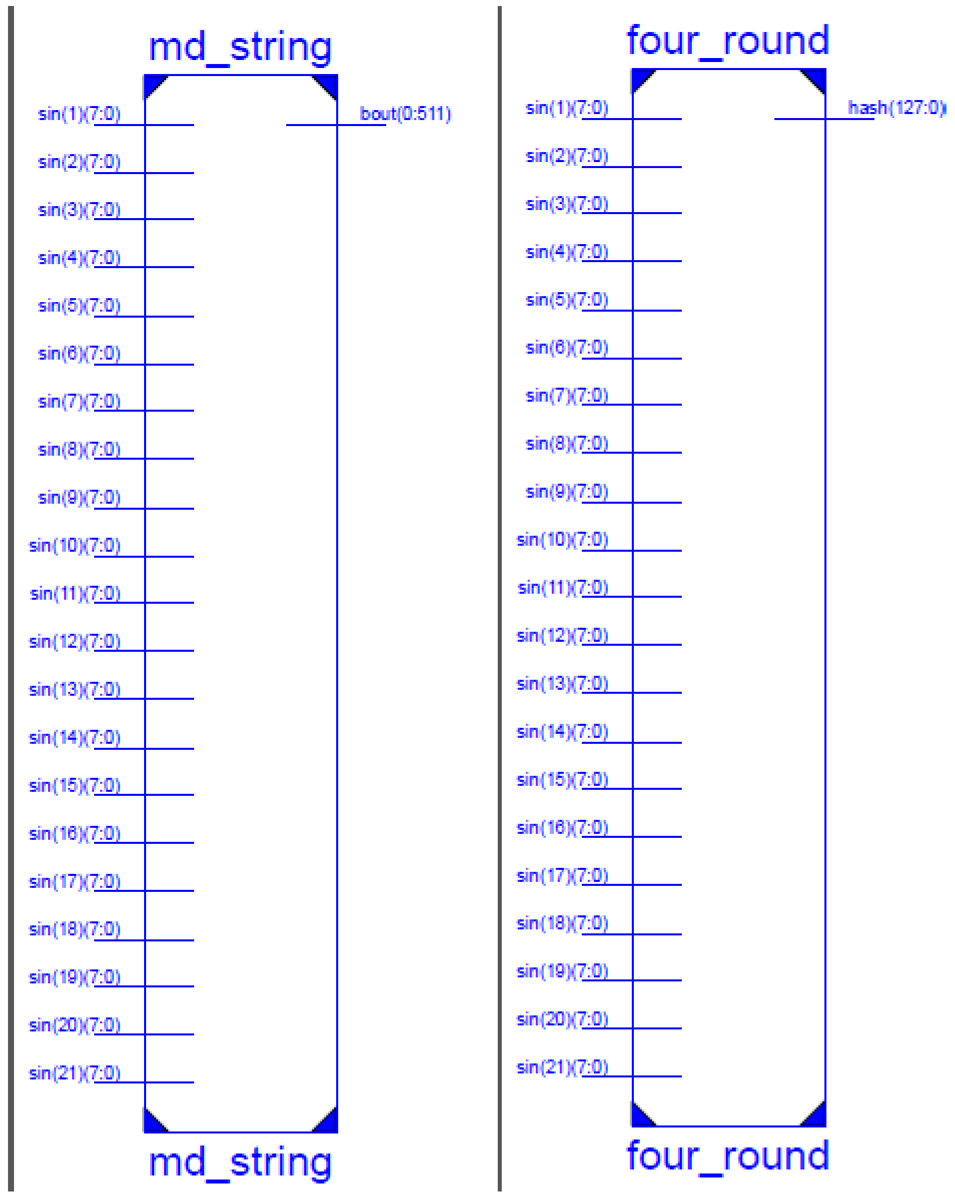


FIGURE R.3

FUTUTRE SCOPE

Introduction

The main computation of the Hash algorithm contains several rounds, which use the same equations for computation. In each round many internal loops are included. So, to improve the performance of the code two pipelining methods have been investigated.

- The coarse-grained pipelining method mounts several steps to one stage. The hardware size will increase significantly together with the number of stages.
- The fine-grained pipelining divides a single step into small stages, and helps increase the frequency, and consequently the throughput of the main module.

We apply this method in our MD5 implementations with two novelties of data forwarding, and two messages processing in an alternative manner.

The data forwarding technique is used to break the critical path in the iterations into 3 stages in the three-stage pipeline design. The data forwarding technique together with the two messages processing in the alternative manner are used to divide the critical path into 4 stages in a four-stage pipeline design. The latter guarantees shorter critical paths to one adder and some data movements at all stages.

Coarse Grained Architecture

The coarse grained pipeline architecture groups several steps together to form a pipeline stage. The common procedure is grouping by round, in which the same function, constant, and key’s location function are used.

Thus, the architectures duplicate the number of adders but keep the number of other function units as small as 1, 2, 4, 8, or 16. The whole algorithm computation for one message is finished after several stages (up to 64 when one repetition is mounted to one stage), and several messages (up to 64) fulfill all the stages to make the computation power. The hardware is duplicated to meet the demand. A high throughput of 5.8Gbps is recorded for a SIG-MD5 system but the hardware use is also over 10 times higher than others (11,498 hardware slices on Virtex-2). This architecture can be used in extremely high speed security systems, which require a high throughput without any restriction on the hardware size and the power consumption.

Data Dependency, Forwarding, and Dependency Removal

The main computation is represented by the equations

A = B + ((A + Func + X[k] + T[i]) <<s) (a)

A ← D; B ← A; C ← B; D ← C (b)

in which, <<represents a rotation shift left operation.

- **Data Dependency**

  Data dependency in equations can be easily seen if we rewrite it as follows:

  tempB = B (a)

  B = B + ((A + T + X + Func) << s) (b)

  A ← D; C ← tempB; D ← C (c)

  Equations show that the values of A, C, D rely on previous values of D, B, C, respectively. The new value of B, which is calculated by equation (b) relies on previous values of A, B on current, values of T, X, s and Func. X itself relies on its location denoted by k, which must be calculated from the step number i. Func depends on the previous values of B, C, D and the current step i. The new value of B (new B) completely depends on the step number and current values of A, B, C, and D. However, the internal values of T, k, X can be pre-computed because they rely on the step number i only. Therefore, we can make a pipeline by pre-computing k, and if we can pre-compute the value of A.

- **Data Forwarding and Dependency Removal**

  The locations of operands A, B, C, and D at each step are required for the forwarding operations. In order to compute the new value B (n3) using equation (4b), the current value A is required. Assume the values of operands A, B, C, and D at two steps before are a, b, c, and d, respectively. The values for A at the current and the preceding steps are transferred from operands D, and C respectively, which means the values of current A for the n3 computation are located in D, or C depending on the step number where we want to use it. In short, the values of A used in equation (b) can be taken as values of the operand D at the preceding step or C at the step before the preceding one. In this 3SMD5 and 4SMD5 implementations, we manage to implement a single step of MD5 into a pipeline based on the data dependency in equations (4).The computation of B requires a huge sequenced computation of k, X, Func and four 32-bit adders. This generates an enormous latency. However, if we pay attention to the trace of A, that latency can be divided into smaller stages. The address of the key of the current step can be pre computed several steps before, because it relies mainly on the step number i. The trace of A in Fig. 4 allows us to define the value of A at the current step as D at the preceding step or C at the step before the preceding one. All that makes it possible to pre-compute A+T+X up to 3 steps before. In other words, the data dependency of the computation of B in the operand A is removed. The new value of B now relies on the value of C at the step before the preceding one. The pre-computation of A+T+X in some pipeline stages by forwarding the value of C or D to A helps ease the delay in the critical path of (4b).

CONCLUSION

MD5 is still one of the widely used hashing technique, though MD5 does have a greater collision risk, still it is quicker to generate an MD5 digest. If we're forced to generate many digests, we'll prefer MD5 because it's much faster to compute the 128 bit digest for MD5 as compare to other hashing algorithms which are required to process about 160 or more bits. Since MD5 is good enough for non-cryptographic purposes, the speed advantage makes it a better choice in most cases.

Applications

MD5 is a cryptographic hash function which, as such, is expected to fulfill three characteristics:

- Resistance to preimages: given x, it is infeasible to find m such that MD5(m) = x.
- Resistance to second-preimages: given m, it is infeasible to find m' distinct from m and such thatMD5(m) = MD5(m').
- Resistance to collisions: it is infeasible to find m and m', distinct from each other, and such thatMD5(m) = MD5(m').

MD5 digests have been widely used in the software world to provide some assurance that a transferred file has arrived intact. For example, file servers often provide a pre-computed MD5 (known as Md5sum) checksum for the files, so that a user can compare the checksum of the downloaded file to it. Most unix-based operating systems include MD5 sum utilities in their distribution packages; Windows users may install a Microsoft utility, or use third-party applications. Android ROMs also utilize this type of checksum.

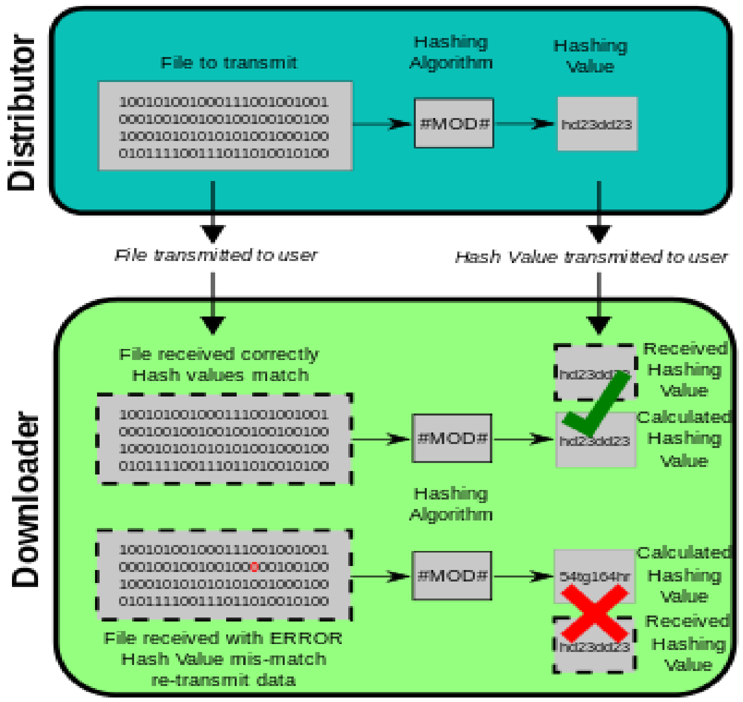


Figure A.1

MD5 can be used to store a one-way hash of a password, often with key stretching. Along with other hash functions, it is also used in the field of electronic discovery, in order to provide a unique identifier for each document that is exchanged during the legal discovery process.

APPENDIX

HDL Synthesis Report

Macro Statistics

# Adders/Subtractors	: 147
25-bit adder	: 21
26-bit adder	: 21
27-bit adder	: 21
28-bit adder	: 21
29-bit adder	: 21
30-bit adder	: 21
31-bit adder	: 21

Primitive and Black Box Usage:

# BELS	: 296
# GND	: 1
# LUT3	: 147
# LUT5	: 42
# LUT6	: 105
# VCC	: 1
# IO Buffers	: 680
#IBUF	: 168
#OBUF	: 512

Device utilization summary:

Selected Device : 6slx4tqg144-3 (Spartan 6)

Slice Logic Utilization:

Number of Slice LUTs: 294 out of 2400 12%

Number used as Logic: 294 out of 2400 12%

Slice Logic Distribution:

Number of LUT Flip Flop pairs used: 294

Number with an unused Flip Flop: 294 out of 294 100%

Number with an unused LUT: 0 out of 294 0%

Number of fully used LUT-FF pairs: 0 out of 294 0%

Number of unique control sets: 0

IO Utilization:

Number of IOs: 680

Number of bonded IOBs: 680 out of 102 = 666%

REFERENCES

[1] R. Rivest, “The MD5 Message-Digest Algorithm, RFC 1321”, MIT LCS & RSA Data Security, Inc., April 1992.

[2] Jayaram Bhasker, “A VHDL Primer”

[3] I.N. Tselepis, M.P. Bekakos, A.S. Nikitakis and E.A. Lipitakis, “MD5 Hash Algorithm Hardware Realization on a Reconfigurable FPGA Platform”

[4] Mohammed A. Noaman , “A VHDL Model for Implementation of MD5 Hash Algorithm”

[5] Janaka Deepakumara, Howard M. Heys and R. Venkatesan “FPGA IMPLEMENTATION OF MD5 HASH ALGORITHM”

[6] http://nsfsecurity.pr.erau.edu/crypto/md5.html

[7] http://tools.ietf.org/html/rfc1321