YODA

Phase 2

Group 17

Daniel Donaldson
Gregory Ireland

VADER

Versatile Accelerated Digital Encryption Recovery

VADER is a digitally accelerated add-on hardware device designed to recover passwords using an acquired hashed password and hashing function. It is designed to be run on an FPGA to reconfigure itself best to the hashing function required

EEE4084F
10 May 2012

Table of Contents

Table of figures

1. Introduction

The YODA topic presents the reconfigurable hardware problem – it is no small task to find a use for the powerful reconfigurable hardware we have in this present day. We were tasked with creating a digital accelerator (YODA - Your Own **Digital Accelerator**), essentially an add-on card which speeds up processing for a particular solution. This report covers first a brief overview of the problem and solution. The skills required to complete such a task are investigated and then it moves on to explain the method we will use in order to solve the problem. The overall problem is modelled and a performance yardstick established – the golden measure. An evaluation of the design solution follows using the PC based golden measure and the digital accelerator based solution. The steps taken to evaluate the performance of the two implementations are discussed finally.

1.1 The problem and solution

The problem proposed is to speed up computationally expensive recovery of passwords for forensic purposes such as instances where a victim or suspect's password protected information could assist in an investigation. In order for the system to begin, the specific hashing function used to create the password hashes would need to be acquired; it is assumed that this is available as many commonly used hashing functions are indeed widely available. The hashed version of the password itself would also need to be acquired. The system will run parallelized functions on an FPGA to accelerate the recovery of the password. The solution for this is the VADER system. VADER is a digitally accelerated add-on hardware device designed to recover passwords using an acquired hashed password and hashing function. It will be designed using reconfigurable hardware such as an FPGA.

Further design problems (to be answered in this report) are:

- Possible decision to use one or more soft-core processors on the FPGA for implementing the cracking function or to control the system.
- How to implement a dictionary on the FPGA memory.
- Possibility of creating a reconfigurable computer, which changes its hardware depending on whether a dictionary or brute force attack is being run.
- Extent to which the functions should be parallelized (fine or coarse grained).
- Number of parallel computational components to use.
- How to effectively develop and test the software or HDL designs.
- How to correctly interface with the system using USB.

2. Skills identification

In order to successfully implement VADER onto a FPGA, a lengthy design process must be undertaken. This is the first skill that will be needed – engineering design. It is important that designing engineers have an understanding of reconfigurable computing, specifically that of an FPGA. Even more specifically, the engineers should be comfortable with learning ins and outs of the Nexys2/3 [1] as this will be the FPGA used for this project. A deep understanding of the reconfigurable and performance aspects of these FPGAs will be critical to the success of the VADER project [2].

Before the FPGA development can begin, however, we need to be acquainted with schematic and UML design. This will aid greatly in the system breakdown and identification. Once broken down into systems, we can begin to delve deeper into the hashing algorithms that we will be implementing in these systems. In order to create a “Gold Standard” the algorithms will need to be implemented on a PC. This requires competency in C programming, as well as the finding and use of reusable hashing libraries. In order to understand the performance metrics and compare them to the gold standard, an understanding of x86 and FPGA instructions should be present at a basic level.

Once the design stage has been finalized we can begin simulating, and eventually programming, the FPGA based solution. This will require proficiency in HDL programming as well as testing with an IDE. Specifically, VHDL programming skills will be used for this programming. Familiarity with the ISE development environment is necessary to properly simulate and demonstrate the code before it is implemented on the development boards. In order to finalise the solution, we will need to be able to program and test on the Nexys2 board.

3. Methodology

The methodology used for this project can be summarized by the following process flow diagram depicted in figure 1. The basic structure of the methodology is presented and a more in-depth explanation is outlined thereafter.

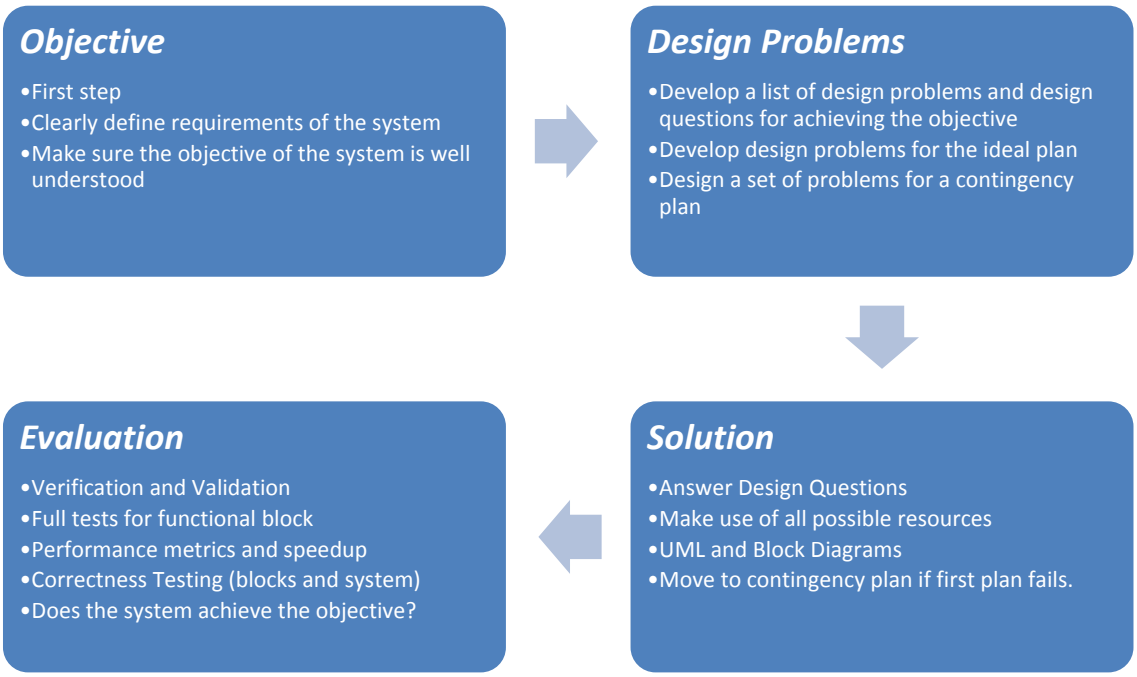

Figure 1: Project Methodology Process Flow

3.1 Project Objective

Deciding on the actual project objective is a vital first step in defining a project. The requirements of the system need to be clearly laid out and the project needs to be designed from the start to satisfy the desired objective. If the objective of the project is poorly understood there is an increased risk of the wrong system being developed. [3] [4]

The proposed “plan A” project objective would be to create a solution with maximum performance(for which metrics will be explained later) and maximum system versatility in terms of password recovery for different types of password hashing functions. This would be the idealized project plan, maximum resources would be put into the development of the best possible system but this plan would also likely have the most risks involved.

The methodology will also include a “plan B”, this plan would be implemented if the plan A failed or is not going to be completed in time. The difference from plan A is that plan B will be a significantly simplified version of plan A. The main objective of plan B would be to make a system with a faster development time. This would involve making the system simpler in terms of capability and mitigating project risks that could set the development back. Plan B should try to use as much working design that plan A already developed to try speed up the process. Performance would probably be lower, but so would total development time.

3.2 Design Problems

Once the project objective is well understood the design of the actual system can begin to take place. The engineers need to identify relevant design problems or design questions which through solving or answering should effectively meet the objective of the system. All of the problems that will be identified will have to be adequately suited to ensure time is spent where it is most valuable to achieving the solution.

For this project some design questions have already been identified in phase 1 and repeated in the introduction of phase 2, later in the Modelling and Analysis section a list of more in-depth design questions will be posed and answered.

Almost all of the design problems for plan A will be relevant for contingency plan B, some will simply be less relevant and others can be ignored. The design decisions for plan B would differ mostly in adding questions on how to improve, change, mitigate risks or speed up plan A. These are mentioned below:

- How to minimize development time, what aspects of the design would be time consuming or risky and how can these be simplified or removed without removing too much functionality?
- How to effectively reduce functionality without defeating the point of the system.
- How many hashing functions should be included?
- Can the system blocks themselves be simplified to make development faster and to make the design as a whole simpler?
- Can testing be simplified to speed up the evaluation stage?

3.3 Proposed Solution

Proposing a solution would be made possible by systematically solving and answering the design problems and questions identified while explicitly keeping the project objective in mind. Solving these problems may need multiple sources of answers for completion. Much of the information needed to solve these issues the engineer may already have but a more in-depth look at the system and external sources is likely needed to propose an effective solution.

Reference manuals or product datasheets would likely be an essentially valuable resource in effectively answering many design problems and can be used to get a good idea of how to compare candidate systems, and once chosen the information within would be essential for completing your solution with that system.

Block and UML diagrams would be fundamental in proposing a solution; they will be used for getting your ideas into a practical form and can be a powerful tool in breaking down the functionality of the system into understandable parts. Solutions to each of these parts of the system can then be tackled in more depth. Diagrams such as these are also very important in conveying your ideas to other people in the design process as they can then see the functionality you wish you incorporate and can more effectively add to change your designs.

Multiple solutions may exist and the best solution may not often be clear to the team, if possible keeping all solutions viable in the design until it is possible to test them for suitability would produce

the best result for the system.

Proposing a solution is no easy task; engineers might find that after battling with a particular design problem for a long time no feasible solution to the problem exists. This is where the plan b proposed would have to be considered; the problematic part of the system may have to be scrapped at the cost of functionality or simplified at the cost of performance. Ultimately the plan B solution used would need to try speed up the development as much as possible while minimizing performance lost or possible functionality or versatility being removed as a result.

3.4 Evaluation:

Once a proposed solution has been formulated Verification and Validation of the system will need to take place. Verification of the system mostly involves testing the system to see if it's actually doing what it was designed to do. I.e. is the system actually working, are the separate components working correctly etc? Validation is done to test if the system as a whole is the right product, i.e. does it fit the original specification and meet its purpose, and if it doesn't meet the objective the design process may have to be repeated to correct the problem. [4]

The testing of each hashing type encompassing its own specific hardware blocks would need to include a full retesting of the system except for the hash unspecific circuitry. This RC component of the system would likely significantly increase the evaluation time of the system, increasing for every separate hashing function implemented.

Although performance metrics such as throughput or latency to solution could be tested in the system these methods are all relative to another implementation and not fully relevant to the system. Latencies for example could be extremely long for finding a long password using the brute force attack and may not be very productive for testing the system. A more relevant performance metric would be the number of hashes per unit time, say hashes per second. This would allow the system performance to be tested without requiring the actual solution to be found.

Evaluation of the speedup seen by adding more computational blocks in parallel is vital for finding the correct number of processing blocks to include in the system. The maximum speedup will ultimately be limited by the sequential part of the processing which will mostly lie within each hashing function; Amdahl's law will limit the total possible speedup. [5]

Correctness tests would also need to be carried out, these could be done in different stages namely: individual processing blocks and the system as a whole.

Individual blocks would need to be tested for correct functionality such as the hashes themselves to see if the hashing function is correct, the test word generator block, the control circuitry, comparators etc. all need to be tested for consistency.

Finally the entire system needs to be validated. The system as a whole needs to be tested to see if it actually meets its original objectives and specification. This would include rating the performance of the system compared to the objective as well as rating the systems versatility. If the validation criteria are not met, the system may have to be redesigned and then re-implemented costing a lot of time and setting back the development significantly. [4]

4. Modelling and analysis

4.1 Modelling Design Problems

Detailed design problems to be answered while creating the model of the system:

1. Choice of FPGA. Based on number of LE's, interconnects, extra peripherals on the board, I/O pins, availability of LAN, USB, external memory are all important criteria for choosing the right FPGA.
2. Decision to create a RC system. By having an RC device, different hardware designed to implement different hashing functions can be programmed onto the device to allow multiple types of hashing functions to be cracked.
3. Decision to include a dictionary based password attack. Could speed up password recovery if password is in dictionary.
4. Implementation of brute force algorithm. How would one implement a brute force algorithm on the FPGA and make it able to output parallel test words.
5. How to interface with the PC. I.e. LAN, USB, Serial etc.
6. Decisions for storing the dictionary on the FPGA. Deciding whether to load it into RAM or use a FLASH based dictionary or both (FLASH loaded into RAM by system). The dictionary could also possibly be implemented using the FPGA itself to create a "RAM block"
7. Composition of "Functional processing blocks" and their usage. This would involve designing a set of digital blocks that will do the actual processing within the system. Each one of these blocks should be able to do its processing in parallel with other copies of the blocks.
8. Dealing with a likely memory bottleneck for dictionary attack. It is likely that the memory won't be able to be read fast enough to send a word to every processing block causing a processing slow down.
9. A decision on how many of these blocks to incorporate would need to be based on a few things, namely:
    a. Tests on the system showing speed versus processing blocks used.
    b. Making use of as much of the FPGA's PLEs to make sure the FPGA is being efficiently minimizing unused elements.
    c. Monitoring performance trade-offs that may include a slow-down of the system from control overhead or generation of test-passwords if too many functional blocks are used.

Here the proposed block and UML diagrams of the system are presented for reference. They are discussed and analysed in the following sections.

4.2.1 Block Diagram

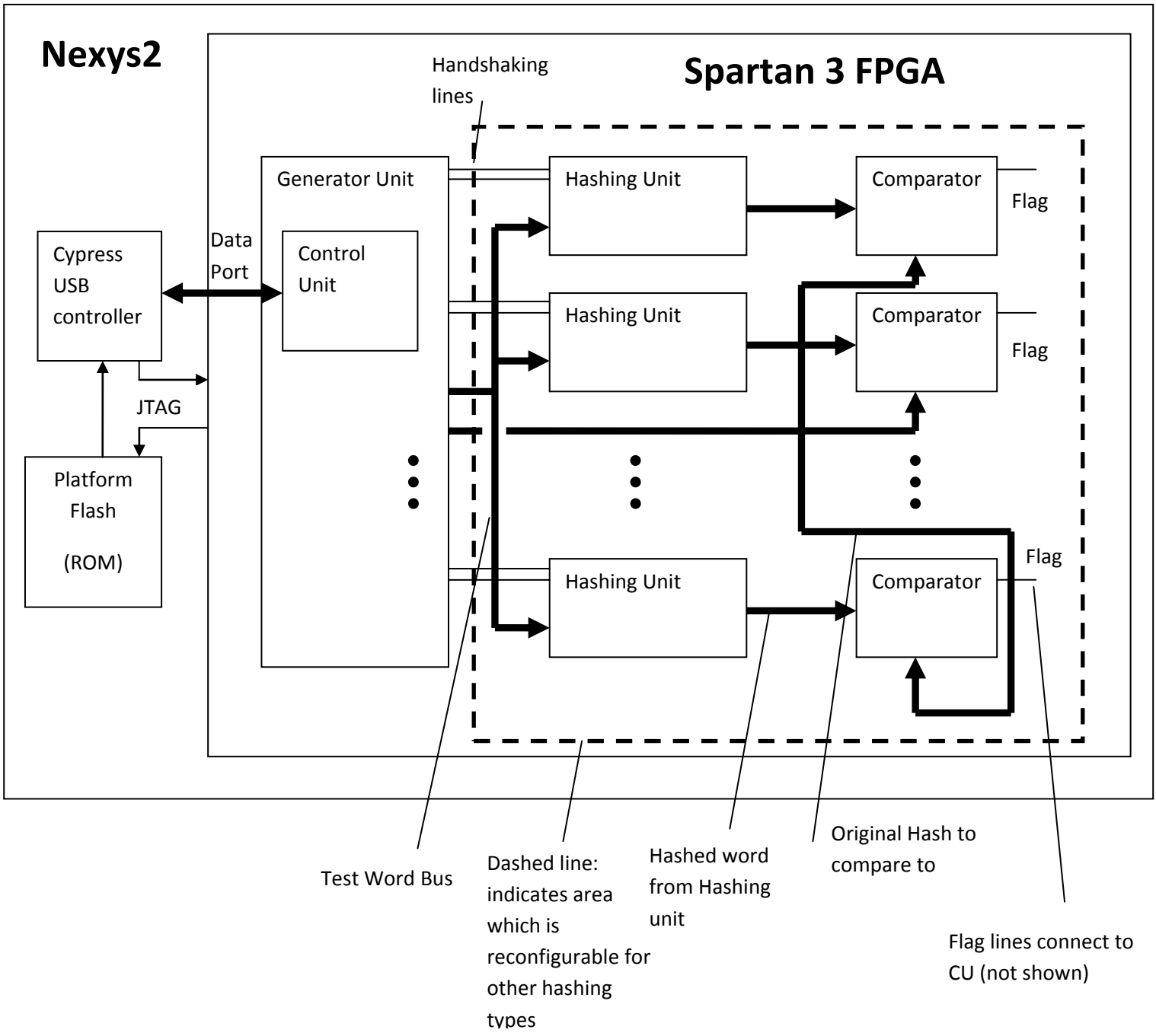

Figure 2: Block Diagram of Vader

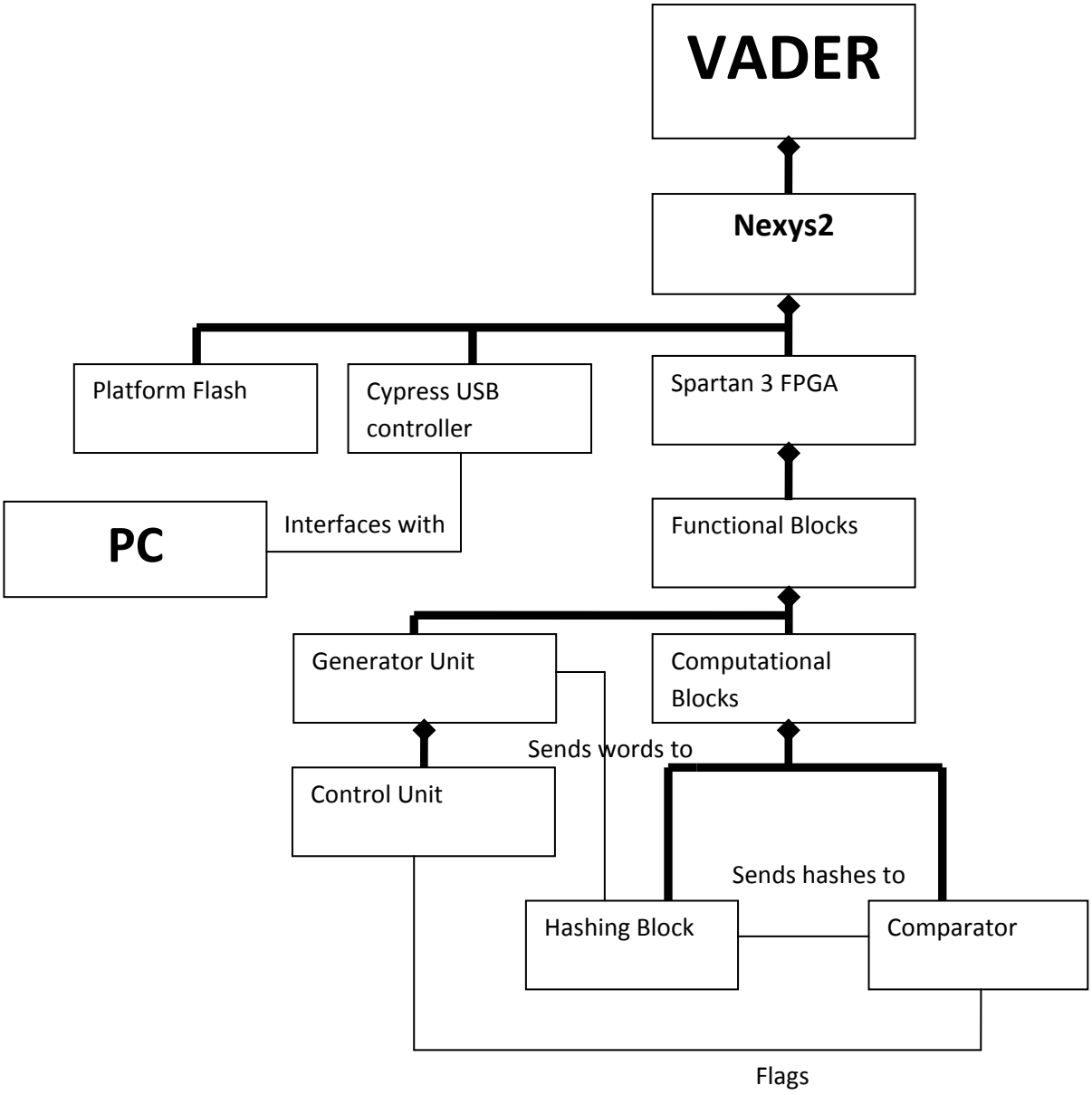

Figure 3: UML diagram of connected systems

4.3 Design Discussion:

The design discussion is mostly laid out in the form of how the design decisions were answered and through this the proposed solution itself is explained and analysed.

The choice of FPGA was not a seriously considered design problem as the decision has been set out for us in the project requirements and as such the Nexys2 board with the Spartan3 FPGA will be used for the design of the system.

The proposed system solution will be partially reconfigurable. The PC will configure the system for the selected type of hashing function and the appropriate hardware required for that type of hash will be configured onto the FPGA. This would require the functional blocks described later to be

designed for each type of hashing function to be implemented. The reconfigurable portion of the system is shown by a dashed box outline for clarity.

The decision was made to forgo the implementation of a dictionary based attack. The cost of implementing a dictionary attack simply didn't balance the possible benefit. Although performance of recovering passwords that are actually in the dictionary would almost definitely be faster by using this approach it probably wouldn't be faster than a PC based dictionary solution. This comes down to the lower clock speed of the FPGA and much slower memory access. To improve the performance of the dictionary attack while still implementing a 2nd try brute force attempt it would likely require the system to be fully reconfigured for each type of attack. The added complexity of creating a dynamically reconfigurable system would significantly increase the risk of the project not being completed or meeting its objectives.

In addition, by not including a dictionary attack many of the design issues relating to the memory access of the flash/ram can be alleviated, relieving a lot of strain on the design and required hardware to access and control the memory.

Interfacing with the device from the PC will be done using the on-board Cypress USB controller. During testing however to use the system without a fully functional USB interfacing block, software on the pc will be used to load values in and out of the Nexys2.

The design will include several "Functional blocks". A fundamental computational component block will be created. It will include a hashing function block and comparator to check if the hash value corresponds to the given password hash, the comparators will put out a flag if the correct password is found. By designing these as functional blocks they can be independently created for different hashing types that would require different hardware for efficient implementation.

These blocks will get a test-word from the generator circuit and will be required to the hash value through to the comparator. Once a comparator matches a correct hash it will assert its flag line, the control unit will then know which comparator found the correct value and subsequently will know the test word that hashed to the correct value. Multiples of these computational components will be designed to be programmed onto the FPGA and run in parallel to form the basis of the system's computation.

A test-word generator block will be needed and is shown in the diagram; it will run the brute-force algorithm and generate test words for the computation blocks. It will then need to feed these test words to the computational blocks. Ideally this generator circuit should be able to run asynchronously to the computational blocks as generating a test word from the brute-force algorithm will take far less time than the actual computation of the hash. If the generator block is asynchronous to the computational blocks handshaking lines will need to be implemented to ensure the computation blocks are getting correct values. Adding the complexity of handshaking to the system for the purpose of speeding up test word allocation will cause a trade off that would need to be tested in simulation to achieve the best solution.

A simple control unit would need to be put in place to guide the basic operation of the system; it would be placed as part of the generator circuit and will dictate the flow of the system. It will have

functions to guide operations of the system, such as making the USB controller read in and out values from the PC, running the password testing process and handling flags generated by the system, this circuit will also store the words that are being tested by the hashing units and if found to be the correct hash will be read back to the PC through the USB connection.

Making a decision on the number of computational blocks would first be based on fitting the maximum possible amount of parallel processing blocks that the FPGA size will allow. The aim would be to program enough computational blocks onto the FPGA to gain as much speed up as possible compared to the golden standard. However, in order to make an informed decision on this simulations would have to be run on the system to see if the desired effect is achieved, for example additional overhead introduced for instance by the increased size of the generator circuit or increased bus lines required may cause a negative effect on performance and fewer blocks may need to be implemented.

4.4 PC-based gold solution

The PC-based gold solution is simple enough to implement since there is a large number of freely available libraries on the internet to perform hashing. The main.cpp of gold standard can be found in Appendix B. It can be used to measure the time of execution to find the correct password, given the hash and hashing algorithm, and will provide a basis for performance evaluations. There is also an implementation in Appendix A, which is useful to calculate the throughput for different hashes, which may be used to compare to the FPGA solution.

The gold solution is written in C++, and is run on only 1 CPU core.

4.5 Wish list features

The main wish-list function to add would be the implementation of a dictionary based attack and with this to make the system dynamically reconfigurable. By doing this the system can reconfigure itself for a more effective implementation of a dictionary attack and following failure through a dictionary attempt reconfigure itself to a specialized brute-force attempt to ensure a solution is found.

Another wish list function could be to find collisions in the given hashing algorithm, by using collisions in a hashing algorithm the method for finding a suitable password from a hash is significantly faster albeit more complicated than a simple brute-force attempt. These are already present in the MD5 [6], SHA1 [7] and SHA2 [8] functions and being actively researched. A different design which implements collisions could be developed at a later stage or revisit to the project.

5 Performance evaluation

The performance of the FPGA based accelerator and the PC based gold standard need to be compared in order to establish the effectiveness of this solution. These will be done separately and then compared, but first it is important to understand the hashing algorithm. This can then be properly analysed for the PC and then FPGA solutions.

5.4 Explanation of hashing algorithms

It is important that the complexity of the hashing algorithm be inspected, in order that we might see how they perform on a PC, and then how they may lend themselves to being implemented on an FPGA. For this, we will look first look at the MD5 algorithm, and then expand this to other cryptographic algorithms. Only the calculation parts are relevant, and so much has been omitted. For the full explanation of this, see reference [9]. Pseudocode for generic MD5 algorithms is as follows:

```
//Initialize constants
//Append message to make 512 bits
//Process the message in successive 512-bit chunks:
for each 512-bit chunk of message
break chunk into sixteen 32-bit little-endian words w[j], 0 ≤ j ≤ 15
//Initialize hash value for this chunk:
varint a := h0
varint b := h1
varint c := h2
varint d := h3
//Main loop:
fori from 0 to 63
if 0 ≤ i ≤ 15 then
f := (b and c) or ((not b) and d)
g := i
else if 16 ≤ i ≤ 31
f := (d and b) or ((not d) and c)
g := (5×i + 1) mod 16
else if 32 ≤ i ≤ 47
f := b xor c xor d
g := (3×i + 5) mod 16
else if 48 ≤ i ≤ 63
f := c xor (b or (not d))
g := (7×i) mod 16
temp := d
d := c
c := b
b := b + leftrotate((a + f + k[i] + w[g]) , r[i])
a := temp
end for
//Add this chunk's hash to result so far:
h0 := h0 + a
h1 := h1 + b
h2 := h2 + c
h3 := h3 + d
end for
var char digest[16] := h0 append h1 append h2 append h3
```

For further investigation, two more common hashing methods will be investigated: SHA-1 [10]and SHA-2 [11]. SHA2 is the most secure – MD5 and SHA-1 are both susceptible to hacking methods other than brute force [12]– but all are still in widespread use on the internet [13]. The performance for the PC and FPGA solutions will now be investigated.

5.5 Performance analysis of PC based solution

The MD5 algorithm when put on a PC runs the code in a sequential manner, and compiler optimisations make little difference to it. In order to get performance measures for the different hashing algorithms, we stated that we shall look at the following: latency, throughput, and the speedup vs. Gold standard. Of these, throughput is the most intuitive measure, and can be measured in hashes per second, or bits per second to compare better between different hashes. The full calculation can be found in Appendix C. All tests were done on an Intel Core i5-2557M 1.7GHz, running at single core turbo of 2.7GHz, with 4GB DDR3 RAM. The phrase "hello world" was encoded using the applicable hashing algorithms for 5 million hashes. The relevant results are shown below:

Table 1: Performance of PC solution

	Average Runtime	Bits	Hashes/second	Mb/s
MD5	17.13	128.00	291883.31	37.36
SHA1	17.55	160.00	284948.99	45.59
SHA256	6.34	256.00	788328.12	201.81
SHA512	8.36	512.00	597893.02	306.12

The above performance measures give something for the FPGA to live up to.

Another interesting measure to look at is the amount of clocks used to calculate each hash. This is shown below, using the turbo clock of the CPU:

Table 2: Clock cycles of PC solution

	Hashes/second	CPU Clocks per second	Clocks/hash
MD5	291883.31	2700000000	9250.27
SHA1	284948.99	2700000000	9475.38
SHA256	788328.12	2700000000	3424.97
SHA512	597893.02	2700000000	4515.86

This is perhaps a more relevant measure. If we look at the amount of clock cycles used to calculate the hashes, we see that it is a very inefficient process. The aim is to create a massively parallel, reconfigurable FPGA solution.

5.6 Performance analysis of FPGA based solution

The implementation of these algorithms onto an FPGA will now be proposed, along with the projected performance metrics. The simplest way to do this is to use the clock cycles from start to finish for the hashing method; this is done below.

Since this hashing operation takes in 512 bit blocks and words are read in using 32-bit words we require 16 clock cycles to fully read in the hashing block.

By looking at first at the SHA-1 algorithm, we see that most of it can be done in a few clock cycles -

one per iteration - rather than a few thousand as with the CPU code. A single MD5 operation is visualised in the following figure [14]:

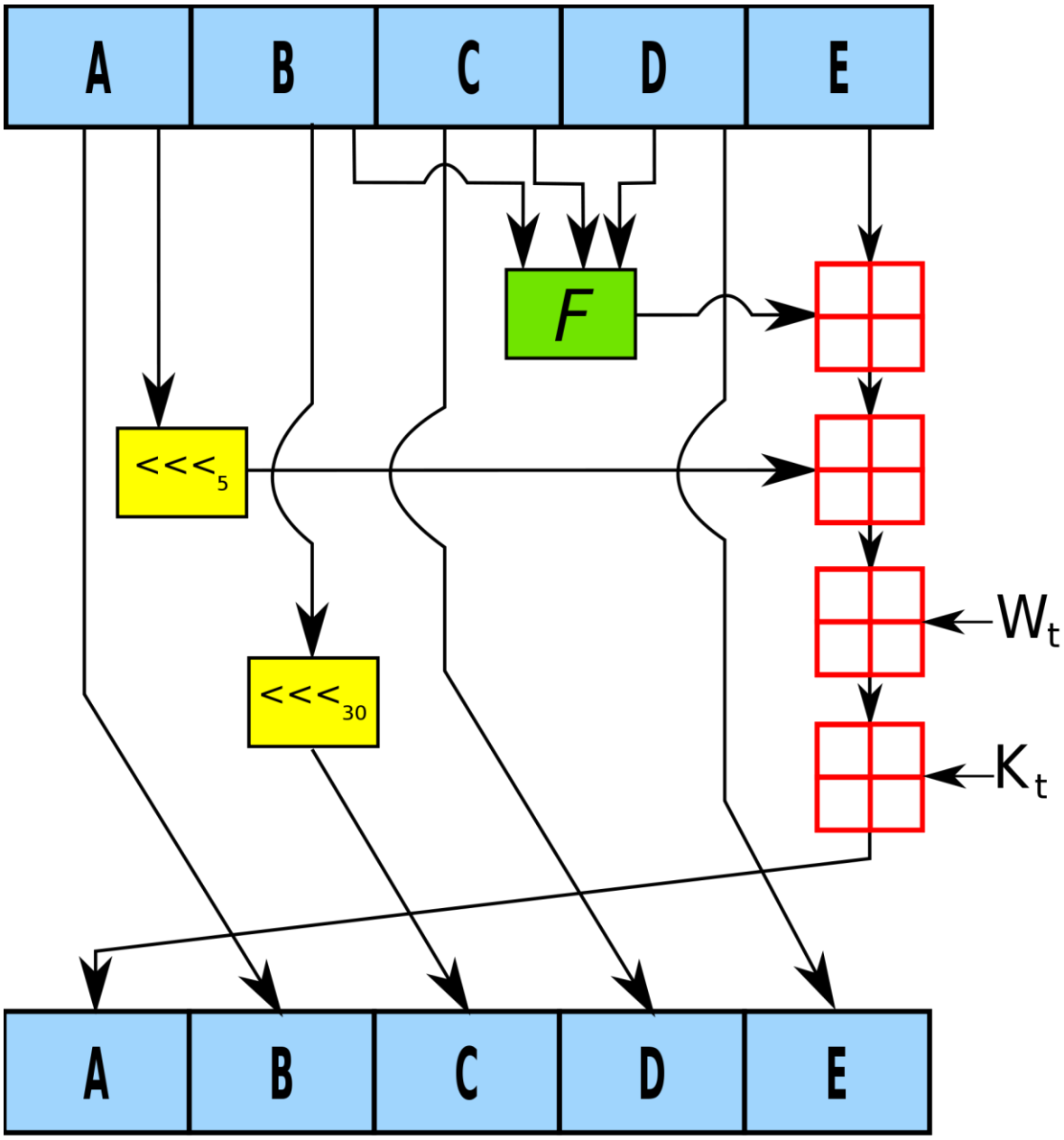

Figure 4: SHA-1 Hashing operation

This above operation must be performed 80 times for the complete hashing operation. The final stage of the hashing operation is to output the 160-bit hash. This is done in 5 clock cycles. The total clock cycles for the above operations comes to 101; however we will see if any optimisations can be done in code to reduce this.

The SHA-1 algorithm was implemented in VHDL code (using an implementation from opencores.org), as well as a comparator in order to make a single reconfigurable block system as described above. The schematic diagram is shown for this below:

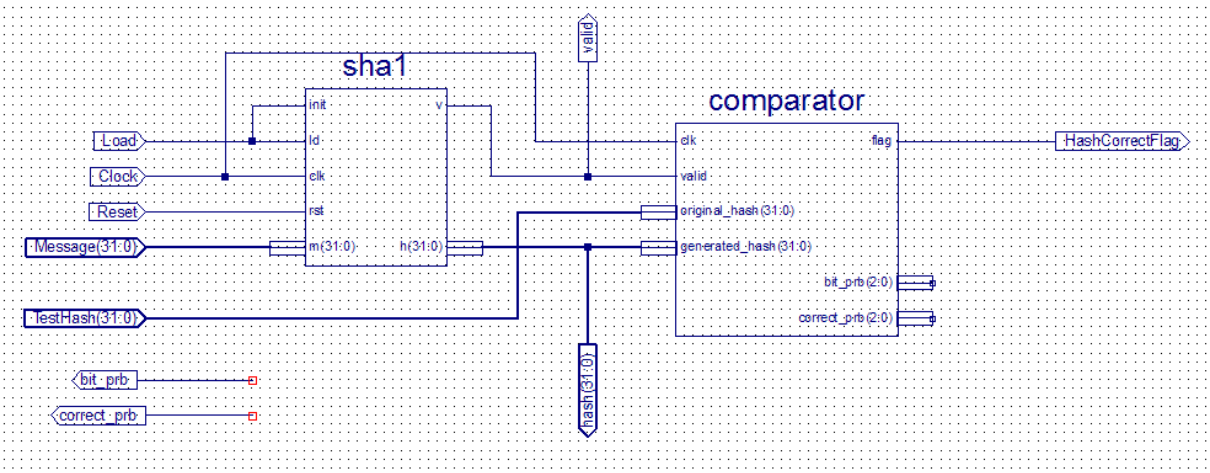

Figure 5: Schematic diagram for one reconfigurable block

Note that this is a debug implementation, and the full solution will not require the intermediate hash, valid, bit_prb and correct_prb lines.

The 16 clock input was parallelised with the 80 clock loop which gave a faster implementation. The following figure simulates the performance of the SHA-1 algorithm using this VHDL code. The input data was 'abcde' in text form, or
0x61626364658000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000028 in the padded hexadecimal message input. The 0x00000028 at the end denotes the length of the code to be hashed (the last 64 bits store the length). This was input over 16 clocks. The golden standard showed the hashed value of this to be 0x03DE6C570BFE24BFC328CCD7CA46B76EADAF4334 in hexadecimal.

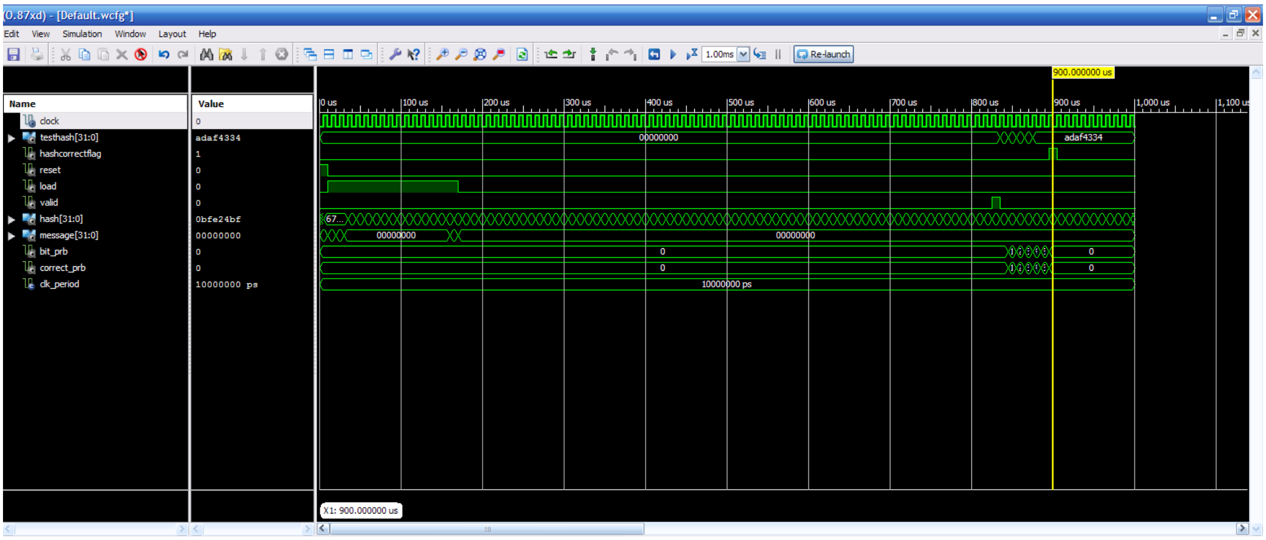

Figure 6: Simulation of SHA-1 algorithm

Counting the number of clocks from the when it is reset to when the full hash is output gives 89 clocks (plus one to compare which was not timed in the golden standard). The output hash is the same as that generated by the PC golden standard (and confirmed by other online SHA-1 generators). The clock period used is 10µs, however the clock period for the SPARTAN-3 FPGA is 20ns (50MHz clock). Using the SPARTAN-3, the total execution time for the hashing function is 1780µs. This gives a total of 561 797 hashes per second, or 89.89Mbits/s.

The SHA-1 core is a large one, and in some places inefficient (e.g. multiple casts need to take place to add signals each time), and as such uses up a large portion of the FPGA slices. Simulated on a SPARTAN-3, the total reconfigurable block uses up 44% of the 1536 flip flops, 57% of the input LUTs, and 74% of the total slices. This is without the control and generator unit, and thus denied us the possibility of placing more than one of these reconfigurable blocks in order to perform calculations in parallel. A larger FPGA or better optimisation would be required for a fully parallel solution.

VADERtest Project Status (05/17/2012 - 10:51:33)			
Project File:	VADER.xise	**Parser Errors:**	No Errors
Module Name:	VADERtest	**Implementation State:**	Placed and Routed
Target Device:	xc3s50-5pq208	**• Errors:**	No Errors
Product Version:	ISE 13.4	**• Warnings:**	4 Warnings (4 new)
Design Goal:	Balanced	**• Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	**• Timing Constraints:**	All Constraints Met
Environment:	System Settings	**• Final Timing Score:**	0 (Timing Report)

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	676	1,536	44%	
Number of 4 input LUTs	890	1,536	57%	
Number of occupied Slices	569	768	74%	
Number of Slices containing only related logic	569	569	100%	
Number of Slices containing unrelated logic	0	569	0%	
Total Number of 4 input LUTs	922	1,536	60%	
Number used as logic	826			
Number used as a route-thru	32			
Number used as Shift registers	64			
Number of bonded IOBs	101	124	81%	
Number of BUFGMUXs	1	8	12%	
Average Fanout of Non-Clock Nets	3.21			

Figure 7: SPARTAN-3 utilisation summary

5.4 Evaluation steps for PC and FPGA solutions

The evaluation steps for comparing the PC and FPGA solutions has been discussed roughly over the past paragraphs, and so it must be properly established. As with the PC based standard, hashes per second remains a very important metric, and shall be used to compare the two solutions; most importantly though, the intended purpose should be tested. This is done using the PC based gold standard in which the user chooses a password to find, and hashing method. The program then iterates through all possible combinations until it finds a correct password, upon which time it exits. A similar scenario will be posed to the FPGA/Nexys2 based solution, where it will receive the password, be configured for a specific type of hashing, and it will run until it finds a password. The CPU program returns its time to find the password, which will be compared to the execution time of the Nexys2 based solution. This will provide a confident measure of effectiveness of the YODA design, and will enable the designers to accurately evaluate the performance of both implementations

Using the above performance metrics, we can compare the PC based golden measure and the VHDL solution (using simulated data). A table best summarises this:

Table 3: Comparison of golden measure and FPGA solutions

SHA-1 Hashing	Golden measure	SPARTAN-3
Clocks/hash	9475.38	89
Clocks/second	2700 000 000	50 000 000
Hashes/second	284 949	561 797
Mbits/second	45.59	89.89

The FPGA based solution is twice as fast as the CPU based solution (using a low-end FPGA). Using a faster FPGA, with more resources and a higher clock speed, and further optimisation of the

algorithm in order to parallelise the hashing could yield a password cracker that vastly exceeds the golden measure.

Works Cited

[1] Digilent. (2007, July) Nexys2 reference Manual. Reference Manual. [Online]. http://www.digilentinc.com/Products/Detail.cfm?NavPath=2%2C400%2C789&Prod=NEXYS2

[2] Simon Winberg. (2012, April) Vula UCT Student Site. [Online]. www.vula.uct.ac.za

[3] Steve Easterbrook. (2004, Janurary) What is Requirements Engineering? dpf Document.

[4] Charles M. Macal. (2005, April) Model Verfication and Validation. Lecture Slides pdf.

[5] Michael R. Marty Mark D. Hill. (2008, July) Amdahl's Law in the Multicore Era. pdf Document.

[6] Vlastimil Klima. (2006, April) Tunnels in Hash Functions: MD5 Collisions Within a Minute. Document Extract.

[7] Stephane Manuel. (2008) Classification and Generation of Disturbance Vectors - International Association for Cryptologic Research. [Online]. http://eprint.iacr.org/2008/469.pdf

[8] Somitra Kumar Sanadhya and Palash Sarkar. (2008) New Collision attacks Against Up To 24-step SHA-2 - International Association for Cryptologic Research. [Online]. http://eprint.iacr.org/2008/270.pdf

[9] R. Rivest. (1992, April) The MD5 Message-Digest Algorithm - The Internet Engineering Task Force (IETF). [Online]. http://tools.ietf.org/html/rfc1321

[10] 3rd D. Eastlake. (2001, September) US Secure Hash Algorithm 1 (SHA1) - The Internet Engineering Task Force (IETF). [Online]. http://tools.ietf.org/html/rfc3174

[11] D. Eastlake 3rd. (2011, May) US Secure Hash Algorithms - The Internet Engineering Task Force (IETF). [Online]. http://tools.ietf.org/html/rfc6234

[12] M. Cochran, T. Highland J. Black. (2009, December) University of Colorado. [Online]. http://www.cs.colorado.edu/~jrblack/papers/md5e-full.pdf

[13] L. Chen S. Turner. (2011, March) Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms - The Internet Engineering Task Force (IETF). [Online]. http://tools.ietf.org/html/rfc6151

[14] Matt Crypto. (2007, August) One MD5 operation. [Online]. http://en.wikipedia.org/wiki/Image:MD5.png

[15] Kris Gaj. (2012, March) ATHENa: Automated Tool for Hardware EvaluatioN. [Online]. http://cryptography.gmu.edu/athena/index.php?id=source_codes

Appendices

Appendix A – Timing of hashes

```
int main (intargc, constchar * argv[])
{
structtimevalstart_time, end_time;
doublerun_time;

hashwrapper *h = newmd5wrapper();
h->test();
std::string md5;
gettimeofday(&start_time, NULL);
for (inti = 0; i<N_RUNS; i++)
md5 = h->getHashFromString("hello world");
gettimeofday(&end_time, NULL);
        std::cout<<"md5: "<< md5 <<std::endl;
run_time = ((double)(end_time.tv_sec) + (double)(end_time.tv_usec)/1000000.0)
    - ((double)(start_time.tv_sec) + (double)(start_time.tv_usec)/1000000.0);
std::cout<<"time taken for md5: "<<run_time<<std::endl;

hashwrapper *s1 = newsha1wrapper();
s1->test();
std::string sha1;
gettimeofday(&start_time, NULL);
for (inti =0; i<N_RUNS;i++)
sha1 = s1->getHashFromString("hello world");
gettimeofday(&end_time, NULL);
std::cout<<"sha1: "<< sha1 <<std::endl;
run_time = ((double)(end_time.tv_sec) + (double)(end_time.tv_usec)/1000000.0)
    - ((double)(start_time.tv_sec) + (double)(start_time.tv_usec)/1000000.0);
std::cout<<"time taken for sha1: "<<run_time<<std::endl;

hashwrapper *s2 = newsha256wrapper();
s2->test();
std::string sha2;
gettimeofday(&start_time, NULL);
for (inti = 0; i<N_RUNS;i++)
sha2 = s2->getHashFromString("hello world");
gettimeofday(&end_time, NULL);
std::cout<<"sha256: "<< sha2 <<std::endl;
run_time = ((double)(end_time.tv_sec) + (double)(end_time.tv_usec)/1000000.0)
    - ((double)(start_time.tv_sec) + (double)(start_time.tv_usec)/1000000.0);
std::cout<<"time taken for sha256: "<<run_time<<std::endl;

hashwrapper *s3 = newsha512wrapper();
s3->test();
std::string sha3;
gettimeofday(&start_time, NULL);
for (inti = 0; i<N_RUNS; i++)
sha3 = s3->getHashFromString("hello world");
gettimeofday(&end_time, NULL);
std::cout<<"sha512: "<< sha3 <<std::endl;
run_time = ((double)(end_time.tv_sec) + (double)(end_time.tv_usec)/1000000.0)
```

```
    - ((double)(start_time.tv_sec) + (double)(start_time.tv_usec)/1000000.0);
std::cout<<"time taken for sha512: "<<run_time<<std::endl;

delete h;
delete s1;
delete s2;
delete s3;
return0;
}
```

Appendix B – Gold standard main.cpp

```
int main (intargc, constchar * argv[])
{
structtimevalstart_time, end_time;
doublerun_time;
stringhashToFind;
stringinitialPass = "!!!a23";
string hash;
bool found = false;

//    hashwrapper *h = new sha512wrapper();
//    hashwrapper *h = new sha256wrapper();
hashwrapper *h = newsha1wrapper();
//    hashwrapper *h = new md5wrapper();
h->test();

char* test = newchar[N_SIZE];
for (inti = 0; i<N_SIZE;i++)
test[i] = '!';

hashToFind = h->getHashFromString(initialPass);
cout<<"Searching for password: "<<initialPass<<" with hash of: "<<hashToFind<<endl;

gettimeofday(&start_time, NULL);
while (!found)
   {

hash = h->getHashFromString(test);
//cout<< "testing: " << test <<" with " << md5 << " against " <<hashToFind<<endl;
if (hash == hashToFind)
      {
found = true;
break;
      }
else
      {
//          if (test[7] == '~')
//          {
//              if (test[6] == '~')
//              {
if (test[5] == '~')
                {
if (test[4] == '~')
                   {
if (test[3] == '~')
                      {
if (test[2] == '~')
                         {
if (test[1] == '~')
                            {
if (test[0] == '~')
                               {
break;
```

```
                        }
else
                        {
test[0]++;
test[1] = '!';
test[2] = '!';
test[3] = '!';
test[4] = '!';
test[5] = '!';
//                          test[6] = '!';
//                          test[7] = '!';
                        }
                    }
else
                    {
test[1]++;
test[2] = '!';
test[3] = '!';
test[4] = '!';
test[5] = '!';
//                      test[6] = '!';
//                      test[7] = '!';
                    }
                }
else
                    {
test[2]++;
test[3] = '!';
test[4] = '!';
test[5] = '!';
//                      test[6] = '!';
//                      test[7] = '!';
                    }
                }
else
                    {
test[3]++;
test[4] = '!';
test[5] = '!';
//                      test[6] = '!';
//                      test[7] = '!';
                    }
                }
else
                {
test[4]++;
test[5] = '!';
//                      test[6] = '!';
//                      test[7] = '!';
                }
            }
else
            {
```

```
test[5]++;
//                      test[6] = '!';
//                      test[7] = '!';
                    }
//                  }
//                  test[6]++;
//                  test[7] = '!';
//              }
//              else
//                  test[7]++;
        }
    }
gettimeofday(&end_time, NULL);
cout<<"Found password as: "<< test <<" with hash of "<< hash <<endl;
run_time = ((double)(end_time.tv_sec) + (double)(end_time.tv_usec)/1000000.0)
    - ((double)(start_time.tv_sec) + (double)(start_time.tv_usec)/1000000.0);
std::cout<<"Time taken to find password: "<<run_time<<std::endl;

}
```

Appendix C – Timing standard test results

	1	2	3	Average Runtime	Bits	Hashes	Hashes/second	bits/second	Mb/s	CPU Speed	Clocks/hash
md5	17.93	**17.51**	**15.94**	17.13	128.00	5000000.00	291883.31	37361063.54	37.36	2700000000	9250.272
sha1	17.28	**17.14**	**18.22**	17.55	160.00	5000000.00	284948.99	45591839.06	45.59	2700000000	9475.38
sha256	**6.45**	**6.45**	**6.13**	6.34	**256.00**	5000000.00	788328.12	201811998.46	201.81	2700000000	3424.9698
sha512	**8.37**	**8.54**	**8.18**	8.36	**512.00**	5000000.00	597893.02	306121228.79	306.12	2700000000	4515.858

Appendix D – VHDL code

See attached