SHA1 decoder
EDA385

Erik Hogeman, ada09eho@student.lu.se
Niklas Hjern, ael09nhj@student.lu.se
Jonas Vistrand, ael09jvi@student.lu.se

October 31, 2013

1 Abstract

This report describes a project implementing a hardware accelerated SHA1 decoder. The main idea is for a user to contol the device with a keyboard and the results and instructions to be output on a VGA display. The project was built and tested on a Nexys3 using a PLB bus and Microblaze processor. Further, a pre-constructed ps2 controller, bram and some non-volatile pcm memory were used.

Custom cores includes a SHA1 hasher, string generator for brute forcing, VGA controller and fsl interface. The end result uses 3 parallel SHA1 hashers and can compute approximately 6.5 million hashes each second. A dictionary prestored in the non-volatile memory was also implemented. The end result is a complete and functional system, very similar to what was intended from the start.

Contents

2 Introduction

The goal of this project has been to build a standalone hardware accelerated system for hash-decoding for the purpose of retrieving the plain text of hashed passwords found online. The system is built to tackle the SHA1 hash algorithm and is capable of performing two different types of attacks, a brute force attack where the system tries to hash every possible character combination up to a certain number of characters to find a match and a dictionary attack where the system searches roughly 70.000 pre-stored words. The hash value to decode is entered into the system by a standard USB keyboard and communication with the end user is done by displaying messages on a VGA display. A full overview of the system can be seen in figure 1. As seen here the system centers around a Microblaze processor connected to a few dedicated hardware cores. The processor handles all inputs from the user and controls what is to be displayed on the VGA display. It also performs the dictionary attack by accessing the PCM non-volatile memory, where all words are stored through the PLB bus. The keyboard is controlled by the processor using a readymade polling-based library and the VGA is controlled by it writing to a certain memory located in the GPU Top that stores the characters currently being displayed. Both of these communications are done over the PLB bus. All signals for driving the VGA display are generated from the custom made VGA core in figure 1 called GPU Top.

The brute force attack is done in a custom made brute force-core that consists of two main parts, an encoder and a string generator. The attack is initialized by the processor sending the hash value entered by the user to the brute force-core over the FSL bus. The string generator then proceeds to generate all possible strings for the encoder to encode until a matching value is found, upon which the result is sent to the processor that displays it on the VGA screen.

The final realization does not differ that much from the original idea apart from a few small details. Originally the dictionary attack was to be done in hardware but this was changed when realizing that it could be implemented quite easily in software leaving more resources on the FPGA board for a faster hash encoder for the brute force attack.

3 SHA1 brute forcer

This section will describe some of the theory behind the SHA1 algorithm, present relevant pseudocode examples and explain the different parts of the hardware implementation.

3.1 The SHA1 algorithm

The hash algorithm chosen for implementation in this project is the so called SHA1 hash function. This function is a collision resistant hash function, which means it is hard to find two inputs with the same output value. It is built using the so called Merkle-Damgård construction. This construction is a way to construct collision resistant hash functions from collision resistant one-way functions, or more specifically if you just use the output from one one-way function as input to another and chain these together, the whole result will be

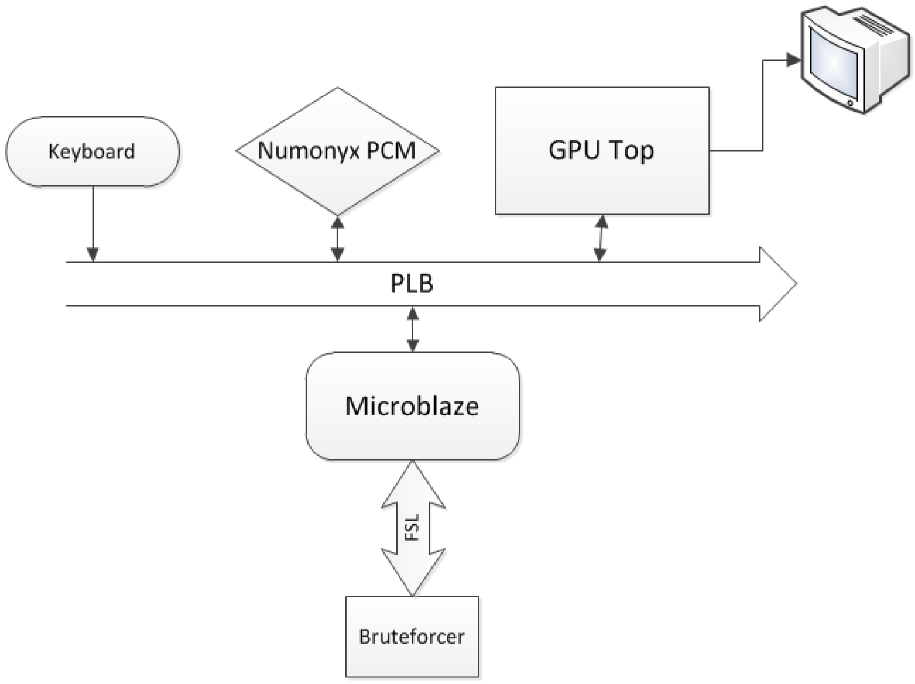

Figure 1: Overall system architecture

collision resistant as long as the one-way function is. The SHA1 algorithm uses this construction, and thus has a loop of size 80 that uses the computed result from the last iteration each time. In order to decode SHA1 hashes, a brute force and a dictionary attack were both implemented.

Pseudo-code for a trimmed version of the algorithm (not supporting strings over 512 bits) can be found below[3]:

```
h0 = 0x67452301
h1 = 0xEFCDAB89
h2 = 0x98BADCFE
h3 = 0x10325476
h4 = 0xC3D2E1F0

//Pre-processing:
append the bit '1' to the message
append 0 ≤ k < 512 bits '0', so that the resulting message length
 (in bits) is congruent to 448 (mod 512)

append length of message (before pre-processing), in bits, as 64-
bit big-endian integer

//Extend the sixteen 32-bit words into eighty 32-bit words:
for i from 16 to 79
        w[i] = (w[i-3] xor w[i-8] xor w[i-14] xor w[i-16])
        leftrotate 1
        //Initialize hash value for this chunk:
        a = h0
        b = h1
        c = h2
        d = h3
        e = h4

        //Main loop:
        for i from 0 to 79
               if 0 ≤ i ≤ 19 then
                      f = (b and c) or ((not b) and d)
                      k = 0x5A827999
               else if 20 ≤ i ≤ 39
                      f = b xor c xor d
                      k = 0x6ED9EBA1
               else if 40 ≤ i ≤ 59
                      f = (b and c) or (b and d) or (c and d)
                      k = 0x8F1BBCDC
               else if 60 ≤ i ≤ 79
                      f = b xor c xor d
                      k = 0xCA62C1D6

               temp = (a leftrotate 5) + f + e + k + w[i]
               e = d
               d = c
               c = b leftrotate 30
               b = a
               a = temp


//Produce the final hash value (big-endian):
digest = hash = a append b append c append d append e
```

3.2 The brute forcer

The two main parts of the brute force related hardware are the string generator and the actual hardware accelerated SHA1-hasher. First, the normal and optimized version of the SHA1-hasher will be explained, then the string generator.

The SHA1 algorithm can be split into two distinct parts; reading and computing w-values, and the main loop actually computing the hash. W-values are named after the w-variable in the pseudocode above, where the initial 16 w-values are parts of the pre-processed input string. The implementation has also been split into two different hardware cores, explained in more detail below.

3.2.1 W-value generator

This core is in an idle state until a start signal is clocked in with value high, which triggers the core to start reading 32-bit values on an in-port. These values represent w[0] to w[15], and one value will be read each clock cycle. After the first value has been read, the core will immediately start clocking out 32-bit output values for the main loop. Waiting until all values have been read would be a huge waste of clock cycles, so the output starts the clock cycle after the input has begun. After clock cycle 16, the core will not read anything more on the input, but will move to another state and start computing and outputting w[16]-w[79]. Since the computations involve references to w:s computed 16 clock cycles earlier, a $16 \cdot 32 = 512$ bit shift register must be used to remember the 16 latest computed w-values. This does waste some hardware, but is unavoidable.

The point is that the main loop shall run in parallel as this core, in order to not waste any clock cycles. After all the values have been output, the core returns to idle state to await another start signal.

3.2.2 Main loop

This core also waits for a high start signal before it moves from the idle state. Once such a signal has been found, the 32-bit w-values are being read one each clock cycle, followed by the necessary computations each round. After a total amount of 80 clock cycles, which is the same as the amount of iterations, the main loop will output the 160 bit hash value. Since the main loop and w-value generator are run in parallel, a complete hash value can be computed in 80 clock cycles.

3.2.3 SHA1-top

The w-value generator and main loop are both connected to each other using a top-module. The 16 first w-values (one value at a time is clocked in) and a start signal are used as input to this module, and 80 cycles later the hash value will appear on the output, together with a signal signaling that there is a new value to be read. The first version can be seen in figure 2.

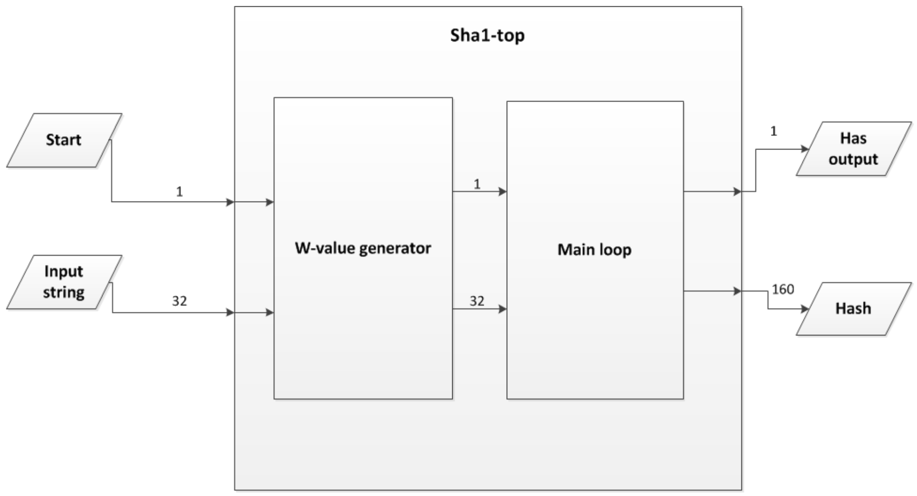

Figure 2: SHA1 hasher first version

An obvious way of optimizing hardware is to increase the clock frequency if the longest combinatorial path allows for it. When we synthesized the SHA1-

hasher, we noticed that our longest combinatorial path had some leeway even when using the maximum clock frequency that we could choose during the Nexys3 project building. Thus we decided to do the opposite in order to speed up our design, in other words make the longest combinatorial path longer. This way more can be computed each clock cycle, reducing the number of cycles required.

To do this, we increased the input to the w value generator to 64 bits, instead reading two w-values at a time each cycle. We also created two instances of the main loop, both connected together in one long combinatorial path, thus basically unrolling the main loop. With this, the clock cycles required went down to 40 instead of 80, making each hasher twice as fast. Our longest combinatorial path was now a little bit longer than the timing constraints seemed to allow in theory, but in practice it has computed the correct result every time so far so the optimization has been kept. An image of the optimized version can be seen in figure 3.

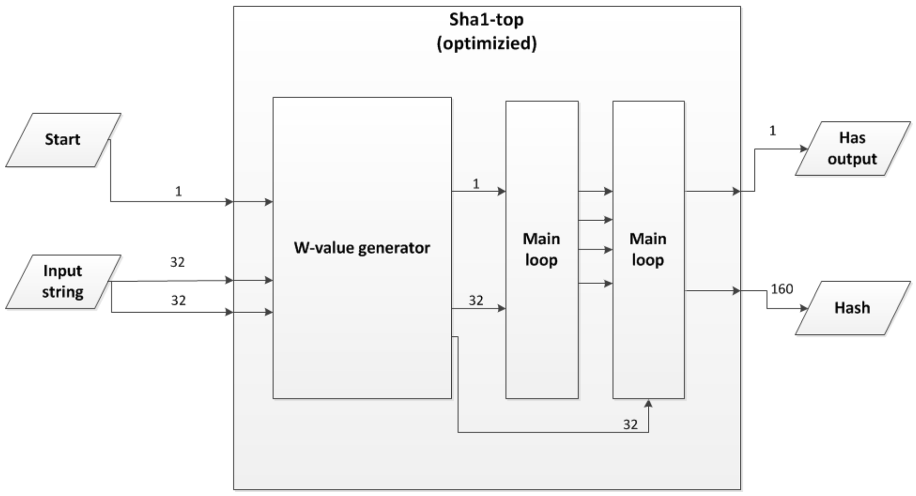

Figure 3: SHA1 hasher second version

3.2.4 String generator

The basic function of the string generator is to generate all possible strings of a specific length combined of all letters (small and capital) and numbers. It achieves this by simulating recursion using a variable keeping track of the current recursion depth, and a shift register with the current letters in the current string (which contains the same number of letters as the current depth). Each clock cycle, a new letter is added, changed or removed depending on the current depth and the letter at the top of the shift register. To make the design easy, the communication with the hasher was also done in this core, and the strings generated was directly generated on the form required for input to the SHA1-hasher.

Thus every time a new string was computed, the core also moves to a new state and start feeding the SHA1-top with 32 bit w-values. Then it waits until the hasher is done, and compares the computed hash value with a reference hash input to it. If the values match, the current string is output. Otherwise, one clock cycle is spent computing the next string, and then the feeding process restarts. An image of this version can be seen in figure 4.

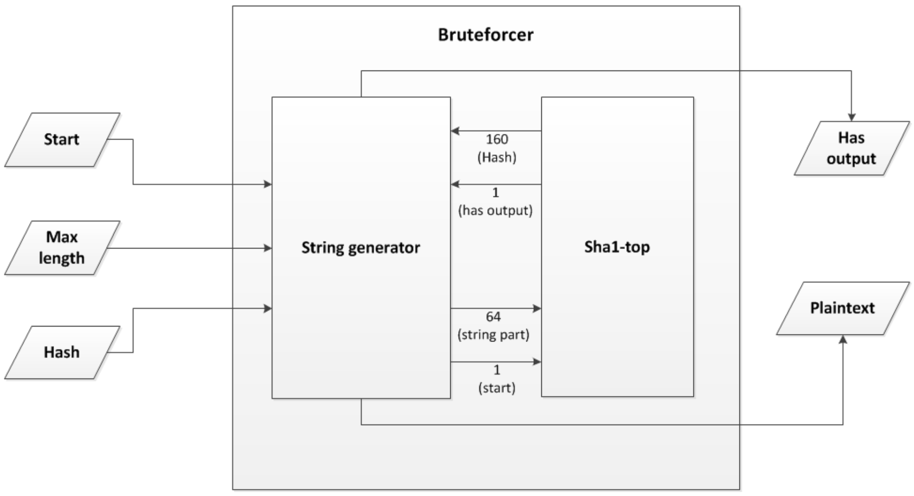

Figure 4: Complete brute forcer first version

Since the string generator spends a lot of time waiting, it could be using that time to compute the next string and start feeding that to another hasher. We implemented this, and thus created several SHA1-tops in a single string generator. This required saving several strings at the same time, which introduced some new hardware in the generator. The hasher was also updated to wait for confirmation after outputting its hash, to make sure the string generator doesn't miss it. The new loop for the string generator now instead became to compute a string, feed it to a hash if there is one not working, and then repeat. If there at some point is no free hasher working, the generator goes to a waiting state, and each clock cycle checks for a new free hasher. When there is one, the feeding process restarts. An image of the optimized version can be seen in figure 5.

We could fit 3 optimized 40-cycle hashes on the Spartan 6-board before the hardware was overused. A VGA-controller and some other things also shared the hardware though, which limited the number of hashes we could use a bit. Since the frequency is 88MHz, we can compute approximately $\frac{3 \cdot 88 \cdot 1000 \cdot 1000}{40} = 6600000$ hashes each second.

4 VGA controller

4.1 VGA theory and basic idea

The VGA screen needs two types of signals to be able to display images. Two sync signals, one horizontal and one vertical, are used to keep track of the current position on the screen. A figure demonstrating the horizontal sync signal can be seen in figure 6. As we can see the screen has so called "porches" used to retrace the sync signals when the end of the screen is reached. During this time nothing is displayed and this is controlled by a signal in the controller

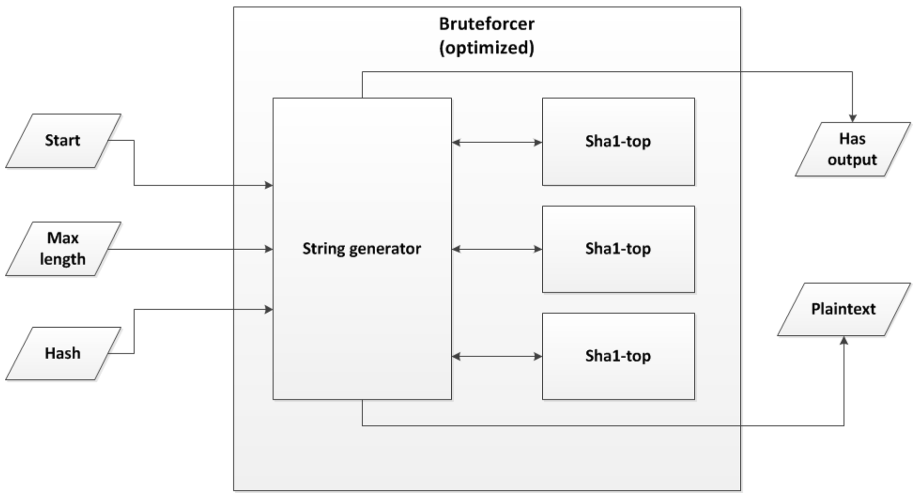

Figure 5: Complete brute forcer second version

called Blank, active high when one of the sync signals are outside of the visual area. Inside the controller there are also two signals keeping track of the current relative position on the screen, h_count and v_count. These are used by logic connected to the controller when displaying images on a certain part of the screen. In this project these signals are connected to the character generator. This component has divided the screen into larger tiles, 8x16 pixels large, and when a new tile is reached by the pixel counters (calculated by dividing the counters by 8 and 16) the generator calls upon this position in a memory where the character currently located in each tile is stored and retrieves the value. This value is then sent to a font memory holding the actual appearance of each character and the pixel value for the current pixel is masked out and displayed on the screen. This is done continuously at 60 frames per second, meaning that the entire screen is redrawn 60 times each second.

The VGA controller in this project is made up of five main parts, see figure 7. Connecting the entire controller and making it accessible from the Microblaze processor is a GPU Top that was generated from Xilinx Core Generator and modified to fit the needed preferences. The communication with the Microblaze processor is done through a PLB bus 32 bits wide. Inside this top is a character memory and a VGA controller and a memory controller.

4.2 Components

4.2.1 VGA controller

The VGA controller is connected to the VGA port on the FPGA board through external ports and continuously generates the horizontal and vertical sync-signals needed by the screen. The controller generates signals at 25 MHz, close enough to the 25.175 MHz really needed. It also generates the RGB signals actually displayed on the screen. The sync-signals are generated independently from the rest of the signals and their implementation comes from the ready-made VGA controller found on the course home page.

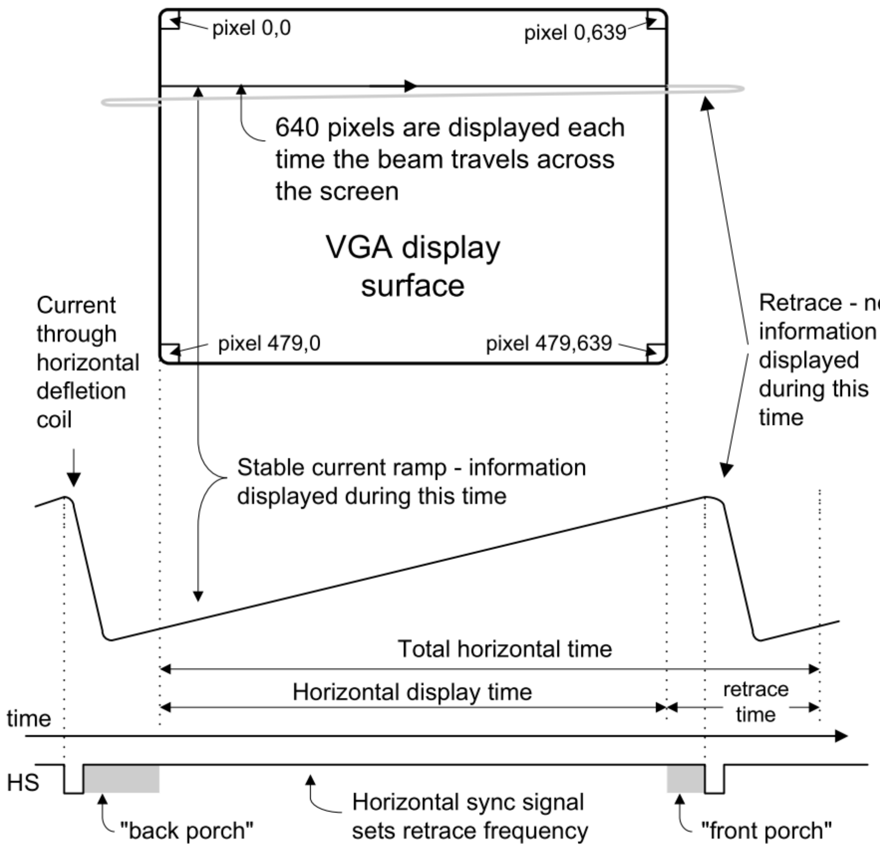
pixel 0,0
pixel 0,639
640 pixels are displayed each time the beam travels across the screen
VGA display surface
Current through horizontal defletion coil
pixel 479,0
pixel 479,639
Retrace - no information displayed during this time
Stable current ramp - information displayed during this time
Total horizontal time
Horizontal display time
retrace time
time
HS
"back porch"
Horizontal sync signal sets retrace frequency
"front porch"

Figure 6: Synchronization diagram for horizontal sync signal[2]

4.2.2 Character memory

The character memory is based on an open source memory[1], and modified to fit the project. It consists of 128x32 8 bit values each representing a character on the screen. The screen is divided into 80x30 tiles, each tile 8x16 pixels large and each holding one character. The 8 bit value in the character memory is the font code for the character currently being displayed in that part of the screen, a font code that is later used to retrieve the actual pixel composition of the character. This character memory is read from the Character generator inside the VGA controller and written to from the Microblaze processor. Software in the processor keeps track of the current screen position and when messages are to be output to the user it writes the correct font code of this character in the correct memory location with a 32 bit message where the first 12 bits are position data, the following 8 bits the font code and the last 12 bits are filled with zeroes.

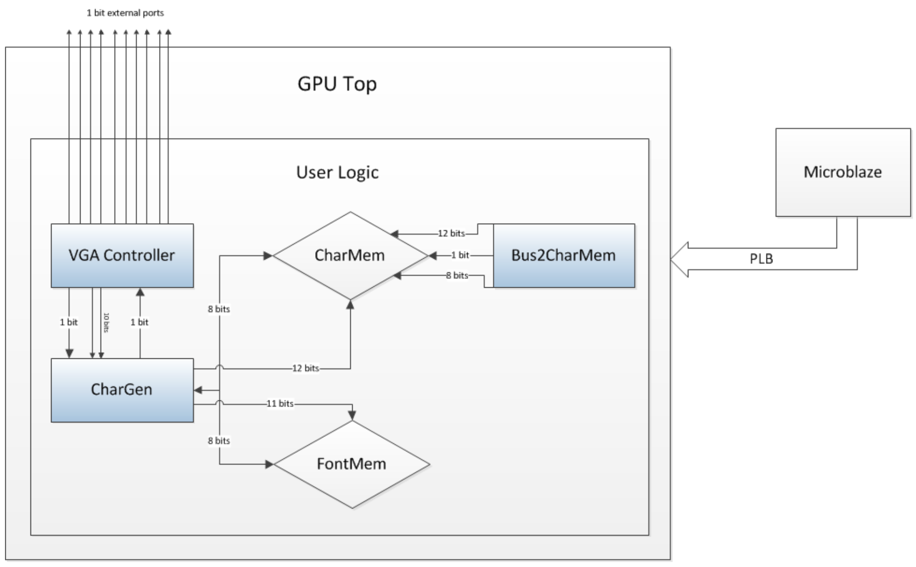

Figure 7: Schematics of VGA controller

4.2.3 Character generator

This component reads an 8 bit value from the character memory and uses this value as a read address from the font memory. The last 4 bits needed by the font memory as an address are the y pixel count based on the current position in the tile, 0 to 15. The received value from the font memory is an 8 bit vector containing the pixel values for the 8 pixels in the current y position of the tile. The correct pixel value is then masked out using a counter and the character generator outputs a single pixel value to the VGA controller. The character generator is controlled from the VGA controller through a pixel clock only active when the sync-signals are in the active positions of the screen. The two memories have a one clock cycle read delay, meaning that the pixel signal output to the screen is offset three pixels to the right.

4.2.4 Bus2CharMem

This is the memory controller that takes signals from the PLB bus and connects them to the character memory. All signals necessary for the PLB bus such as acknowledgements are asserted by this component. It first checks that the write address on the bus is equal to that defined as the address of the character memory and then takes the input signals and writes the correct value in the correct location of the memory.

4.2.5 Font memory

The font memory is taken in its entirety from an open source project[1] and is built to take as input a 12 bit address that consists of a font code that is the base address to the wanted character and a vertical offset that goes from 0 to 15. The memory outputs an 8 bit vector holding the pixel values of the part of the character indicated by the vertical offset.

5 Software

The software on this system is run on simple polling mechanisms. When the system starts the user is greeted by a greeting screen while the system waits for an enter key input from the keyboard, which also is polling based. This is becaue the system will always know when it wants the user to input information to it. The program then lets the user input a hash value and it will only except hash values of correct size (40 hexadecimal number). The user is then presented with the choice to perform a dictionary or a brute force attack. If the brute force attack is selected the program will ask the user to input the maximum size of the password which hash value that is going to be decoded. If one of the attacks is chosen and it fails to retrieve the plain text of the hash the other attack choice will become available. When the correct plain text is found or both of the attacks have been performed on the hash the user can choose to try and decode another hash value or to terminate the program.

5.1 Keyboard Controller

The keyboard communication is done through PLB and is as mentioned above polling based. The input characters are all converted to ascii and saved in a string. This string is then printed to the VGA screen and saved on the heap to allow for processing further down the line.

5.2 VGA controller

The software part of the VGA controller consists mainly of the function vga_print_string(), which is designed as a general function to be used as the single method of writing to the screen, be it a single letter when writing on the keyboard or an entire string as a message to the user. The function reads the string from the input and calls on the function gen_font_code() for each character in the string to generate the correct font code for that character. The function gen_font_code adds the x_count and y_count values to the 8 bit font code of the current character.

5.3 Dictionary attacker

The dictionary consists of the ten thousand most common passwords and an english word list. These words and their corresponding hashes are stored in the FPGA:s non volatile memory. The dictionary attacker simply iterates through the hashes stored in the memory until one of them matches the hash we want to decode. The memory address of the hash is saved so that the program can iterate over the plain text words in the memory until the word with the same address is found, thus retrieving the plain text. All communication with the memory is done with a PLB interface.

5.4 Brute force attacker

After the user has provided the hash value and the maximum size of the password, the hash is converted from a string to hexadecimal values. These hash values is then sent 32 bits at a time with FSL to the brute force IP core. When the entire hash has been sent the program will wait for either the brute forcer to return the plain text or until the brute forcer has gone through all possible password combinations and returns the plain text zero.

5.5 Memory Requirements

All the software on this system needs to be stored on a minimum of 32 kb block ram.

6 User manual

6.1 User interface

The user interface includes the VGA screen and the keyboard. The VGA screen will at all times provide the user with instructions. All the user input is done with a PS2 keyboard.

6.2 Providing the hash

The program will ask the user to input the hash value on which the attack is to be performed on. This hash value need to be of type SHA1 and it is always 40 hexadecimal digits long. The program will tell the user if the hash value is not of the correct size but not if it is the wrong format or if it's containing non hexadecimal characters. Providing an incorrect hash will not lead to an error in the execution, the only result will be that no plain text is found in either the brute force or dictionary attack. If the hash is of the wrong size however the user will simply be asked to input the hash again. An important side note is that the decoder does not provide support for passwords containing special characters, only lower- and uppercase letters of the english alphabet and numbers are supported.

6.3 Selecting the attack

After inputing the hash value, the user will be presented with the choice of performing a brute force or a dictionary attack. No matter what choice is made, if the attack comes up empty handed the user will get the chance to perform the other attack on the same hash value. If the user provides a non valid input the program will simply let the user input another value until the input is in the correct format.

6.4 Brute force attack

When the brute force attack is selected the program will ask the user to specify the maximum length of the password that we have obtained the hash value for. The program will allow a size up to 7 characters but a size larger than 6 is not feasible since the number of possible plain text combinations are too large for the program to have time to go through them all in a reasonable amount of time. In fact a password of 6 characters will in a worst case scenario take 2 h and 20 min to decode. If the attack is successful the plain text will be printed on the screen but if it is unsuccessful the user will be asked if the system should perform a dictionary attack on the hash instead. The user can also choose to terminate the program or to get the opportunity to input another hash value to try and decode.

6.5 Dictionary attack

When the dictionary attack is chosen it will start going through the pre-stored database of words immediately. If the password under attack is one of the ten thousand most common passwords or a lowercase word in the english dictionary, the hash will be decoded. If the attack is successful the user will get the chance to restart the program and if it's unsuccessful a brute force attack can be performed on the same hash value instead.

6.6 Uploading word list

The FPGA comes pre-loaded with the ten thousand most common passwords and an english word list with sixty thousand words and their respective hash values. If the user wishes to upload their own word list this is done by uploading a ANSI encoded .txt file containing the words seperated by a new line character to the memory address 0x87000000 (start address of the non volatile memories). The corresponding hash values are uploaded in the same fashion but to the memory address 0x87100000 instead. The uploading is done with Adept under the memory tab. Choose the bpi interface and upload the files starting with the offset 0 (plain text) and 100000 (hash values).

7 Problems

7.1 Hardware

The first problem encountered was how to make custom IP cores communicate with the microblaze processor and what bus system to use. After discovering that there already was an existing PS2 keyboard controller with a PLB interface it was decided that the system would use PLB.

7.1.1 Brute forcer

The main problem with implementing the brute forcer was to keep hardware usage to a minimum. Some problems were very hard to solve without using quite a bit of new hardware. In the start, only one hasher could be used before the hardware overflowed. This was because a gigantic shift register was used in the string generator to support very long strings. We decided that only 7 characters were feasible in reality using our design, and thus trimmed this shift register to only support those 7. This decreased the hardware enough to allow for 2, almost 3 hashes.

A big cluster of if and case statements were also initially used to append the bit 1 required for string preprocessing before sending it to the hasher, which also wasted a lot of hardware. This was finally solved by simply putting this bit at the top of the shift register in the beginning instead, which automatically keeps it at the correct position. This decreased the hardware usage enough to allow for a third hash. To support a fourth hash would require more hardware or a completely new, smarter design.

7.1.2 VGA

A big problem faced was when trying to read from the character memory from the Microblaze processor. Writing to the memory was never a problem but when trying to read values from the memory to be able to create a scrolling effect with the text on the screen some very strange and undefined behaviour occurred. The hardware was modified to also contain a component similar to the already present Bus2CharMem to function as a read-controller for the bus. However, due to reasons unknown this never worked as planned and as the idea was mostly a fun feature from the start this was dropped in favour of working to optimize the hash-cores.

7.1.3 Area constraints

At first not much attention was payed to the area constraints of the FPGA board which lead to a non optimized SHA1 hasher that took a lot of the available LUT slices. A lot of optimizing was done to the first implementation of the algorithm to allow for more to be placed on the FPGA board.

7.2 Software

One problem was how to find out which commands in C that would communicate with the IP cores in the hardware architecture and also what files to include to enable these commands. This was a recurring problem but after a time of searching the authors were able to figure it out. Approximately half way into the projects when when the system first started to come together the excessive usage of strings in the code lead to memory leaks and undefined behaviour of the system. This was fixed by thoroughly going through the code and freeing all of the allocated memory.

7.2.1 Keyboard controller

The premade keyboard controller caused a lot of problems in software. Mainly because it sent a lot of make and break codes instead of just one of each. The solution contained some workarounds in C which lead to problems later in the project when sometimes the keyboard didn't read the first key pressed after the microblazed had performed an attack on a hash value. This was later solved with another workaround.

7.2.2 Memory requirements

Not much thought was originally put on how much BRAM the system would need to store and execute the C code. The size was therefore needed to be increased two times.

8 Lessons learned

The biggest leason learned during this project is to not write hardware implementations of things that already exists as IP cores in Xilinx. Not doing this lead to two of the authors dedicating the first week to writing a keyboard controller only to discover that one already existed. This knowledge came in handy when the authors were about to write a memory controller to the non volatile memories on the FPGA board but checked that there already existed a functioning ip core with this functionality. Another important discovery was that the area constraints were much more limiting than imagined, thus not allowing for as many SHA1 hashers as wanted.

9 Contributions

- Niklas Hjern: Wrote the hardware and software controlling the VGA screen.
- Erik Hogeman: Implemented the SHA1 hasher and brute force attacker in hardware. Optimized the SHA1 algorithm to be as fast and area efficient as possible.
- Jonas Vistrand: Designed an FSL interface to the brute force attacker and implemented the dictionary attacker. Wrote software for user interface.
- Testing and assembling of the system was done by all the authors.

References

[1] http://ece320web.groups.et.byu.net/labs/VGATextGeneration/VGA_Terminal.html Retrieved October 16 2013.

[2] VGA controller reference design from http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,789&Prod=NEXYS2 Retrieved October 16 2013.

[3] http://en.wikipedia.org/wiki/SHA-1 Retrieved October 16 2013.