Microelectronics System Design Research Group
School of Electrical, Electronic & Computer Engineering

Hardware Implementation of SHA-1 and SHA-2 Hash Functions

James Docherty

Albert Koelmans

Technical Report Series

NCL-EECE-MSD-TR-2011-170

Contact: james.docherty@newcastle.ac.uk; albert.koelmans@ncl.ac.uk

NCL-EECE-MSD-TR-2011-170

Microelectronics System Design Research Group

School of Electrical, Electronic and Computer Engineering

Merz Court

Newcastle University

Newcastle-upon-Tyne, NE1 7RU, UK

http://async.org.uk

Abstract

In this thesis, an ASIC capable of performing the SHA-1 and 2 Hash Functions is presented. Following an overview of the importance of cryptography in the modern age and a brief history of computer-based cryptography, the significance and operation of hash functions is presented with focus on the SHA family of algorithms. After this is the design and testing section where the modular method of manufacture; components and tools used; methods of testing and results are shown. The ASIC is at first only capable of performing SHA-1, but modifications are demonstrated that give capability to perform SHA-2, with the possibility of SHA-3 or another hash function such as MD5 being suggested. Finally, conclusions regarding the throughput, modular method of design, comparisons to other published work and future developments are given, with reference to the rise in remote connections requiring dependable, high speed cryptographic functions.

List of Principal Symbols and Acronyms

Symbol	Definition
AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
ASM	Asynchronous State Machine
CLA	Carry Look Ahead Adder
CSA	Carry Save Adder
DES	Data Encryption Standard
FA	Full Adder
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GUI	Graphical User Interface
IV	Initialisation Vector
K	Round Constant
LSB	Least Significant Bit
MAC	Message Authentication Codes
MAR	Memory Address Register
MD	Message Digest
MDR	Memory Data Register
MSB	Most Significant Bit
NIST	National Institute of Standards and Technology
NSA	National Security Agency
PC	Program Counter
ROM	Read Only Memory
RTL	Register Transfer Layer
S/DRAM	Static/Dynamic Random Access Memory
SHA	Secure Hash Algorithm
SSL	Secure Sockets Layer
TCG	Trusted Computer Group
TLS	Transport Layer Security
TPM	Trusted Platform Module
VCS	Version Control System
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLIW	Very Long Instruction Word
Wt	Word Size for Hash Function

Table of Contents

Table of Figures/Tables/Equation

Introduction

Modern use of Cryptography

Over the last three decades, the use of information technology in our everyday lives has increased dramatically. Due to this, the growth rate for e-commerce has been double-digit over the last decade, with an estimated $301 billion expected online retail sales in 2012 [1]. This extreme increase in online trading has lead to a rise in online attacks to obtain money through deception or other illegal means. Due to this, companies and consumers using e-commerce have become more aware of security risks exchanging information over such an open medium. This increased knowledge has lead to several third parties setting up secure areas for credit card and bank account details to be shared with minimal risk of the numbers being obtained and used fraudulently. Major credit companies such as Visa and MasterCard have set up subsidiaries (e.g. Verified by Visa) to give consumers confidence that the sites they are buying from are safe. These “security seals” are becoming more common on commercial sites, as customers have been found to avoid purchasing from companies who do not use them [2].

When shopping on The Internet, a connection is set up between the computer being used and the company server. This is done using a “Challenge and Response” through the Transport Layer Security (TLS), or its predecessor Secure Sockets Layer (SSL) [3].

Figure 1: Challenge and Response

Challenge and Response uses a mixture of symmetric block ciphers and Message Authentication Codes (MAC). The MAC is constructed using a Hash Function such as the MD or SHA families. These Hash Functions output a fixed length digest of a message which may have arbitrary input length. The original message is converted to blocks of a fixed size, which are sequentially reduced. This is repeated for the entire file, giving a message digest.

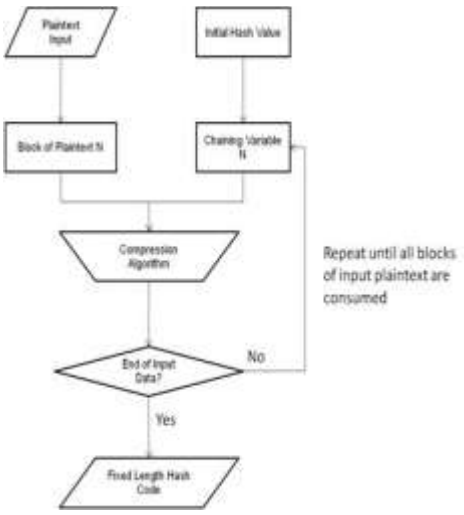

Figure 2: General Hash Function Operation [4]

Due to the increase in attacks, available security has been modified and updated to increase user security. The first governmental cipher approved was in 1976, when the National Security Agency (NSA) of the United States approved The Data Encryption Standard (DES), based on IBM's Lucifer Cipher with a 2^{56} key as the standard for Cryptography [5]. Due to increases in computing power, this is now breakable and Triple DES is used to increase encryption. To increase available security, the National Institute of Standards and Technology released the Advanced Encryption Standard (AES) based on the Rijndael Cipher in 2001 [6]. The majority of cryptographic systems are either block ciphers or hash functions. Block ciphers work by taking the entire message and map plaintext blocks of size *n* to ciphertext blocks of the same size before applying the cryptographic functions [7]. While for a short message this is acceptable; for a large document the time required could be unacceptable [8], especially in a time critical application such as commodity trading. In this case, it is more desirable to have the receiver know the message is unchanged from being sent to being received [8]. To ensure the message is identical, hash functions are used [6].[6]

Hash Functions: Operation and Weaknesses

Hash functions take a plain-text file as their input and convert it to fixed-size blocks, which are then compressed to a fixed size sequentially. These blocks are then used as the Chain Variable in each step until the entire file has been completed, producing a digest of the message [4]. This digest can be transmitted with the message and a second digest created from the received message at the recipient. If the digest received and the one computed match, the message can be thought of as authentic [9]. The compression algorithm varies between algorithms and can be made public, as a would-be attacker gains no advantage from knowing how the hash function performs its digest creation.

Hash functions are susceptible to the Birthday Paradox. This greatly reduces the number of attempts needed to guess the message from the received hash function. With a SHA-1 Hash

Function, the digest is 160-bit, meaning a brute force attack would require 1.4×10^{48} guesses, whereas a Birthday Attack reduces this to 1.4×10^{24} [4].While this is still an excessively large number of guesses to feasibly perform, it has halved the number needed and exposes a weakness in hash functions. Several of the more commonly used hash functions have been broken in recent years. MD-4, using a 128-bit Hash Function had collisions identified in 1996 in [10]. MD-5, developed to remove known weaknesses in MD4 [11] had collisions identified in 2004 in [12], which lead to further research in [13] on the SHA-1 Algorithm, which uses a 160-bit Hash Function. Due to these attacks, none of these hash functions are deemed suitable for use in high-security applications. SHA-2, using 256 and 512 bit hash functions is now recommended as the best hash function for normal use [14], but with the penalty of reduced processing speed. It is noted in [15] that performing a hash function with SHA-2 takes the same length of time as encryption with AES, so unless throughput can be increased, a standard message may as well be encrypted using AES rather than sent with a calculated signature. However AES is a symmetric cipher, so has issues regarding key distribution. When using AES, the same key is used to encrypt as decrypt. Therefore this cannot be sent on an unsecure channel. So while using AES between two users who regularly communicate and have developed their own key is feasible, for two strangers the distribution of the key remains an issue. A way round this would be to use an asymmetric cipher, such as RSA, but this is significantly slower than AES and therefore worse than using it or a hash function if speed is a priority. So for these cases, SHA-2 is the most logical solution to ensure a message is not tampered with in transit.

The three primary goals of a Hash Function are:

- Preimage Resistance: The inability to find a message from its hash function
- 2nd Preimage Resistance: No second input can be found that has the same hash function as the first (soft collision resistance)
- Collision Resistance: No two distinct inputs can have the same hash function as an output (hard collision resistance) [7]

Along with this, a small change in input should change a great deal of the output, this is known as the Avalanche Effect and was proved to take place up to the 73rd round of 80 in SHA-512 [16].

SHA-1 and 2 Operation

Secure Hash Algorithm (SHA) is the most widely used Hash Function in the world [17]. It was developed by the National Institute of Standards and Technology (NIST) in the United States and first published in 1993. This version (SHA-0) was found to have a serious security flaw, though NIST never published the details of this, and was replaced in 1995 with SHA-1 [9]. In recent years, SHA-1 has been found to have weaknesses, meaning a collision may be found. It was suggested in [18] that with a $1 million budget, a message could be broken in 25 days. Due to this, NIST developed SHA-2, which has a larger output (256 or 512-bit over the 160-bit in SHA-1) and differences within the message computation. SHA is ratified by NIST under ISO10118-3/FIPS180-1. Due to the Preimage resistance of SHA, the method of operation can be made public. Would-be attackers gain no benefit from knowledge of the SHA Algorithm, as no key is used to create the message digest.

SHA-1

SHA-1 is published in [19] and works by padding the message (M) of *l*-bits with a value *k* to give the smallest solution of $l + l + k \equiv 448 \bmod 512$, before appending a "1" at the message end due to the function being big-endian in operation. Following this, the 64-bit block equal to the binary representation of *l* is appended. This padding makes the message a multiple of 512-bits, allowing it to be split into N 512-bit blocks. Once the message is padded, 4 rounds of 20 steps are carried out (giving 80 steps in total) to create a digest. These blocks are split into 80 words of 32-bits (W_t), which are run through logical operations such as Exclusive OR (XOR) and Left Rotations (RotL) through 80 rounds of operation. In this, the hash value will change due to the previous values calculated in the round; as well as due to the initialisation vector (IV), which is a predefined constant. Once all this is complete, the result is a 160-bit hash value based on the inputted message. The left-shift operation on the expansion was added to SHA-1 as a countermeasure to the weaknesses identified in SHA-0 [20].

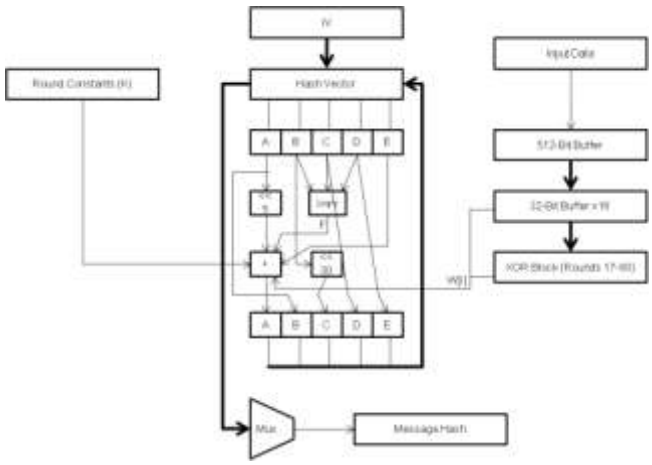

Figure 3: SHA-1 Round Calculation

SHA-2

SHA-2 operates in much the same way as SHA-1, but because values of 256, 384 and 512-bits are possible, the primary differences are more initial blocks of messages (8 instead of 5), fewer numbers of rounds (64 instead of 80), the use of right shifts as well as left shifts (SHA-1 only uses left shifts), constants for each round instead of blocks of rounds and the use of 64-bit inputs on the 512-bit function over the 32-bit for all others [7]. Messages are initially padded from 16 to 64 rounds using logical calculations based on the inputted message and each round is calculated using six variables based on the eight message blocks and 64 rounds. Unlike SHA-1, where there are four round calculations, SHA-2 only uses one. Its inherent strength comes from the use of 64 round constants over the four in SHA-1. This greatly reduces the risk of collisions and to date none have been found.

Although SHA-2 is increasing in popularity, SHA-1 is still significantly used, not least for its incorporation into the Trusted Platform Module (TPM) ASIC [4]. However, TPM are currently investigating the use of SHA-2 in later modules [21]. If this takes place, the ASIC will also have to contain backwards-compatibility to SHA-1 for communication with older systems. Therefore, it is important that an ASIC can manage creation of SHA-1 and SHA-2 message digests at a speed where delay would not be noticed on a high-speed internet connection ([22] suggests a speed of 40Gbit/sec for a Fibre-optic line).

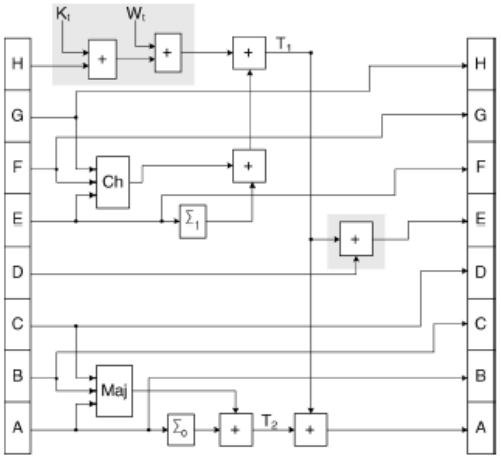

Figure 4: SHA-2 Round Calculation [23]

Comparison of SHA Hash Functions

While SHA-1's strength has been compromised in recent years, it is still deemed suitable for many applications. This combined with the use within the Trusted Platform Module means SHA-1 will be in use for the significant future, unless a major collision or other issue is found. If this occurs; deploying a replacement would be difficult [8]. The work by NIST and academics strongly shows that the importance of hash functions is clearly recognised and their strengths widely used. However, we should not be complacent and believe the currently used functions are suitable for the significant future. Since SHA-1 has already been shown to have possible collisions and with further cryptographic techniques under constant investigation, weaknesses in currently "safe" hash functions could quickly be identified and exploited. This, combined with the increase in computing power in line with Moore's Law means 512-bit keys may be insufficient in the near future. Some authors, such as Buchmann, believe that cryptographic functions are limited by governments as currently they are only breakable by an organisation with a great deal of computing power such as the FBI [6]. When DES was originally developed and released in the 1970's, only the US government had computers powerful enough to break its key in a feasible timeframe. As the number of transistors in home computers increased, DES became unusable and AES was introduced, with triple-DES as a temporary countermeasure. This principle will continue and algorithms must be successfully future-proofed to prevent their exploitation. Current recommendations by security experts such as Vacca suggest that SHA-2 be used until SHA-3 is available in late 2012 [24].

Literature Review

The Importance of Trusted Computing

Trusted Computing is gaining in importance due to the greater volumes of data stored electronically. Systems that are tamper evident, able to authenticate to a network, maintain integrity of the system and store data securely are needed to reduce the susceptibility to hackers. Due to this, the Trusted Computing Group (TCG) developed the Trusted Platform Module (TPM) [4]. TPM uses an ASIC that stores all cryptographic keys and hash values on the motherboard of a PC in a tamper evident and resistant way. Using an ASIC allows for all cryptographic resources, along with a true random number generator (unlike a pseudo-random generator used in software) and signatures to be performed in hardware; away from direct hacker attack. Since these signatures are stored in the ASIC, any hardware changes will be easily spotted [4]. TPM's are currently manufactured by several companies, including Atmel and Infineon using the SHA-1 Hash Algorithm [21]. As has been shown previously, SHA-1 is known to have weaknesses; therefore the TCG infrastructure group has been investigating the use of SHA-2 in future modules. [9, 24] suggest using SHA-2 until further notice and using SHA-3 as soon as it is ratified.

Hardware Implementation of SHA1 and 2

Due to the high use of SHA, a great deal of research has taken place into implementing it into ASICs or FPGAs. Most published work concentrates on either SHA-1 or SHA-2 with none found on a fully integrated ASIC. [23] proposes core layouts for both SHA-1 and 2, but does not integrate these into a common core.

The majority of papers published aim to improve throughput or power consumption through the use of Pipelining or Unrolling the function; or clock enabling respectively.

[25] identifies reducing the critical path for calculation through the substitution of Carry Save Adders (CSA) for slower Carry Look-Ahead Adders (CLA). CSAs are capable of performing addition of three numbers, rather than the two that CLA or Full Adders (FA) can perform [26]. [27] however, uses Full Adders, placing them into a Wallace Tree to reduce the critical path. Both methods show speed benefits, with [25] giving a maximum throughput of 2073 Mbit/sec for SHA-256 and 950Mbit/sec for SHA-1 in [27]. Throughput is calculated using:

$$Throughput = \frac{Block\ Size}{Clock\ Period * Latency}$$ **Equation 1**

The reduction in Critical Path for the longest calculation allows the remainder of the SHA operation to be unrolled and therefore completed in one, rather than multiple, clock cycles. It is found in [28] that unfolding the design for two operations (so performing two operations in one clock cycle) gives a factor-of-two speed improvement for the same increase in area. Unfolding

is also performed in [29], increasing throughput to 76Mbit/sec in conjunction with pipelining. Changing the architecture to a pipeline allows for simultaneous processing of blocks while not affecting the SHA operation. Using the pipeline also gives a level of delay balancing to the circuit, preventing incorrect signal propagation through a circuit and causing an incorrect message digest [30]. Use of these improvement techniques comes at a gate penalty and therefore would not be ideal for applications where power consumption is critical.

As both SHA-1 and 2 have aspects of their calculation that are repeated several times, the operation can be thought of as a Finite State Machine (FSM) with output feeding into input until all folding and reductions are complete [31]. By using an FSM layout, all control signals and padding can be simply added as extra states, allowing easy changes from SHA-1 to SHA-2 with the majority of states staying common.

Power Consumption is a key factor in all modern integrated circuit design. Due to the increase in wireless and mobile internet, systems are more commonly running from battery power rather than a mains supply. Therefore, reduction in both dynamic and static power consumption must be considered. The main consumer of dynamic power is the clock signal, so reducing either its speed or proliferation through the circuit would have a noticeable effect. Reduction of the clock speed is not recommended, as this will reduce the number of computations that can be performed per second and therefore the throughput of the circuit. Due to this, methods of reducing its presence in the circuit take priority. In [32], both Locally Explicit and Bus Specific Clock Enabling are suggested as methods to reduce power. These take the clock signal and only allow it to propagate into areas of the circuit when a signal is applied to the sub-sections input. By doing this, the number of transistors the clock is applied to is reduced and therefore the capacitance it must charge/discharge in a clock cycle, thus reducing the dynamic power consumed on each clock event. As:

$$Dynamic\ Power = Frequency * Capacitance * Voltage^2$$ **Equation 2 [33]**

This technique is found to save up to 65% of the dynamic power, without altering any gate propagation times or overall operational speed.

Static power reduction is more difficult to implement, but equally important as leakage power loss can account for up to 50% of power consumed by an ASIC [32]. Little work can be found on reducing this in cryptographic ASICs, but techniques such as Logic Gating could be used to deactivate areas when not in use. However this would mean any data in these areas has to be stored before deactivation, increasing the register count for the ASIC. Due to the time constraints of the project, reductions in static power will not be investigated further.

A final method for increasing throughput suggested in [34] and tested in [35] is the use of Very Long Instruction Word (VLIW) to increase parallelism. These papers suggest that having

multiple sets of instruction commands to set logic prior to data arriving would improve throughput. While qualitative improvements have been noted, this technique is still in its infancy and was briefly tested during this thesis. VLIW allows a level of parallelism to take place, by running two functional areas of an ASIC simultaneously, but can also be used to pass data on a bus without using a RAM or cache to store the data prior as an intermediate device. While this can increase the instruction set and therefore technically mean the ASIC is no-longer using a RISC setup, most commercially-available chips such as ARM or MIPS have a large number of available bits to create and operation code and can therefore operate with 16-bit or greater instructions. For example, The MIPS M14K Chip allows the use of User-defined instructions to reduce processor loading and increase throughput [36, 37]. ARM also have this facility available through their THUMB code-set.

Design

Due to the time constraints involved in an MSc project, the SHA ASIC was not physically manufactured. Instead it was simulated using VHISC Hardware Description Language (VHDL). Many hardware description languages (HDLs) exist, with Verilog as the most well-known alternative to VHDL. The decision to simulate the project in VHDL was primarily due to its standardisation by the IEEE as standard 1076 [38]. Since VHDL was developed from a US Government request, it is publically available and can be used on any ASIC or other programmable logic device (PLD) [39]. Also, unlike some other HDLs, VHDL can be used to create a test bench in conjunction with the device [40]. This test bench can simulate the clock signal, along with input parameters; essentially generating the “whole world” outside the ASIC [41]. This means that the ASIC can initially be synthesised in VHDL and simulated to confirm correct operation. If this design was found to be suitable, and a desire to manufacture was expressed, the VHDL would be easily shared and recognised by any design tool due to its standardisation. This code could also be synthesised into either an ASIC or a programmable chip such as a Field Programmable Gate Array (FPGA) with minimal modifications.

To synthesise and simulate logic on the department computers, Xilinx was chosen as an acceptable tool. This allows presentation of data in nested VHDL files and simulation of a test bench in a graphical user interface (GUI). Figure 5 below shows the GUI used by Xilinx to simulate a circuit and show all inputs and outputs. Bussed inputs or outputs can be seen either as a single entity in hexadecimal, binary or as its separate data pins, which is very useful for debugging and investigation. As can be seen, the output is very clear and therefore more than acceptable to show a hexadecimal message outputted following calculation of the hash function.

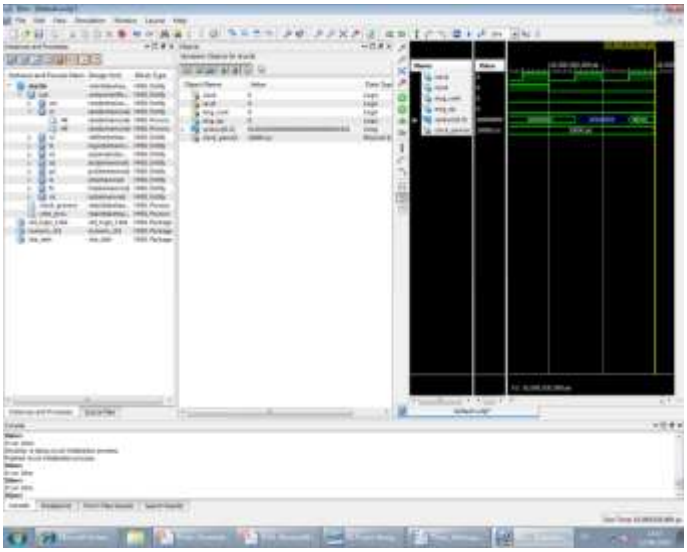

Figure 5: Xilinx Test bench Interface

Using a test bench also allows messages to be included in the VHDL program. Since the ASIC must successfully create the SHA-1 and 2 Hash Functions for a message, these can be calculated using commercially available software and placed in the test bench program as a comparison for the ASIC output. If the two do not match, the simulator can automatically give a failure message through use of the *assert* statement and halt if necessary. This saves time during debugging as messages are checked without user input.

Code Control and Project Management

As this project extended to several thousand lines of code, with many revisions throughout the practical phase, code management was identified early-on as a priority. To ensure accuracy at all times, A Version Control System (VCS) was adopted at inception. VCS allows tracking of a project throughout its phase through the use of revision and phase numbers. Therefore, all code in the project has the nomenclature *"section_phase_revision"* where phase and revision were incrementing numbers as changes were made. Section signifies whether the code was written during the SHA-1, SHA-2, Integration or test bench phase and gave an easy to remember method of code management, along with the ability to roll-back if a bug or issue was found.

The original project plan can be seen in Figure 6, this gave a target code freeze data of 19th July 2010. It was aimed to have no further code changes after this point, allowing time for test bench construction and debugging before a practical demonstration in August. The project was split into two distinct halves, with the practical work being the sole reserve of the project for the first five weeks and in parallel with the write up until the beginning of August. Following this, four weeks were reserved for the task of writing up. This was based on a standard 40 hour week, with five days contingency (three practical, two write up) to allow for unexpected issues.

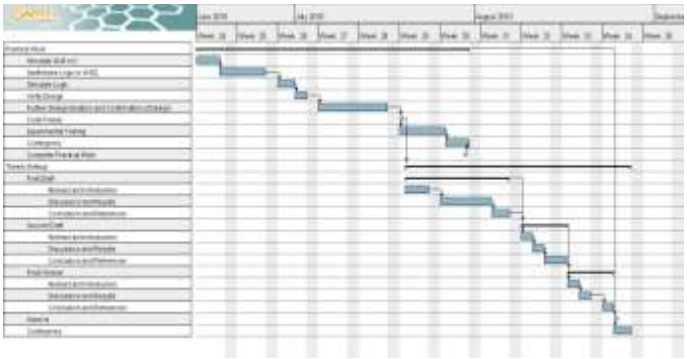

Figure 6: Initial Project Plan

The actual project plan can be seen in Figure 7. This shows the practical work took longer than expected due to the time for individual component testing and designing blocks. However, as the thesis write-up was begun at a much earlier date, the workload for this in August was greatly reduced. This allowed extra time to be spent on the practical work and adequate preparation for the presentation at the end of August. This time was logged and shared with the MSc

supervisor through weekly update sheets. These summarised the accomplishments, issues and goals for the week, along with how the project was progressing to plan and goals for the following week. These were found to aid identification of possible issues and prevent bottlenecks occurring in the project.

Figure 7: Actual Project Plan

Development of SHA-1

Overview

Figure 3 shows the operation of SHA-1 over one round. The operation can be thought of as a Finite State Machine (FSM) with nine states. A version of this can be seen in Figure 10. For this state machine, each sub-section of the hash function has a specific state with exit criteria and actions on entry at the rising edge of the clock pulse. Therefore, the design used can be thought of as a Mealy Machine [42].

As the goal for this project is to create an ASIC capable of performing both SHA-1 and 2, a modular approach to the design was taken. This can be seen in Figure 9 and is based on a microprocessor design in [41]. The ASIC will have a common 32-bit bus, allowing movement of data between sections of the architecture based on the current state of the FSM. All blocks are common to both SHA-1 and SHA-2 (256 bit) operation, with the only differences being the values of Initialisation Vector (IV), round constants (K), and number of rounds (80 for SHA-1, 60 for SHA-2). Therefore, by using a modular design, these values can be stored in sections of memory and extracted using selector pins, leading to an efficient use of gates and area [43]. The ASIC uses tri-state buffers to store data, reducing read/write operations to RAM and eliminating the need for a cache, as well as ensuring the bus can be shared by all components. Tri-state buffers allow a logic 1, logic 0 and high impedance state to be adopted by the output. Therefore, using the high-impedance state when not reading or writing to the bus prevents any corruption of the data taking place. Using direct access registers like these is a well known method of overhead reduction [44] and since 74% of most microprocessor execution is involved with either data transfer (45%) or Program Control (29%) [45], reducing the time for these to an almost atomic state is critical in increasing speed. Figure 8 shows the high-level flow chart to calculate a hash value for an arbitrary length message using this ASIC design.

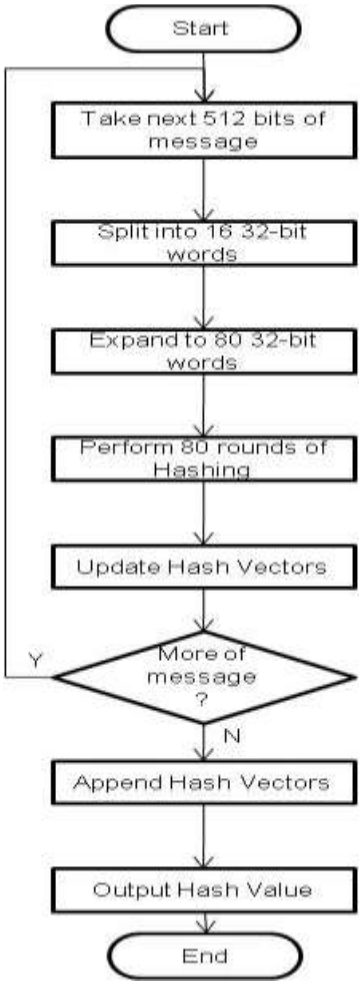

Figure 8: Flowchart of SHA-1 Operation

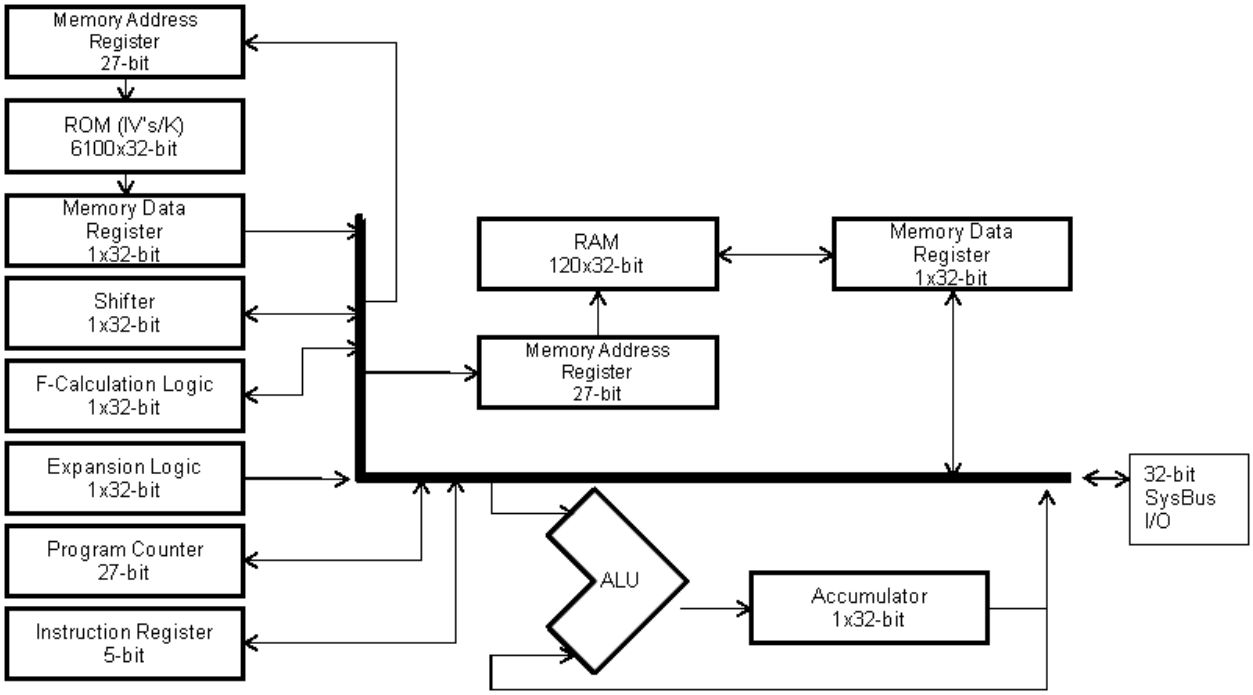

Figure 9: Architecture of SHA-1 ASIC

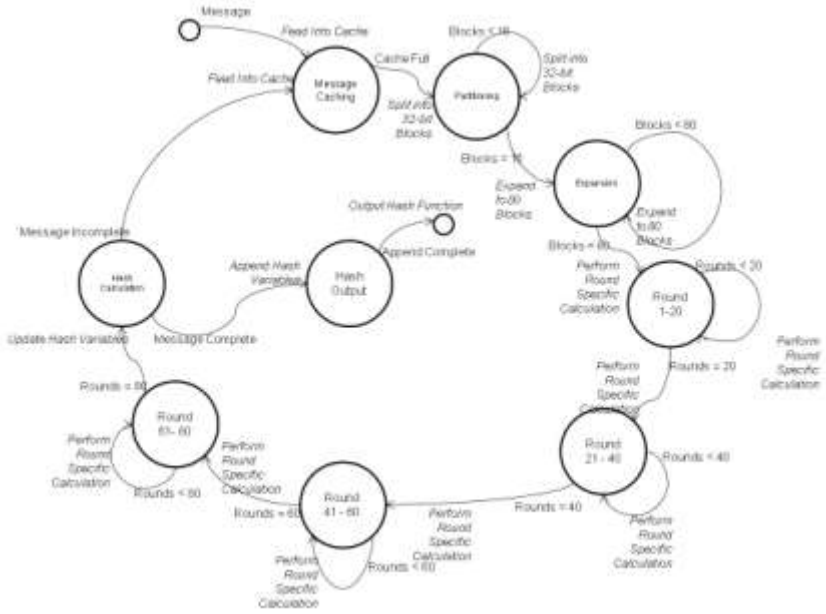

Figure 10: FSM of SHA-1 Operation

The ASIC will have 36 data pins; with 32 connected to the data bus, plus a clock pin, a “more message” pin, a selector for SHA-1 or SHA-2 and an asynchronous reset. The reset will place the FSM back into its initial state whenever it is set, as well as clear all register and RAM content. IV’s and K values will not be affected, as these are stored in ROM and only transferred to RAM once the ASIC has begun operation. Since these pins are common across all components of the ASIC, they are declared at a component level with all functional blocks at the next level of design connected to them an all internal signals using the *port map* command [38].

To control the ASIC operation, an operations code (opcode) will be used. This will decide which action the system is to take (store, load or execute) and therefore which state in the FSM is next to be executed. Coding for this opcode was one-hot encoding in the original design to allow easy distinction and expansion if necessary. Although one-hot encoding is not the most efficient in the number of flip-flops used, it uses less resources than other coding techniques for large number of states and will give easy debugging during integration [41, 46]. Owing to a shortage of available bits for accessing memory addresses, the decision was made to switch to standard encoding. As the program in ROM totalled approximately 21000 lines, 12-bits were needed to access these within the ROM MAR. With one-hot encoding, only 11 were available and the system would enter an infinite loop. Other coding techniques such as grey codes were investigated as these are known to be more power saving due to minimal bit changing [33], but were not implemented in the initial design as it was felt standard encoding would be sufficient. This removed the advantages one-hot encoding could give, such as improved debugging and the exploiting of parallelism, but this was deemed a minor issue over having a fully functional ASIC.

While Figure 10 gives a high level flowchart of SHA-1 operation, in actuality the program is more complex. While constructing the circuit, 20 operational codes were found to be needed. These are used to store information in architectural blocks to RAM, place data from RAM in the blocks, execute routines in logic and output the message. A full list of opcodes can be seen in Table 1.

Opcode	Use	Opcode	Use
Store ACC	Stores Value In Accumulator to MDR	*Load ACC*	Loads Accumulator with value from MDR
Store Expansion	Stores Value In Expansion Logic to MDR	*Load Expansion*	Loads Expansion Logic with value from MDR
Store Logic	Stores Value In Logic to MDR	*Load Logic*	Loads Logic with value from MDR
Store Shifter	Stores Value In Shifter to MDR	*Load Shifter*	Loads Shifter with value from MDR
Store ROM	Stores Value In ROM (selected by ROM MAR) to MDR	*Left Shift*	Runs Left Shift Operation
Logic Calculation1	Runs AND Logic	*Logic Calculation2*	Runs OR Logic
Logic Calculation3	Runs NOT Logic	*Logic Calculation4*	Runs XOR Logic
ALU Add	Performs Modular Addition	*More Message*	Denotes if message on input to Bus is complete
Message Out	Outputs Message to IO	*Expansion Calculation*	Runs Block Expansion
Store IO	Stores Value in IO to MDR	*Load IO*	Loads IO with value from MDR

Table 1: Opcodes for SHA-1 ASIC

In conjunction with these opcodes, there are flags internal to the ASIC to allow progression through the FSM when the PC is incremented. This is a standard method of microprocessor design, which has been widely used previously, meaning that the system has a set of instructions specific to its architecture [47]. In total for this ASIC, there are 33 flags, which are detailed in Table 2.

Flag	Use	Flag	Use
ACC_Bus	Load Accumulator onto Bus	*Load_ACC*	Load Accumulator with Bus Contents
PC_Bus	Load PC onto Bus	*Load_PC*	Loads PC with Bus Contents
Load_MDR	Load MDR with Bus Contents	*MDR_Bus*	Load MDR onto Bus
Addr_Bus	Load IR onto Bus	*Load_IR*	Load IR with Bus Contents
Load_MAR	Load RAM MAR with Bus Contents	*Load_MAR_ROM*	Load ROM MAR with Bus Contents
Shift_Bus	Load Shifter Contents to Bus	*Load_Shift*	Load Shifter with Bus Contents
Logic_Bus	Load Logic Contents to Bus	*Load_Logic*	Load Logic with Bus Contents
Expand_Bus	Load Expander Logic Contents to Bus	*Load_Expand*	Load Expander Logic with Bus Contents
Logic_Ex1	Execute AND Logic	*Logic_Ex2*	Execute OR Logic
Logic_Ex3	Execute NOT Logic	*Logic_Ex4*	Execute XOR Logic
Shift_Ex	Execute Shifter	*Expand_Ex*	Execute Expansion
Inc_PC	Increment PC	*ALU_ACC*	Load Accumulator with ALU Contents
CS	Chip Select. Uses MAR to set up Memory Addresses	*R_NW*	Read, Not Write: Stores MDR to memory when 0 and memory to MDR when 1
ROM_CS	ROM Chip Select. Uses ROM MAR to set up memory addresses	*Read_ROM*	Outputs contents of ROM Memory to MDR when 1
MDR_ROM_Bus	Load ROM MDR onto Bus	*ALU_Ex*	Execute ALU Function (addition)
PC_Reset	Resets Program Counter to allow loop to run again	*IO_Bus*	Load IO onto Bus
Load_IO	Load IO with bus contents		

Table 2: Flags for ASIC

To reduce the design down to the Register Transfer Layer (RTL) model, the FSM model in Figure 10 was refined to an Algorithmic State Machine (ASM) Chart. This is a widely used tool for developing logic for a complex circuit [41] allowing the high-level design of FSM to be reduced to possible Boolean expressions. This ASM for this circuit can be broadly split into two sections; the first is the instruction fetch, where the PC is incremented and all items in the memory set up accordingly; the second section is the execution area, where the instruction to be executed takes place based on the opcode.

The ASM for the ASIC can be seen in Figure 11. Each box signifies a stage in the ASM, meaning there are 33 stages, compared to the 9 in the original FSM. Stages 0 – 3 are concerned with instruction fetching, where the PC is outputted, incremented and used to pull the next instruction from the ROM into the instruction register. These instructions can be split broadly into three sections: Load, Store or Calculate. The load section pushes data from the RAM into the MDR and onto the Bus, where the block requiring the data will store the data in its register. The store section allows blocks to push data onto the bus, where the MDR will collect and store it in RAM. The execute section allows a logic block to execute its specific action. While this is not the most efficient architecture, meaning the system only executes one block per line of the PC; it gives a layout that is easy to debug and allows parallelism to be brought in at a later date though the use of VLIW encoding, or possibly one-hot encoding if the number of operational codes or lines of firmware code could be reduced to a point where all memory addresses could still be accessed. Parallelism will not affect bus-loading, as the load and store sections of the program can still be executed sequentially, preventing two sections attempting to use the bus at the same time. A more advanced design may use a mutual exclusion system to block bus access when a system has requested it. This is similar to work in [48] and could lead to an asynchronous system with globally asynchronous local clock (GALS) for the logic blocks. However, such a concept is beyond the scope of this project, where the system will remain fully synchronous and running from a single clock.

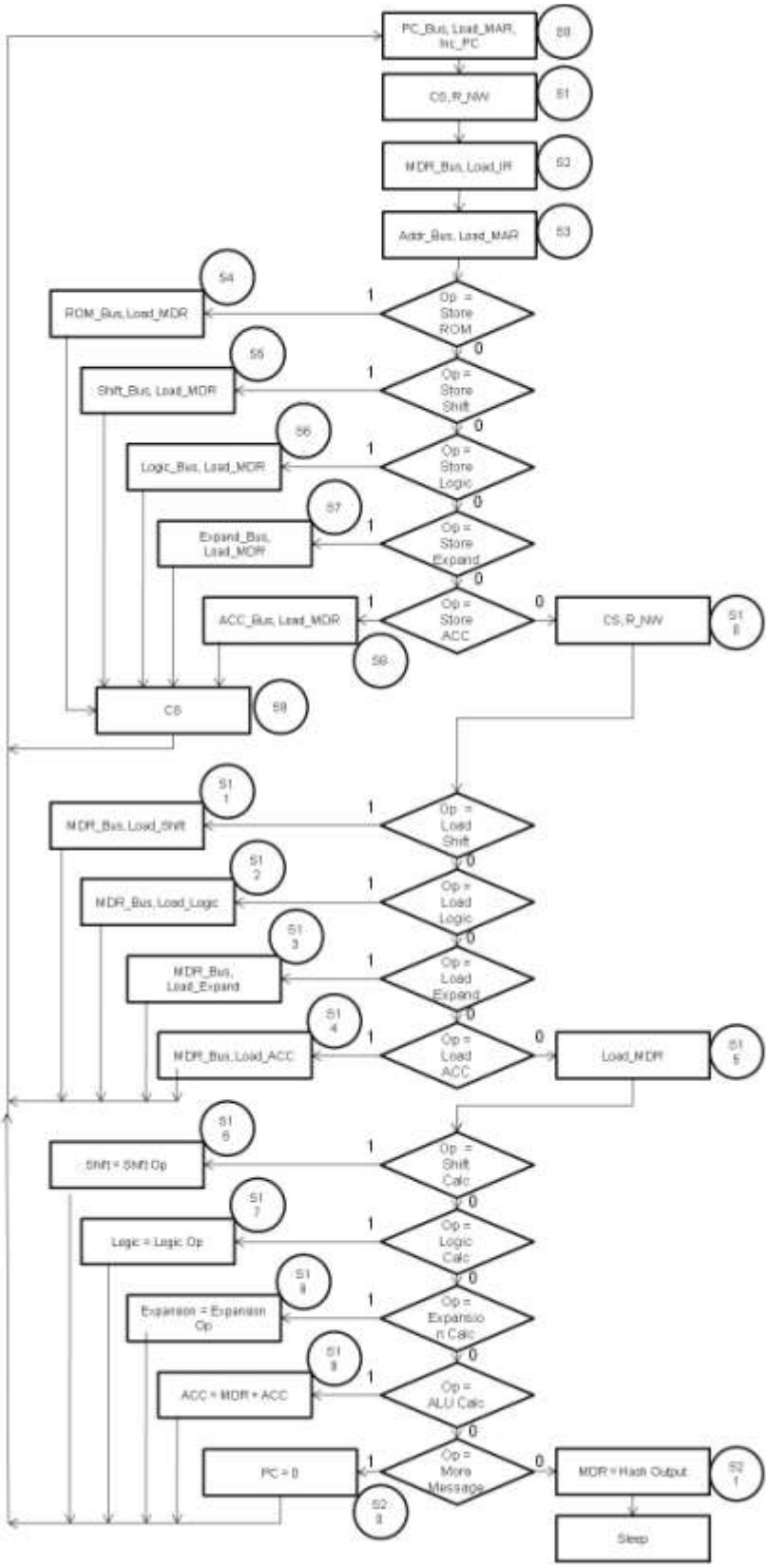

Figure 11: ASM Chart for ASIC

Logic F_t

A key part of the SHA-1 implementation is the logic functions applied across each round. These vary over the 80 rounds and are used to update the "A" value within the round calculation. The three Boolean equations used are:

- (B.C)+($\overline{B}$.D) for Rounds 1-20
- B⊕C⊕D for Rounds 21-40 and 61-80
- (B.C)+(B.D)+(C.D) for Rounds 41-60

The gate implementations for these can be seen in Figure 12 to Figure 14. These diagrams include the reduction of these gates to NAND gates only. Using this technique would improve gate balancing, as all gates would have the same delay time; as well as reduce complexity of a part library. While the VHDL for this was constructed, it was deemed overly complicated to implement in the time available and not used. Instead, four opcodes were created; logic_ex1 – logic_ex4 that run the four logic functions used in the calculations (AND, OR, NOR and XOR). Therefore, the logic calculations will be split into their separate components and run one gate at a time to progress through the logic functions. Although this technique is slower than other logic implementations, this implementation allows any logic function to be implemented without updating the chip architecture. To change the functions, only the program stored in ROM would require updating. This give the ASIC the flexibility to run different hash functions if needed and possibly run SHA-3 once its architecture is announced by NIST in Quarter 4 of 2010 [49].

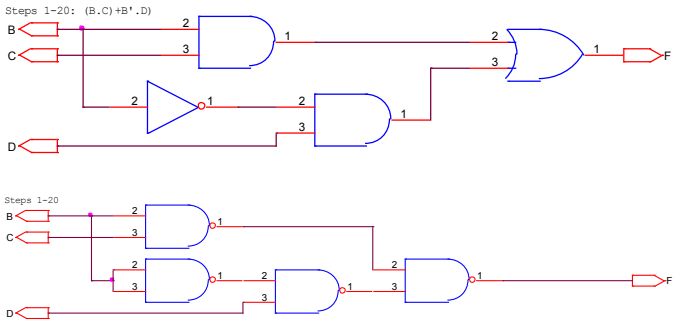

Figure 12: Round 1-20 Logic and NAND Gate Reduction

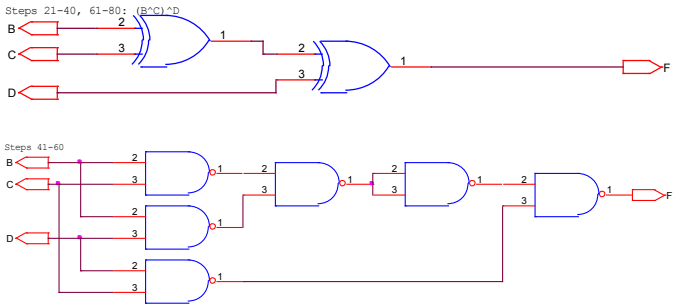

Figure 13: Round 21-40 and 61-80 Logic and NAND Gate Reduction

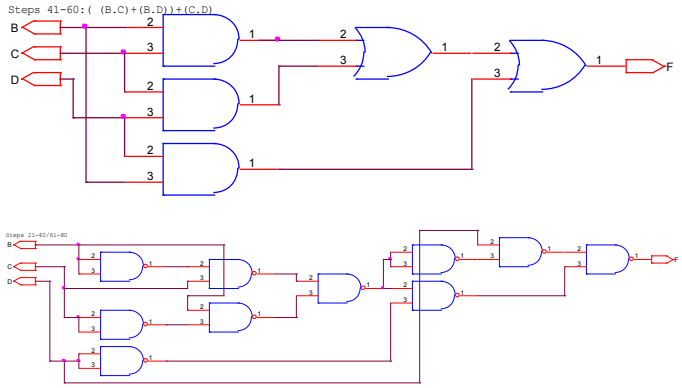

Figure 14: Round 41-60 Logic and NAND Gate Reduction

To test the operation of the logic gate section, each gate was tested individually and in combinational functions. The *sysbus* was asserted with a 32-bit value, which was fed into the logic and outputted back to the *sysbus*. Since the logic output can be pre-defined, the test bench was created with an *assert* statement to allow easy comparison of the *sysbus* output to the expected value. Any errors would stop the test bench and flag an error. The test bench proceeded correctly and the logic section of the ASIC was deemed to be operating successfully.

Circular Shift Register

A key item used in SHA-1 is the circular left shifter. This allows movement of data in a left shift, with the Most Significant Bit (MSB) becoming the Least Significant Bit (LSB) on each shift. This is different to a normal shift register, where data would simply be lost [50]. For this ASIC, the shift register is parallel loaded with data from the system bus, then will perform a shift each clock cycle that *shift_ex* is asserted. After the shift cycle is complete, data will be outputted onto the system bus in parallel for use in round calculations.

To test the shift register, a test bench was constructed that loaded a binary sequence into the components. The shift operation was active for a set number of clock cycles and the value in the registers outputted. This was compared to the expected value and would bring up an error message if a problem was found. No errors were found and the Circular Shift section of the ASIC was deemed to be operating correctly.

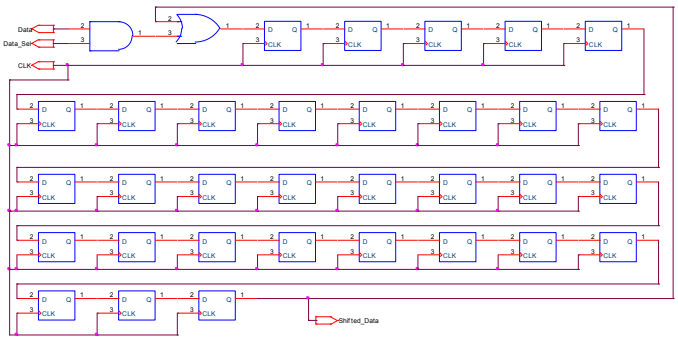

Figure 15: 32-bit Circular Shift Register [51]

Modulo-32 Adder

For SHA-1, a full Arithmetic Logic Unit is not required, as a Carry out of the final block is unused. Essentially, at a gate level, the adder can be thought of as a bitwise exclusive OR (XOR) within the word [38], with carry followed through but not outputted after summation is complete. However, for the VHDL implementation, the approach taken was to create a full ALU within the architecture and not output the final carry. This decision was taken to allow the ASIC to remain as multi-purpose as possible and allow minimal upgrading of components if SHA-3 requires full addition.

To test the modular adder, sequences of 32 bit numbers known to cause and not cause wrap-around of the modular arithmetic were fed into the ALU and accumulator using a test bench program. The results were outputted and compared automatically to the known correct results (e.g. 0xFFFFFFFF + 0x00000001 = 0x00000000 as the 32-bit number wraps around and does not propagate a carry). Any failures occurring would bring up an error message during the test bench process. No errors were found and the ALU section of the ASIC was deemed to be operating correctly.

Storage of Constants (ROM)

SHA-1 uses five initial values, plus four round-dependant constants. To ensure these are stored safely and cannot be corrupted, Read Only Memory (ROM) was used. This ROM also contained the program used to execute the Hash Function, which is accessed by the PC. The VHDL implementation of a ROM chip simulates a mask ROM. In these, the values are set permanently and cannot be changed. This is useful for high-volume manufacture, but means data cannot be changed once it is written. If the chip were to be actually manufactured commercially, EEPROM would be most likely used due to its ability to be re-written if necessary and update either constants or the program, giving flexibility to the ASIC if bugs were found after launch or possibly allowing SHA-3 to be installed if this uses the same components as SHA-1 or 2.

In VHDL, ROM is constructed as an array of constants [41]. The implementation is relatively simple, with an address requested at the input of the entity and the corresponding line of the array given as an output.

To test the ROM designed for this ASIC, a test bench was constructed that would request all lines of data in a non-sequential order. These memory locations were then outputted on the 32-bit system bus and compared to the known values of the memory addresses requested. Any failures occurring would bring up an error message during the test bench process. No errors were found and the ROM section of the ASIC was deemed to be operating correctly.

Storage of Values during Round Calculation (RAM)

The Initialisation Vectors used in SHA-1 are updated at the end of each round calculation. Therefore, these cannot be simply stored in RAM and updated once all logical operations have been performed. Likewise, 512 bits of data must be extracted from the initial message, stored, expanded and used for the round calculations. For this to take place, the ASIC requires a form of writable memory in conjunction with the ROM. The two main types of memory used are Static Random Access Memory (SRAM) and Dynamic RAM (DRAM). While DRAM is more space efficient that SRAM [52], it is slower due to its need to be refreshed periodically. The reason for this is DRAM uses fewer transistors (typically 1 compared to 6 on SRAM) and holds the value of these using capacitors, which require recharging. Since the memory used on this ASIC is small (3840 bits), along with the goal of high speed with low power consumption, SRAM will be used over DRAM. The layout of an SRAM cell can be seen in Figure 16, showing how it remains stable so long as power is applied. M1 – M4 are a cross-coupled inverter, allowing either a logic 0 or logic 1 to be represented by the cell. M5 and M6 are access transistors, used to allow the value stored in the SRAM to be changed [53].

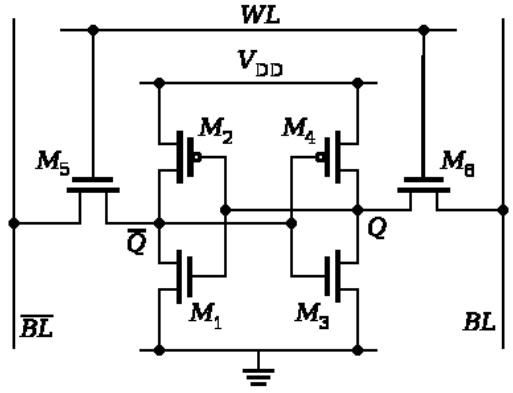

Figure 16: 6-Transistor SRAM Cell [54]

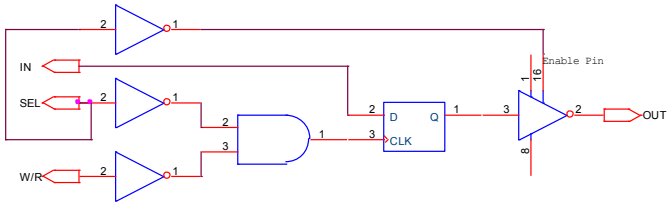

Figure 17: Static RAM Cell [53]

For the VHDL implementation of RAM, registers are added external to the RAM for storage of data and addresses for access. These two register sets are known as the Memory Address Register (MAR) and Memory Data Register (MDR). The MAR allows the RAM to know which address data on the MDR is to be written to or from. RAM input and output are on shared pins, accessing the *sysbus*, which is declared as an *inout* data set in the entity. The MDR and MAR are controlled by Boolean flags, which inform the RAM whether the MAR is to be loaded with the address on the system bus, the MDR is to be loaded with data from the RAM address

specified by the MAR, the data on the MDR is to be loaded to into the RAM address specified in the MAR or the data in the MDR is to be outputted onto the system bus.

To test this section of the system, a test bench was created that loaded each section of the RAM with a number corresponding to its memory address. These were then extracted in a random order, outputted on the 32-bit system bus and compared to the known values of the memory addresses requested. Any failures occurring would bring up an error message during the test bench process. No errors were found and the RAM section of the ASIC was deemed to be operating correctly.

Program Counter (PC)

To ensure progression through the program correctly, a Program Counter is used. This is essentially a simple counter to progress though the addresses in memory where instructions are stored; extract these instructions onto the bus and allow execution. If a section of operation requires a jump within the PC, the current value will be stored to RAM and extracted once the branched instruction is complete – therefore allowing the program to continue in a linear fashion until completion. In the event of the message requiring hashing being greater than 512-bits in size, the “more message” flag will be set and cause a hard reset of the program counter to a predefined value. This will allow repetition of the program, without overwriting the current hash value with the IV values stored in ROM.

To test the program counter, a test bench was created that stimulated the flags in a specified order to ensure the PC would increment, output to the system bus and reset to a pre-defined value when selected. Any failures occurring would bring up an error message during the test bench process. No errors were found and the PC section of the ASIC was deemed to be operating correctly.

Instruction Register and Sequencer

Due to the length of the opcode being smaller than the size of the sysbus, the code is padded prior to sending. Therefore the instruction register removes this padding prior to sending the code to the sequencer. Constant *rfill* is defined in the ASIC package and concatenated to the opcode when the instruction register is sent to the sysbus. To perform this conversion, two subroutines; *slv2op* and *op2slv* are also defined in the package. These allow the sysbus signal to be read, decoded and the corresponding opcode used in the ASM for moving between states.

To test these components, a test bench was created to place signals onto the sysbus corresponding to opcodes in the ASIC package. If the register and sequencer were operating correctly, the opcode output would change accordingly and output a command corresponding to

the binary signal input. No errors were found and the Instruction Register section of the ASIC was deemed to be operating successfully.

Creation and Testing of SHA-1

Once all parts of the ASIC were found to be operating correctly individually, they were integrated into a complete system through the use of packages and component instantiation in VHDL. This gives a partitioned design, which is simpler for addition and debugging during tests.

The *package* command is used in conjunction with *package body* to declare constants and functions used across the whole ASIC [38]. In this example, items such as the bus size (word_w), opcode size (op_w) and conversion of opcode to bus message (slv2op and op2slv) were declared in the package. The package body is used to add definition to slv2op and op2slv. Using the *case* command, the corresponding opcode is outputted based on the input present on the sysbus. This only allows one item to be executed in the array present, but since this ASIC is operating sequentially, this is acceptable for the design. This package is declared in all parts of the ASIC using the syntax *use work.asic_defs.all* in the library definitions. Through the use of a package, updates to the ASIC core take place by changing constant values, as these will permeate through all systems. This technique was used to increase the number of opcodes by only changing the table and constant value within the package. In a non-partitioned design, a change like this would require a complete redesign.

Once all parts of the ASIC are designed and operating successfully, they are defined as components at the top level. An *entity* called ASIC is created with only the externally connected inputs and outputs declared. All other internal signals are defined within the *architecture* and assigned to the relevant component through the *port map* command. This allows VHDL to create the design with all relevant connections automatically set. A syntax check on this top-level program initially identified issues with its connection to some components. The errors were found to be caused by the automatic addition of some unused libraries by the Xilinx tool. These libraries were causing conflicts within the ASIC and errors on compilation. Once these libraries were removed, all parts of the ASIC compiled correctly, leading to simulation of the entire system through a test bench.

Initially testing was performed on all sections as separate entities. The test benches were designed to test all aspects of operation and feasible conditions of use to identify any issues in the modules [55]. Through this separate testing, some small coding errors were identified and corrected prior to their integration in to the main ASIC. Most errors identified were omissions of commands such as setting the tri-state register to high impedance when not requested to output to the bus. An omission such as this could cause the system bus to be used by two components simultaneously and therefore cause communication collisions. As no protocol to identify this

(such as CDMA [56]) was implemented along with its associated error checking, any collisions could cause severe issues to the ASIC operation. Therefore, the correction of this code was critical to allowing successful operation of all modules within the system. Results from this initial testing can be seen in the Results section, with code for these and their test benches in **Error! Reference source not found.**.

Once all aspects of the ASIC operation as separate entities was complete, the VHDL files were connected together with a component level declaration and an overall test bench designed. As the ASIC progresses through the FSM automatically, this test bench simply operated a clock and provided a reset pulse to start the ASIC, as well as place the message into the I/O buffer. As the firmware program took several thousand lines of code, sections were checked with several test programs loaded into ROM which would check the operation of all components during the instruction fetch and execution phase. These programs concentrated on running each opcode in turn and ensuring that items could be called from RAM, placed into a functional block and executed. Outputs from the functional blocks were checked against the known correct answers using the *assert* statement. This waited until the *msg_out* pin was high and then would compare the *sysbus* to the value stored within the test bench. Any incorrect answers would cause an error message to be reported in the console window of Xilinx. All tests were completed successfully, allowing development of the full program to take place.

The SHA-1 program totalled 5200 lines of code. This is a large program, but this is because all instructions were placed in-line. With further time and knowledge of firmware programming, this code could be reduced using branch instructions to reset the PC and only execute instructions as part of an FSM similar to that in Figure 10. With the clock running at 10ns, this gives a throughput of 1.32Mbit/sec when calculated with Equation 1. The use of a faster clock would increase throughput, but also increase dynamic power dissipation. Practically however, a ten-time increase to a 1GHz clock would be practical, which would give this system in its current form a throughput of 13.2Mbit/sec. A further development for this ASIC would be to reduce the instruction set and possibly unroll functions, as this could increase the throughput to a level of practical use.

To test whether the design could be implemented, Xilinx was used to synthesise the components for FPGA use. A QPRO Virtex4 FPGA was chosen as this was known to be capable of supporting the ASIC size with the full firmware program loaded and had been used in previously published papers on SHA Hardware Implementation [57]. The ASIC successfully routed with a total register count of 3258 used. This investigation was not taken further, as the primary goal of the project was simulation, not to synthesise the unit into an FPGA. However, this shows that the design can be implemented by an automated tool and therefore could be used in a commercial application.

Development of Integrated Architecture

Once the SHA-1 ASIC was deemed to be operating successfully, the addition of SHA-2 to the core was made. Due to the modular design used, minimal additions needed to be made to give this chip SHA-2 capabilities: A larger RAM and ROM was declared, to contain the extra IV and K values used by SHA-2 for its round calculations, along with the new program, and the addition of right linear and circular shifters to the shifter logic block. The ROM size was increased from 6100 lines to 22000 lines due to the large size of the SHA-2 program.

To allow use of these new parts, two new opcodes had to be introduced to allow the shifter to operate in one of its three methods of operation. Due to the method employed for the opcode creation, along with the package definition used, this work was minor and added further functionality to the ASIC. These additional items of logic may allow SHA-3 to be used on this ASIC with a simple firmware program upgrade, if the components required by this are already declared within the architecture.

The major change for the addition of SHA-2 is the addition of new code within the program to allow execution of this hash function. This code is placed at a known point within the memory, which a hard jump can be made to if the selection pin is set to SHA-2. This jump is performed by setting the count variable in the Program Counter to the value of the branch and allowing execution from this point. The PC will then increment normally and progress to the end of the program, where the ASIC will again enter a wait mode until reset prior to the next message arriving.

Since all sections of the ASIC were unchanged, with the exception of the Logical Shifter, these components did not require re-testing. Instead, the program was modified as above and tested as a whole entity.

Pre-processing

SHA-2 takes the 512-bit input message and expands to 64 blocks of 32 bits each. This is performed using two variables S0 and S1, created using previous blocks passed through the right circular shift and the Exclusive OR operation. These are then added to previous blocks within the ALU and give the blocks 17-63. This is similar to the program for SHA-1, and only required the addition of the right-circular and logical shifter to the shifter block, as well as two extra opcodes to allow this part of the logic to execute. The design of a right shifter is identical to the left circular shift, except the LSB now becomes the MSB, rather than the other way round. Due to testing of this at SHA-1, the entity was not retested individually, but instead as part of the integrated program.

Round Calculations

The round calculation in SHA-2 is constant for all 64 rounds. Six variables are used, all using a mixture of logical, shifting and addition operations, as well as adding the round-dependant blocks and constants to create the eight hash values for the round. Therefore, these variables were calculated using already existing blocks from SHA-1 and were not individually tested. Instead, as with the pre-processing, they were tested within the final program to check for a successful output.

The logical values were as follows:

- S0 = (a ROTR 2) XOR (a ROTR 13) XOR (a ROTR 22)
- MAJ = (a AND b) XOR (a AND c) XOR (b AND c)
- T2 = S0 + MAJ
- S1 = (e ROTR 6) XOR (e ROTR 11) XOR (e ROTR 25)
- CH = (e AND f) XOR ((NOT e) AND g)
- T1 = h + S1 + CH + k[i] + w[i]

Where k[i] is the round constant and w[i] is the message round value

These values are added to the variables a – h, which are added to the initial hash values h0 – h7. This is repeated 64 times, giving a hash output of h0 – h7 after this is complete.

Increase in Throughput

Following the completion of the experiments, methods to increase throughput were investigated. Two key methods were identified: The refinement of code and the use of Very Long Instruction Word (VLIW) coding to allow functional blocks to write directly to one another, rather than using the RAM as an intermediary.

Investigations of the Code can be seen in Figure 18, this shows the majority of opcodes used in SHA-1 are the shift-left command, mostly due to the left-shift of B 30-times during round calculation. This could be reduced in the improved architecture by using the right-shift command available due to its use in SHA-2. 30 left-shifts are identical to 2 right-shifts when employing a circular shift and this would reduce the time to perform a round calculation by 25%.

The second most common opcode used is the storage and recalling of ALU data for the round calculation. A method to reduce this would be the introduction of two accumulators within the ALU, allowing addition of either Bus/Accumulator or Accumulator/Accumulator. Due to time constraints, this was not carried out but is considered as a further improvement if this project were to be revisited at a later date.

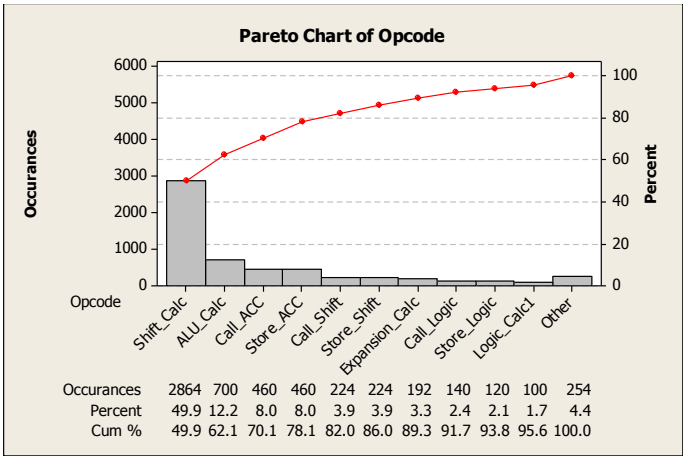

Figure 18: Pareto Graph of OpCode Use

During the expansion calculations, data is stored in RAM after being XOR'd, followed by being passed to the shifter block for a left circular shift. If the expander output could be passed directly to the shifter, this would save one opcode being used and make the expander block 12% more efficient. To implement this for all blocks would require an increase in opcodes from the current 20 to 33. This would allow all blocks to pass data directly to one another and reduce regular RAM accesses. Initial tests of this by directly passing information from the shifter to the ALU proved successful, but this could cause issues with a shared bus, as RAM would not know when other elements were accessing the bus and issues could result. To prevent this, a mutual exclusion element or other such form of token bus could be used, but this would affect throughput and could make the system slower than without the use of VLIW. Further work on this area has already been proposed by Kakarountas [34] and is further endorsed by this work.

Since all items output their results to the tri-state registers before outputting to the sysbus, an investigation into parallelism was also made. Items not using the sysbus for an input value could be run simultaneously and therefore reduce the operation time of the ASIC. This was initially tested by running the Shifter and ALU during the same FSM state, but was not pursued further due to time constraints and the level of redesign required to implement this successfully.

Comparison against published ASIC/FPGA Implementations

SHA-1			
Implementation	**Logic Blocks Used (FPGA)**	**Clock Frequency (MHz)**	**Throughput (Mbit/sec)**
Kakarountas [34]	950	98.7	2526.7
Sklavos [58]	1004	42.9	119
Lee	2894	118	5900
This Work	3528	100	1.31
SHA-1 and 2 Integrated Architecture			
Implementation	**Logic Blocks Used (FPGA)**	**Clock Frequency (MHz)**	**Throughput (Mbit/sec)**
Chaves	565	227	1420
Sklavos [58]	2384	74	291
Khalil	4489	50	644
This Work (SHA-2)	6836	100	0.49

Table 3: Comparison of ASIC to Current Published Work

Results

Note: All throughput calculations are based on Equation 1

Logic F_t Calculation

Objectives

To load Tri-state register with data, output data to system bus, perform AND, OR, NOT and XOR logic and output results to system bus

Test bench pseudo code

- Load system bus with value and load to Logic register
- Load register to system bus, ensure value matches initial value
- Perform Logical AND on value in register with matching value in system bus, ensure output is correct
- Perform Logical AND on value in register with non-matching value in system bus, ensure output is correct
- Perform Logical OR on value in register with non-matching value in system bus, ensure output is correct
- Perform Logical NOT on value in register, ensure output is correct
- Perform Logical XOR on value in register with matching value in system bus, ensure output is correct

Results

- Value initially loaded: 0x00000001
- Value returned before Logical AND: 0x00000001. Result: Pass
- Value returned from AND of 0x00000000 and 0x00000001: 0x00000000. Result: Pass
- Value returned from AND of 0x00000001 and 0x00000001: 0x00000001. Result: Pass
- Value returned from OR of 0x00000000 and 0x00000001: 0x00000001. Result: Pass
- Value returned from NOT of 0x00000001: 0xFFFFFFFE. Result: Pass
- Value returned from XOR of 0xFFFFFFFE and 0x550aa333: 0xaaf55cd. Result: Pass
- Time taken for AND Function: 40ns, giving throughput of 800Mbit/sec

Conclusion

Logic block accepts and outputs values successfully and performs all logic functions correctly. Therefore this component operates successfully.

Circular Shift Register

Objectives

To load Tri-state register with data, output data to system bus, rotate data in logical and circular shift, with output to system bus

Test bench pseudo code

- Load system bus with value and load to Shift Register
- Load register to system bus, ensure value matches initial value
- Execute shift register, output result, ensuring one left shift
- Execute shift register until loaded value is beyond most significant bit, output result, ensuring values have moved to least significant bit and therefore a circular shift has been successful

Results

- Value initially loaded: 0x40000000
- Value returned before shift: 0x40000000. Result: Pass
- Value returned after one shift: 0x80000000. Result: Pass
- Value returned following second shift: 0x00000001. Result: Pass
- Time taken for one shift function: 30ns, giving throughput of 1.067Gbit/sec
- Time for 30 shift operation (longest in SHA-1): 320ns, giving throughput of 10Mbit/sec

Conclusion

Circular shift register successfully accepts values, performs left shift and wraps shift in a circular manner. Therefore this component operates successfully

Modulo-32 Adder

Objectives

To load Tri-state register with data, output data to system bus, perform simple addition and overflow addition, with output to system bus

Test bench pseudo code

- Load system bus with value and load to Accumulator register
- Load accumulator to system bus, ensure value matches initial value
- Perform addition of values on system bus and in accumulator, ensuring correct result
- Perform addition of values known to cause overflow, showing modular addition takes place

Results

- Value initially loaded to accumulator: 0x00000001
- Value returned before addition: 0x00000001. Result: Pass
- Value returned from addition of 0x00000001 and 0x00000001: 0x00000002. Result: Pass
- Value returned from addition of 0x00000002 and 0xFFFFFFFE: 0x00000000. Result: Pass
- Time taken for addition operation: 40ns, giving throughput of 800Mbit/sec

Conclusion

The Adder successfully accepts and returns values from the accumulator, as well as performing addition and modular addition successfully. Therefore this component is acceptable for use.

ROM

Objectives

To extract data from specified memory locations in ROM to system bus

Test bench pseudo code

- Load MAR with address 0x00000000
- Command readout of memory address in MAR to MDR
- Output MDR to system bus
- Repeat for memory addresses 0x0000001, 0x0000008, 0x0000007 and 0x0000003

Results

- Result for readout of memory address 0x00000000: 0x67452301. Result: Pass
- Result for readout of memory address 0x00000001: 0xEFCDAB89. Result: Pass
- Result for readout of memory address 0x00000008: 0xCA62C1D6. Result: Pass
- Result for readout of memory address 0x00000007: 0x8F1BBCDC. Result: Pass
- Result for readout of memory address 0x00000003: 0x10325476. Result: Pass
- Time taken for readout operation: 30ns, giving throughput of 1.067Gb/sec

Conclusion

The ROM Module successfully stores constants within its addresses and outputs the corresponding constant based on the address in the MAR. Therefore, this section of the circuit operates as specified.

RAM

Objectives

To read and write data from specified memory locations in RAM to system bus

Test bench pseudo code

- Load MAR with address 0x00000000
- Load MDR with data 0x0F0F0F0F
- Write contents of MDR to RAM
- Repeat for addresses 0x00000001 and 0x0000000F with data 0x0E0E0E0E and 0xF0F0F0F0 respectively
- Load MAR with address 0x00000000
- Command readout of memory address in MAR to MDR
- Output MDR to system bus
- Repeat for addresses 0x00000001 and 0x0000000F

Results

- Readout of address 0x00000000: 0x0F0F0F0F. Result: Pass
- Readout of address 0x00000001: 0x0E0E0E0E. Result: Pass
- Readout of address 0x0000000F. 0xF0F0F0F0. Result: Pass
- Time taken for write operation: 30ns, giving throughput of 1.067Gb/sec
- Time taken for read operation: 30ns, giving throughput of 1.067Gb/sec

Conclusion

The RAM Module successfully stores data within its addresses and outputs the corresponding constant based on the address in the MAR. Therefore, this section of the circuit operates as specified.

Instruction Register

Objectives

To ensure instructions presented on the system bus are correctly interpreted and outputted by the sequencer

Test bench pseudo code

- Load value 0x80000000 to system bus
- Load into sequencer and ensure correct opcode is called
- Repeat for values 0x40000000, 0x00010000 and 0x00008000

Results

- Opcode for value 0x80000000: “logic_calc4”. Result: Pass
- Opcode for value 0x40000000: “logic_calc3”. Result: Pass
- Opcode for value 0x00008000: “store_acc”. Result: Pass
- Opcode for value 0x00010000. “store_expansion”. Result: Pass

Conclusion

The instruction register successfully stores data from the system bus when requested. This is then passed to the sequencer, which outputs the correct operation based on the decoded Opcode. Therefore this section of the program operates correctly.

Program Counter

Objectives

To ensure Program Counter can jump to a value on the system bus and increment this value when the increment flag is asserted, as well as output this value onto the system bus

Test bench pseudo code

- Load system bus with value 0x00000002
- Load system bus to Program Counter
- Output Program Counter register to system bus
- Increment Program Counter and output result, ensuring increment operation has taken place

Results

- Value loaded to system bus: 0x00000002
- Value outputted from Program Counter: 0x00000002. Result: Pass
- Value outputted from Program Counter following one increment operation: 0x00000003. Result: Pass
- Value outputted from Program Counter following second increment operation: 0x00000004. Result: Pass
- Time taken for operation: 30ns, giving throughput of 1.067Gb/sec

Conclusion

The program counter can accept and therefore jump to a value presented on the system bus, as well as increment when commanded. Therefore this section of the circuit operates as specified.

Expander

Objectives

To load Tri-state register with data, output data to system bus, perform XOR logic and output results to system bus

Test bench pseudo code

- Load system bus with value and load to Logic register
- Load register to system bus, ensure value matches initial value
- Perform Logical XOR on value in register with matching value in system bus, ensure output is correct

Results

- Value initially loaded: 0xAA000000
- Value returned before Logical XOR: 0xAA00000000. Result: Pass
- Value returned from XOR of 0xAA00000000 and 0x82000000: 0x28000000. Result: Pass
- Time taken for XOR Function: 40ns, giving throughput of 800Mb/sec

Conclusion

Expander block accepts and outputs values and performs logic functions correctly. Therefore this component operates successfully

Finite State Machine

Objectives

To ensure the FSM successfully sets flags upon entry to a state and moves to the state specified by the relevant opcode.

Test bench pseudo code

- Set opcode “store_acc” and ensure FSM progresses through as per Figure 11
- Repeat for “store_ROM”, “Call_shift” and “shift_calc”

Results

- State order for “store_acc”: s0, s1, s2, s3, s8, s9, s0. Result: Pass
- State order for “store_ROM”: s0, s1, s2, s3, s4, s22, s23, s9, s0. Result: Pass
- State order for “call_shift”: s0, s1, s2, s3, s10, s11, s0. Result: Pass
- State order for “shift_calc”: s0, s1, s2, s3, s10, s15, s16, s0. Result. Pass
- Time taken for progression: 10ns per state. Best case 60ns, worst case 80ns, giving range of throughput from 533Mbit/sec to 400Mbit/sec per instruction

Conclusion

The FSM successfully identifies the store, call and execute branches of operation and jumps accordingly when these occur. Therefore, this section of the ASIC is deemed acceptable for use.

SHA-2 Logical and Circular Shift Register

Objectives

To load Tri-state register with data, output data to system bus, rotate data in logical and circular shift, with output to system bus

Test bench pseudo code

- Load system bus with value and load to Shift Register
- Load register to system bus, ensure value matches initial value
- Execute shift register, output result, ensuring one left shift
- Execute shift register until loaded value is beyond most significant bit, output result, ensuring values have moved to least significant bit and therefore a circular shift has been successful
- Perform right circular shift on values in shift register to ensure right circular shift operates
- Perform right logical shift on values in shift register to ensure right circular shift operates

Results

- Value initially loaded: 0x40000000
- Value returned before shift: 0x40000000. Result: Pass
- Value returned after one shift: 0x80000000. Result: Pass
- Value returned following second shift: 0x00000001. Result: Pass
- Value returned following right circular shift: 0x80000000. Result: Pass
- Value returned following one right logical shift: 0x40000000. Result: Pass
- Value returned following 31 logical shifts: 0x00000000. Result: Pass
- Throughput same as left circular shift register

Conclusion

Multi-function shift register successfully accepts values, performs left and right shift and wraps shift in a circular manner, as well as logically when requested. Therefore this component operates successfully

SHA-1 Test 1

Objectives

To load RAM with 512 bit message, perform all logic, arithmetic and shifting functions, storing the result to RAM and outputting to the system bus.

Test bench pseudo code

- Load value from RAM to shifter
- Perform Left Circular Shift
- Load value from shifter to RAM
- Output result
- Place result in ALU
- Add with second value
- Store result in RAM
- Output result
- Load value to Logic
- Perform NOT
- Perform AND with second value
- Perform XOR with second value
- Store result in RAM
- Output result

Results

- Value loaded to shifter: 0x67452301
- Value after left shift: 0xCEA84602. Result: Pass
- Sum loaded to ALU: 0xCEA84602 + 0xEFCDAB89
- Value after summation: 0xBE57F18B. Result: Pass
- Value loaded to logic: 0xBE57F18B
- Value after NOT: 0x41A80E74. Result: Pass
- Value after AND with 0xEFCDAB89: 0x41880A00. Result: Pass
- Value after OR with 0xBE57F18B: 0xFFDFFB8B. Result: Pass
- Value after XOR with 0xEFCDAB89: 0x10125002. Result: Pass

Conclusion

All functional blocks in SHA-1 Architecture successfully read and write to RAM with a simple sequential program stored in ROM successfully executing all logical and arithmetic functions. Therefore, all items within the ASIC operate successfully

SHA-1 Test 2

Objectives

To load RAM with 512 bit message, perform expansion from 16 to 80 32-bit blocks, complete 80 rounds of hashing and output 160-bit Hash Function

Test bench pseudo code

- 16 blocks of message in RAM expanded to 80 blocks
- 80 round specific logic, shift and addition functions
- Update hash variables
- Output hash function onto system bus

Results

- Hash Output for an empty Hash Function:
    - 0xDA39A3EE
    - 0x5E6B4B0D
    - 0x3255BFEF
    - 0x95601890
    - 0xAFD80709
- Hash Output for ASIC:
    - 0x1EED8838
    - 0x0DC98439
    - 0x3326A454
    - 0x61C9888F
    - 0xCC6F23B6
- Result: Fail
- Time to perform hash function and enter sleep state: 1.6µs
    - 1.3Mbit/sec total throughput

Conclusion

ASIC executes 80 rounds of hash function and outputs result. However, hash function is not correct. Believed to be issue with firmware program caused during creation.

SHA-1/2 Integrated Architecture Test 1

Objectives

To load RAM with 512 bit message, perform all logic, arithmetic and shifting functions, storing the result to RAM and outputting to the system bus.

Test bench pseudo code

- Load value from RAM to shifter
- Perform Left Circular Shift
- Place result in ALU
- Add with second value
- Load result to Logic
- Perform NOT
- Perform AND with second value
- Perform XOR with second value
- Perform Right Circular Shift with result
- Perform Right Logical Shift three-times with result
- Store result in RAM
- Output Result
- Jump to ALU section and repeat ALU, Logic and Right Shifter Sections

Results

- Value loaded to shifter: 0x67452301
- Value after left shift: 0xCEA84602. Result: Pass
- Sum loaded to ALU: 0xCEA84602 + 0xEFCDAB89
- Value after summation: 0xBE57F18B. Result: Pass
- Value loaded to logic: 0xBE57F18B
- Value after NOT: 0x41A80E74. Result: Pass
- Value after AND with 0xEFCDAB89: 0x41880A00. Result: Pass
- Value after OR with 0xBE57F18B: 0xFFDFFB8B. Result: Pass
- Value after XOR with 0xEFCDAB89: 0x10125002. Result: Pass
- Value after Right Circular Shift: 0x08092801. Result: Pass
- Value after three Right Logical Shifts: 0x01012500. Result: Pass
- Sum loaded to ALU: 0x01012500 + 0xEFCDAB89
- Value after summation: 0xF0CED089. Result: Pass
- Value loaded to logic: 0xF0CED089

- Value after NOT: 0x0F312F76. Result: Pass
- Value after AND with 0xEFCDAB89: 0x0F012B00. Result: Pass
- Value after OR with 0xF0CED089: 0xFFCFFB89. Result: Pass
- Value after XOR with 0xEFCDAB89: 0x1002500. Result: Pass
- Value after Right Circular Shift: 0x08012800. Result: Pass
- Value after three Right Logical Shifts: 0x01002500. Result: Pass

Conclusion

All functional blocks in SHA-2 Architecture successfully read and write to RAM with a simple sequential program stored in ROM successfully executing all logical and arithmetic functions. Therefore, all items within the ASIC operate successfully

SHA-1/2 Integrated Architecture Test 1

Objectives

To load RAM with 512 bit message, perform all logic, arithmetic and shifting functions, storing the result to RAM and outputting to the system bus.

Test bench pseudo code

- Jump to Right Shift Section of Program (using SHA-2 Selection)
- Perform Right Circular Shift
- Perform Right Logical Shift
- Perform Right Logical Shift
- Perform Right Logical Shift
- Output Result
- Jump to ALU section and repeat ALU, Logic and Right Shifter Sections from Test 1

Results

- Value loaded to shifter: 0x67452301
- Value outputted from Shifter: 0x16745230. Result: Pass
- Result of 0x16745230 + 0xEFCDAB89 = 0x0641FDB9. Result: Pass
- Result of Logical Tests: 0x00005430. Result: Pass
- Result of Shift Tests: 0x00000543. Result: Pass

Conclusion

All functional blocks in SHA-2 Architecture successfully read and write to RAM with a simple sequential program stored in ROM successfully executing all logical and arithmetic functions. The Recall jump for more message and hard jump to select SHA-2 both operate successfully. Therefore, all items within the ASIC operate successfully

SHA-1/2 Integrated Architecture Test 3

Objectives

To load RAM with 512 bit message, complete 64 rounds of hashing and output 256-bit Hash Function. Followed by Reset and completion of SHA-1 Tests.

Test bench pseudo code

- Hard Jump to SHA-2 section of ROM
- 64 round specific logic, shift and addition functions
- Update hash variables
- Output hash function onto system bus
- Reset
- 16 blocks of message in RAM expanded to 80 blocks
- 80 round specific logic, shift and addition functions
- Update hash variables
- Output hash function onto system bus

Results

Hash Outputs for Text String “ABCD”

SHA-1		**SHA-2**	
Expected	**Actual**	**Expected**	**Actual**
0xFB2F85C8	0xFA4ACC36	0xE12E115A	0x3E351546
0x8567FBC8	0xE6AF0398	0xCF4552B2	0x3E4F2822
0xCE9B799C	0x70A29222	0x568B55E9	0xD11E4213
0x7C54642D	0xBD8BD55F	0x3CBD3939	0x84228C72
0x0C7B41F6	0x3F36B0E4	0x4C4EF81C	0x64330F24
		0x82447FAF	0xB3536C6A
		0xC997882A	0x042AA085
		0x02D23637	0x1596E897

Result: Fail

Conclusion

ASIC runs both hash functions and outputs results. However, hash function is not correct. Believed to be issue with firmware program caused during creation.

SHA-1/2 Integrated Architecture Test 4

Objectives

Load Shifter with value from RAM, perform shift, use direct pass VLIW instruction to move direct to ALU, perform arithmetic, use direct pass VLIW instruction to move direct to Logic, perform Logic, store to RAM and output.

Test bench pseudo code

- Load Shifter with value from RAM and perform Circular Left Shift
- Pass Result to ALU and add with second value from RAM
- Pass Result to Logic, perform NOT, AND, OR and XOR
- Store Result in RAM and Output

Results

- Value loaded to shifter: 0x67452301
- Value outputted from Shifter: 0xCE8A4602. Result: Pass
- Result of 0x CE8A4602 + 0xEFCDAB89 = 0xBE57F18B. Result: Pass
- Result of NOT 0xBE57F18B: 0x41A80E74. Result: Pass
- Result of 0x41A80E74 AND 0xEFCDAB89: 0x41880A00. Result: Pass
- Result of 0xEFCDAB89 OR 0x67452301: 0x67CD2B01. Result: Pass
- Result of 0x67CD2B01 XOR 0xEFCDAB89: 0x88008088. Result: Pass
- Value outputted on System Bus: 0x88008088. Result: Pass

Conclusion

This test architecture has allowed completion of SHA-1 Test 1 in 11 instructions instead of 15, giving a 26% overhead reduction by not using RAM as an intermediate step between calculation blocks. All results are correct, and the ASIC functions as expected. Therefore, this instruction set could be expanded to allow a throughput improvement across the full SHA calculation.

Conclusions

While this ASICs throughput was below the suggested goal of 40Gbit/sec, the goal of creating an ASIC capable of creating a hash function in both SHA-1 and SHA-2 was achieved. The modular concept used allows the ASIC to be used for multiple functions, with only a simple firmware change in ROM required. Therefore, this chip is capable of performing the SHA-1 and 2 Hash Functions, as well as the possibility of performing other commonly used functions such as MD5, or some of the shortlisted SHA-3 possibilities such as JH or Skein.

The actual hash results outputted were not correct, which is believed to be an issue within the firmware program. This program was created at a low-level without the use of a complier or syntax checker. Therefore, a small error could have been generated which has not been successfully debugged and would be difficult to find without a time consuming line-by-line simulation of the program. If an established microcontroller or FPGA were used instead of a custom-designed ASIC, the program could be created in a higher level language such as C and compiled into an assembly instruction set. If the goal still were to create an ASIC with no existing compiler, a program such as AWK or the C Pre-processor could be used to compile the program from a high-level language. This would make the code more user-readable and allow debugging in a more methodical and easier manner. With extra time to perform these tasks, it is highly likely that the faults within the program could be identified and removed, giving correct hash functions for both SHA-1 and SHA-2.

Hardware implementation of cryptographic functions is not a new concept. The Trusted Computing Group has ensured a chip capable of performing these has been fitted to Trusted Platform Modules since its inception in 2003. However, the chips in current use have SHA-1 as their onboard hash function. While this is suitable currently, papers identifying theoretical collisions have been published. Therefore, the next round of TPMs should upgrade to a higher level hash function to increase security. However, they must also be backwards compatible with SHA-1 to allow current systems to continue communicating without any hardware upgrades. It is the authors' belief that this ASIC is capable of performing this function. With extra development time, issues with throughput could be removed by increasing the base clock rate, introducing parallel operations and possibly unrolling some slower operations such as the 30 left shift present in SHA-1. The key causes of slow throughput have been identified by a statistical analysis of the program in its current form and this would give any future work a key point to begin program or hardware improvements if the decision was made to optimise the ASIC for research or commercial use.

Through the use of VHDL, no limitations have been placed on the manufacturing techniques that can be used to create this ASIC. Therefore, it could be constructed on FPGA or using existing CMOS processes. The architecture already has been successfully synthesised in Xilinx for the xq4vlx160 FPGA, and could no-doubt be used on others also. Since most other work has been completed at the 0.13-0.18µm level, it is believed that this ASIC could also be built using these dimensions without major issues occurring.

If further time and resource were available to develop this ASIC, throughput could be increased significantly. The low number of opcodes used in this ASIC greatly affected its speed as all items had to be written to RAM and fetched from RAM to be outputted. As has been initially investigated, work using VLIW instructions could take place to allow direct movement between modules or to I/O without the intermediate use of RAM. Performing this could reduce the instructions by 30% and therefore increase speed by this factor also. This; combined with code optimisation and use of better programming techniques could double the speed of the ASIC with minimal resource investment.

This project has the possibilities of further research into the use of other methods to increase throughput, or the reduction in power consumption for practical implementation into RFID or other wireless communications methods. Therefore, the author's recommendations for future work are as follows:

- Debug program using AWK to give correct SHA-1 and 2 outputs
- Optimise code to reduce time intensive left/right shifts in expansion calculations
- Implement direct passing code to prevent using RAM as an intermediate and save operation time
- Implement parallel operations of functional blocks to allow other operations to take place while items such as the shift block are running
- Add sleep functions to blocks to power down when not being used and save static power draw by the ASIC
- Synthesise architecture into FPGA and test for correct results and similar throughput to the simulation
- Construct architecture in ASIC and test for correct results and similar throughput to FPGA and simulation

As hash functions continue to grow in use, a chip capable of high throughput with advanced power management could be of great use within industry or specific markets within battery powered applications. A major user of this could be the rise in cloud computing, where systems request resources on demand and many users may share one item [59]. This will mean users may not store their files on a local system, instead having them placed in an area of the cloud and sent to the accessing device when needed. If items contain sensitive information, is it

important that this is kept safe and also that it is known someone else has not tampered with the file while it has been stored. For this, a mixture of symmetric and asymmetric ciphers in conjunction with hash functions will be the most pragmatic solution. The ASIC presented in this thesis gives one possible solution to an aspect of these security concerns. It is felt that with extra time and resource, this IC could close to the minimum speed required for a cloud computing setup (around 2-4Gbit/sec) and deliver a safe and reliable method of computing the SHA family of hash functions, with the possibility of upgrading the functionality without changing the hardware.

References

[1] C. I. Tangpong, Muhammad ; Lertpittayapoom, Nongkran "The Emergence of Business-to-Consumer E-Commerce," *Journal of Leadership & Organizational Studies,* vol. 16, pp. 131-140, 2009.

[2] B. C.-T. Ho, and Kok-Boon Oh, "An empirical study of the use of e-security seals in e-commerce," *Online Information Review,* vol. 33, pp. 655-671, 2009.

[3] J. Seberry and J. Pieprzyk, *Cryptography : an introduction to computer security*. New York ; London: Prentice Hall, 1989.

[4] R. R. Dube, *Hardware Based Computer Security Techniques to Defeat Hackers*. Hoboken NJ: John Wiley and Sons, 2008.

[5] S. Singh, *The code book : the secret history of codes and code-breaking*. London: Fourth Estate, 2000.

[6] J. Buchmann, *Introduction to cryptography*, [2nd] ed. New York: Springer-Verlag, 2004.

[7] P. C. v. O. Alfred J. Menezes, Scott A. Vanstone, *Handbook of Applied Cryptography*: CRC Press, 1996.

[8] P. Gutmann, D. Naccache, and C. C. Palmer, "When hashes collide," *IEEE Security and Privacy,* vol. 3, pp. 68-71, 2005.

[9] R. A. Mollin, *An introduction to cryptography*. Boca Raton: Chapman & Hall/CRC, 2001.

[10] H. Dobbertin, "The status of MD5 after a recent attack," *CryptoBytes,* vol. 2, pp. 1-6, 1996.

[11] A. B. B. den Boer, "Collisions for the compression function of MD5," in *Advances in Cryptology, Proceedings Eurocrypt'93*, 1994, pp. 293-304.

[12] H. Y. X. Wang, "How to break MD5 and other hash functions," in *Advances in Cryptology, Proceedings Eurocrypt'05*, 2005, pp. 19-35.

[13] Y. L. L. X. Wang, H. Yu, "Finding collisions in the full SHA-1," in *Advances in Cryptology, Proceedings Crypto'05*, 2005, pp. 17-36.

[14] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography engineering : design principles and practical applications*. Hoboken, N.J.: Wiley.

[15] N. Ferguson and B. Schneier, *Practical cryptography*. New York: Wiley, 2003.

[16] I. Ahmad and A. S. Das, "Analysis and detection of errors in implementation of SHA-512 algorithms on FPGAs," *Computer Journal,* vol. 50, pp. 728-738, 2007.

[17] A. G. Konheim and Ebooks Corporation., *Computer Security and Cryptography*. Hoboken: John Wiley & Sons Inc., 2007.

[18] A. Satoh and T. Inoue, "ASIC-hardware-focused comparison for hash functions MD5, RIPEMD-160, and SHS," in *International Conference on Information Technology: Coding and Computing, ITCC*, 2005, pp. 532-537.

[19] N. I. o. S. a. Technology, "Announcing the Standard for Secure Hash Standard," N. I. o. S. a. Technology, Ed. Washington DC, 1995.

[20] P. K. Yuen, *Practical cryptology and web security*. Harlow: Addison-Wesley, 2005.

[21] "Trusted Computing Group www.trustedcomputinggroup.org," 2009.

[22] L. Dadda, M. Macchetti, and J. Owen, "An ASIC design for a high speed implementation of the hash function SHA-256 (384, 512)," in *Proceedings of the ACM Great Lakes Symposium on VLSI*, 2004, pp. 421-425.

[23] R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis, "Cost-efficient SHA hardware accelerators," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 16, pp. 999-1008, 2008.

[24] J. R. Vacca, *Computer and information security handbook*. San Francisco, Calif.: Morgan Kaufmann, 2009.

[25] M. Zeghid, B. Bouallegue, A. Baganne, M. Machhout, and R. Tourki, "A reconfigurable implementation of the new secure hash algorithm," in *Proceedings - Second International Conference on Availability, Reliability and Security, ARES 2007*, 2007, pp. 281-285.

[26] W. H. Wolf, *Modern VLSI design : systems on silicon*, 2nd ed. Upper Saddle River, NJ: Prentice Hall PTR, 1998.
[27] H. Xia, H.-m. Ning, and J.-y. Yan, "The Realization and Optimization of Secure Hash Algorithm (SHA-1) Based on LEON2 Coprocessor," in *Computer Science and Software Engineering, 2008 International Conference on*, 2008, pp. 853-858.
[28] E. H. Lee, J. H. Lee, I. H. Park, and K. R. Cho, "Implementation of high-speed SHA-1 architecture," *IEICE Electronics Express,* vol. 6, pp. 1174-1179, 2009.
[29] L. Jiang, Y. Wang, Q. Zhao, Y. Shao, and X. Zhao, "Ultra high throughput architectures for SHA-1 hash algorithm on FPGA," in *Proceedings - 2009 International Conference on Computational Intelligence and Software Engineering, CiSE 2009*, 2009.
[30] C. Piguet, *Low-power CMOS circuits : technology, logic design and CAD tools*. Boca Raton, FL: Taylor & Francis, 2006.
[31] Z. Dai and N. Zhou, "FPGA implementation of SHA-1 algorithm," in *ASIC, 2003. Proceedings. 5th International Conference on*, 2003, pp. 1321-1324 Vol.2.
[32] G. Feng, P. Jain, and K. Choi, "Ultra-low power and high speed design and implementation of AES and SHA1 hardware cores in 65 nanometer CMOS technology," in *Electro/Information Technology, 2009. eit '09. IEEE International Conference on*, 2009, pp. 405-410.
[33] J. M. Rabaey and M. Pedram, *Low power design methodologies*. Boston: Kluwer Academic Publishers, 1996.
[34] A. P. Kakarountas, H. Michail, A. Milidonis, C. E. Goutis, and G. Theodoridis, "High-speed FPGA implementation of secure hash algorithm for IPSec and VPN applications," *Journal of Supercomputing,* vol. 37, pp. 179-195, 2006.
[35] D. ZiBin, Y. XueRong, S. JinHai, and C. Xing Yuan, "Accelerated Flexible Processor Architecture for Crypto Information," in *Pervasive Computing and Applications, 2007. ICPCA 2007. 2nd International Conference on*, 2007, pp. 399-403.
[36] M. Technologies, *Accelerating DSP Filter Loops with MIPS® CorExtend® Instructions*, 1.01 ed. Sunnyvale, CA: MIPS Technologies INC, 2008.
[37] G. Kane and J. Heinrich, *MIPS RISC architecture*. Englewood Cliffs, N.J.: Prentice Hall, 1992.
[38] D. Pellerin and D. Taylor, "VHDL made easy!," Upper Saddle River, N.J.: Prentice Hall, 1997.
[39] R. Lipsett, C. F. Schaefer, and C. Ussery, *VHDL : hardware description and design*. Boston: Kluwer Academic Publishers, 1989.
[40] J. Bhasker, *A VHDL primer*. Englewood Cliffs, NJ: Prentice Hall, 1992.
[41] M. Zwolinski, *Digital system design with VHDL*, 2nd ed ed. Harlow: Prentice Hall, 2004.
[42] R. Williams, *Real-time systems development*. Oxford: Butterworth-Heinemann, 2006.
[43] J. P. Hayes, *Computer architecture and organization*, 3rd ed. Boston: WCB/McGraw-Hill, 1998.
[44] M. Slater, *A Guide to RISC microprocessors*. San Diego: Academic Press, 1992.
[45] F. Anceau, *The architecture of microprocessors*. Wokingham, England ; Reading, Mass.: Addison-Wesley, 1986.
[46] S. G. Shiva and Ebooks Corporation., *Computer Design and Architecture*. Hoboken: Marcel Dekker Inc, 2000.
[47] R. Y. Kain, *Advanced computer architecture : a systems design approach*. Englewood Cliffs, N.J.: Prentice Hall, 1996.
[48] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev, "Deriving Petri Nets from finite transition systems," *IEEE Transactions on Computers,* vol. 47, pp. 859-882, 1998.
[49] NIST, "Tentative Timeline of the Development of New Hash Functions," Washington DC, 2008.
[50] J. Catsoulis, *Designing embedded hardware*, 1st ed. Sebastopol, CA: O'Reilly, 2002.
[51] J. R. Gibson, *Electronic logic circuits*, 3rd ed. London: Edward Arnold, 1992.
[52] M. R. Zargham, *Computer architecture : single and parallel systems*. Upper Saddle River, N.J.: Prentice Hall, 1996.

[53] J. F. Wakerly, *Digital design : principles and practices*, 3rd , updated ed. Upper Saddle River, N.J.: Prentice Hall, 2001.
[54] Wikipedia, "6-Cell SRAM Model." vol. 2010: Wikipedia, 2010, pp. 6-Cell Model of SRAM Diagram http://en.wikipedia.org/wiki/File:SRAM_Cell_(6_Transistors).svg.
[55] G. Russell and I. L. Sayers, *Advanced simulation and test methodologies for VLSI design*. London: Van Nostrand Reinhold (International), 1989.
[56] A. S. Tanenbaum and Safari Books Online (Firm), *Computer networks*, 4th ed. Upper Saddle River, NJ: Prentice Hall PTR, 2003.
[57] Xilinx, "Virtex-4Q FPGA Data Sheet," 27th April 2010 ed: Xilinx, 2010, p. 40.
[58] N. Sklavos and O. Koufopavlou, "Implementation of the SHA-2 hash family standard using FPGAs," *Journal of Supercomputing,* vol. 31, pp. 227-248, 2005.
[59] T. Anderson, "The Rise of Cloud Computing," in *The Guardian* Manchester, 2010.