Department of Computer Science
Lund University, Faculty of Engineering

EDA385
Design of Embedded Systems – Advanced Course

Final Report

A Parallelized Hash Generator System

Authors:
Niklas Aldén
ael10nal@student.lu.se
Gabriel Jönsson
ael10gjo@student.lu.se
Jonathan Sönnerup
ael10jso@student.lu.se

Supervisor:
Flavius Gruian

October, 2014

Abstract

This report covers the implementation of a bruteforce password cracker, using the MD5 hash algorithm, implemented on a Nexys 3 FPGA board. The system is built with parallelization in mind to maximize performance depending on system constraints. Using generics the user can easily increase the number of calculating cores, thus increasing the throughput. Our goal was to have a proportional gain in cracking speed, by doubling the number of cores the time to find a password should decrease by a factor of two. This was achieved, at least in simulations.

Contents

Niklas Aldén
Gabriel Jönnson
Jonathan Sönnerup

1 Introduction

This is the final report of the project in the Design of Embedded Systems Advanced Course (EDA385). The purpose of this project was to create a embedded system, a combination of hardware and software, and to run this system on an FPGA. The system which this report covers is a parallelized hash generator system used for cracking MD5-hashed passwords. The original idea was to create a rainbow table based password cracker, but since this would need a lot of overhead we chose to focus only on the MD5 cracker and to make it fast and parallel, see figure 1.

Hashes are used everywhere today when it comes to handling passwords. A hash is simply the original password, which can be of variable length and has gone through an arithmetic process to get a new message of fixed size. The important thing about these hashes is that they are non invertible which means that if you have the hash and the algorithm you can not reverse engineer the process to get the password. There are multiple processes to create a hash but the one this report covers is the MD5 algorithm.

1.1 Concept

To use the system an existing hash is needed. This hash can be acquired from for example existing databases or web pages which converts messages to MD5 hashes. When this hash is put on the UART to the processor the software puts the data on the FSL bus to the custom IP-core, then the cracking begins. After some time, depending on the length of the password and the specified character set, the password is returned in clear text to the user.

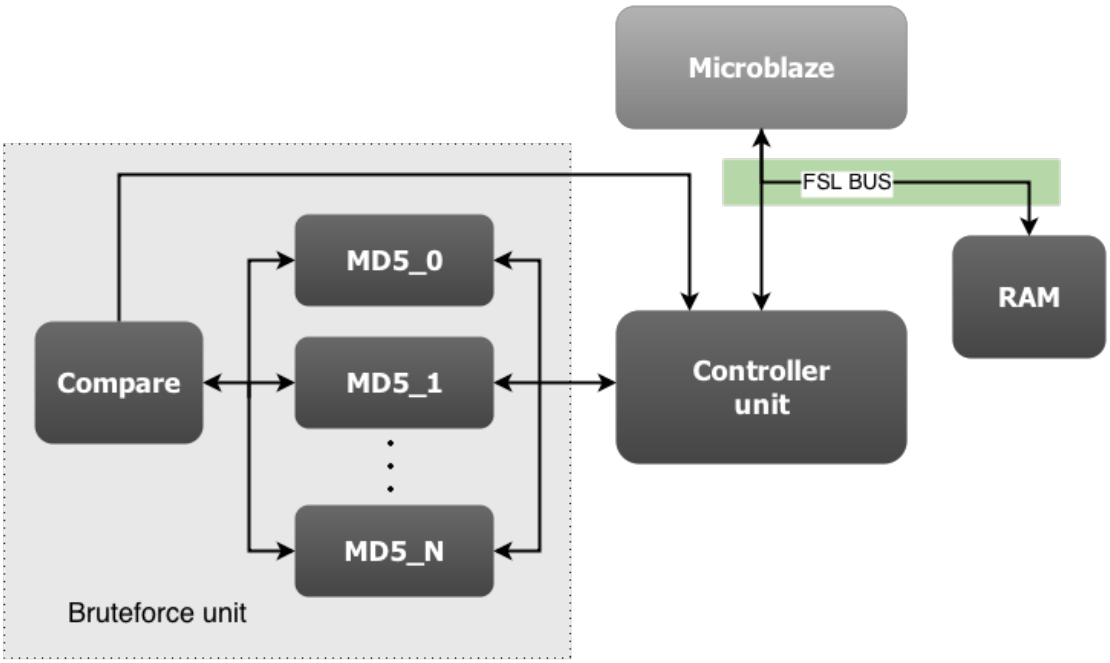

Figure 1: Architecture overview.

Niklas Aldén
Gabriel Jönnson
Jonathan Sönnerup

2 Hardware

The key thing in a bruteforcer is speed, which means a lot of hardware accelerated calculation is needed to find passwords in a reasonable amount of time. The critical path in the design is a multiplication in the MD5 block which limits the clock frequency to 50 MHz instead of the native 100 MHz the Microblaze usually runs at. Each MD5 core also needs 65 clock cycles for its calculations, so with one MD5 core we are able to calculate 770'000 hashes every second and with four MD5 cores almost 3.1 million hashes per second, see equation 1.

$$\frac{\text{clock frequency}}{\text{MD5 delay}} \cdot \#\text{cores} \Rightarrow \frac{50\text{ MHz}}{65} \cdot 1 \approx 770000\text{ s}^{-1}\ ,\ \frac{50\text{ MHz}}{65} \cdot 4 \approx 3077000\text{ s}^{-1} \quad (1)$$

As seen in table 1 we only occupy 53% of the FPGAs slices with four MD5 cores, so there is still room for more cores. The number of cores is only limited by the size of the FPGA. The difference in speed between one and four MD5 cores is shown in table 2 and 3 where the maximum time for cracking a hash based on the password length and character set used.

Table 1: Hardware utilization for the bruteforcer with *four* cores.

Item		
Devices	**Used**	**Utilization**
# of slice registers	2453	13%
# of LUTs	3673	40%
# of Occupied Slices	1229	53%
# of RAMB16	16	50%

Table 2: Number of characters in the password and the corresponding maximum time needed to find a password using *one* MD5 core. The clock frequency is 50 MHz and the char set consists of all lower case letters, {a–z}, or all numbers, upper- and lower case letters, {0–9, A–Z, a–z}.

	Maximum time for char set	
password length	**{a–z}**	**{0–9, A–Z, a–z}**
1	35.1 μs	80.6 μs
2	914 μs	4.92 ms
3	23.8 ms	300 ms
4	618 ms	18.3 s
5	16.1 s	18 min 36 s
6	6 min 58 s	18 h 55 min
7	3 h 1 min	48 days 2 h

Niklas Aldén
Gabriel Jönnson
Jonathan Sönnerup

Table 3: Number of characters in the password and the corresponding maximum time needed to find a password using *four* MD5 cores. The clock frequency is 50 MHz and the char set consists of all lower case letters, {a–z}, or all numbers, upper- and lower case letters, {0–9, A–Z, a–z}.

	Maximum time for char set	
password length	**{a–z}**	**{0–9, A–Z, a–z}**
1	9.18 μs	21.1 μs
2	239 μs	1.29 ms
3	6.21 ms	78.5 ms
4	162 ms	4.79 s
5	4.20 s	4 min 52 s
6	1 min 49 s	4 h 57 min
7	47 min 20 s	12 days 14 h

2.1 FSL Controller

This block receives the hash from the user via the *FSL bus* and sends it to the controller unit. When a password is found it is sent back to the user.

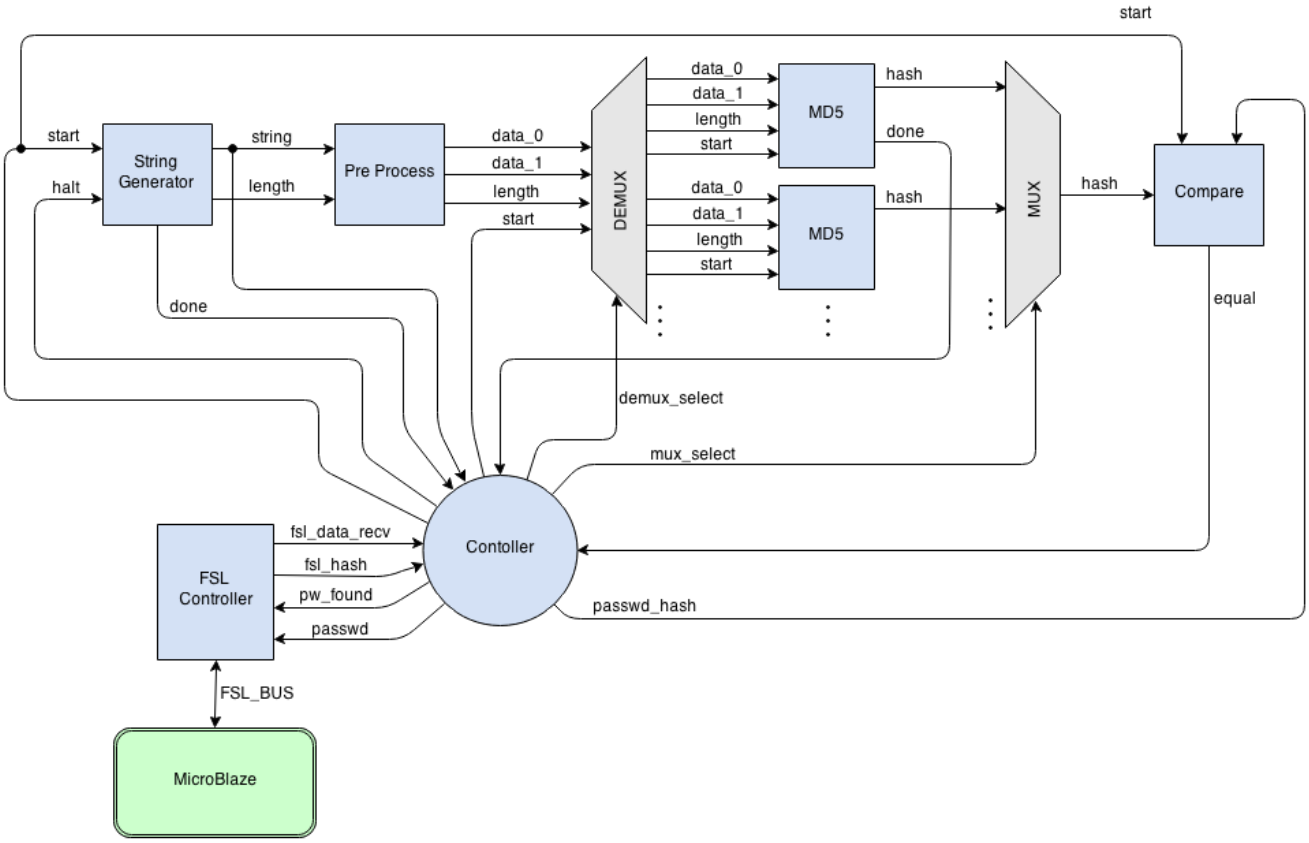

Figure 2: The architectural implementation of the IP-core together with the Microblaze and the FSL bus.

2.2 String generator

Starts with an empty string with only *NULL* characters. It can output any character of the ASCII table, the character set is defined by a range stored in two registers.

If the set is defined to only support lower case letters (a–z), the string generator will start by outputting an “a”, then a “b” up to a “z”. After that it will append one more letter and do the same loop again, e.g. “aa”, “ab”, . . . , “az”, “ba”, . . . , “zz”, “aaa” and so on.

The *length* of the string is also forwarded to the *Pre processing* unit because the string length is appended before the calculations in the MD5 cores starts. To halt the generation, when the MD5s are working, there is a *halt* signal used by the controller. A *done* signal is used to indicate that the generator has completed, thus every potential password has been generated. An example of the string generator in action is shown in figure 3.

Niklas Aldén
Gabriel Jönnson
Jonathan Sönnerup

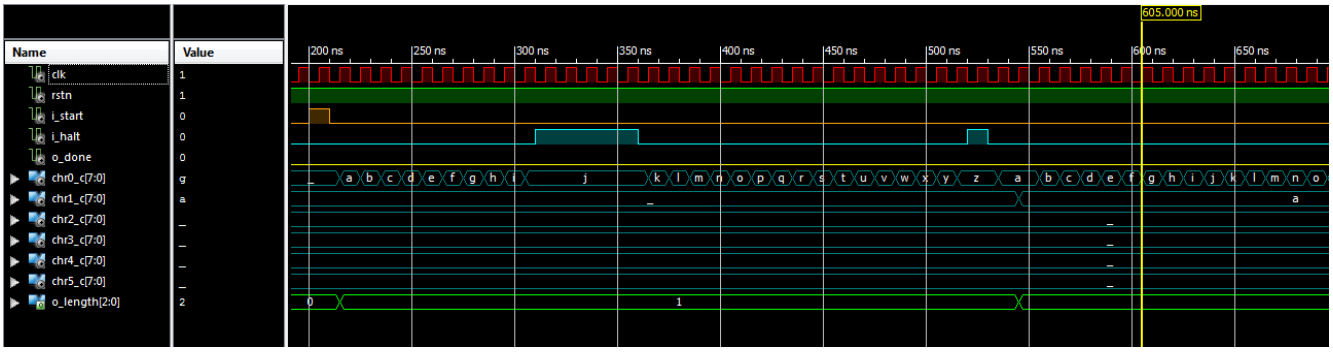

Figure 3: The string generator in progress, generating potential passwords. The functionality of the start- and halt signal is also shown.

2.3 Pre Process

In the pre process, see listing 1, the string from the string generator is arranged into 32-bit words in little endian. After the last character a single bit, set to "1", is appended and the string length is padded with zeros to form a 32-bit word. Since pre processing is needed before every run of the MD5 cores, and we want to have a lot in parallel, we decided to move the pre processing step outside of the MD5 cores to minimize the area. It is only a combinatorial block and doesn't introduce any noticeable delay to the system.

```
//Pre-processing: adding a single 1 bit
append "1" bit to message
// Notice: the input bytes are considered as bits strings, where the
    first bit is the most significant bit of the byte.
//Pre-processing: padding with zeros
append "0" bit until message length in bits = 448 (mod 512)
append original length in bits mod (2 pow 64) to message
```

Listing 1: Pseudocode for the pre processing of the MD5 algorithm [1].

2.4 MD5

The message that the MD5 core starts with can have any length, but it calculates with 512 bits of the message at a time. If the message is shorter, "0"-bits are appended until the 512 bit length is achieved. Then each 512-bit block will be calculated in a 64 cycle loop. When all 512-bit blocks has been looped through the final 128-bit hash is outputted and the *done*-signal is set high for one clock cycle. The MD5 algorithm is already pipelined but it would be possible to improve the pipelining further to achieve a better performance.

The pseudo code for the MD5 algorithm is shown in listing 2.

Niklas Aldén
Gabriel Jönnson
Jonathan Sönnerup

```
var int[64] s, K
//s specifies the per-round shift amounts
s[ 0..15] := {7,12,17,22,7,12,17,22,7,12,17,22,7,12,17,22}
s[16..31] := {5,9,14,20,5,9,14,20,5,9,14,20,5,9,14,20}
s[32..47] := {4,11,16,23,4,11,16,23,4,11,16,23,4,11,16,23}
s[48..63] := {6,10,15,21,6,10,15,21,6,10,15,21,6,10,15,21}
//Use binary integer part of the sines of integers (Radians) as constants
for i from 0 to 63
    K[i] := floor(abs(sin(i + 1)) * (2 pow 32))
end for
//Initialize variables:
var int a0 := 0x67452301   //A
var int b0 := 0xefcdab89   //B
var int c0 := 0x98badcfe   //C
var int d0 := 0x10325476   //D

//Process the message in successive 512-bit chunks:
for each 512-bit chunk of message
    break chunk into sixteen 32-bit words M[j], 0 <= j <= 15
//Initialize hash value for this chunk:
    var int A := a0
    var int B := b0
    var int C := c0
    var int D := d0
//Main loop:
    for i 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
        dTemp := D
        D := C
        C := B
        B := B + leftrotate((A + F + K[i] + M[g]), s[i])
        A := dTemp
    end for
//Add this chunk's hash to result so far:
    a0 := a0 + A
    b0 := b0 + B
    c0 := c0 + C
    d0 := d0 + D
end for
var char digest[16] := a0 append b0 append c0 append d0
```

Listing 2: Pseudo code for the MD5 algorithm [1].

Niklas Aldén
Gabriel Jönnson
Jonathan Sönnerup

2.5 Compare unit

The hashes calculated by the MD5 cores is compared to the hash entered by the user. This is a simple block that latches the user's hash into a register when the *start*-signal is set, then alerts the controller by setting the *equal*-signal high for one clock cycle when it finds a match with one of the MD5's hashes.

2.6 Controller

The main hardware controller handles the timing of everything. When the *start*-signal is set the entered hash is latched into a register in the *compare* block, and the *string generator* begins to generate strings.

The controller has a generic amount of registers for storing passwords created by the string generator. This amount of registers is the same as the number of parallel *MD5 cores*, since only the passwords that are currently being hashed needs to be stored temporarily. In case one of the current hashes is a match we need to send the corresponding password to the user. This also alert the FSL controller that a password has been found by setting *pw_found* high.

One select signal goes to the *demultiplexer* and one to the *multiplexer* to control which potential password (from the string generator) is being inputted and which hash is being outputted from each MD5 core.

When each MD5 core has begun calculating, the string generator is halted with a *halt*-signal so no passwords gets skipped. When the last MD5 has outputted its hash, the halt-signal is dropped and the string generator immediately continues.

The controller only needs to listen to the *done*-signal from the first MD5 core since the rest of the cores will finish one clock cycle after the previous one. So each clock cycle after the first MD5 core is done, a hash will be multiplexed into the compare unit.

3 Software

The software reads a hash inputted by the user from UART, a 128-bit string, splits this into four 32-bit chunks and sends this to the FSL controller. It then receives the password, if found, and displays it to the user in the terminal.

Since the VGA controller was not fully functional the software responsible for the VGA handling had to be excluded.

The memory required was the standard 32 kB RAM since the major part of the software was discarded.

4 Problems

Our final product was not as the initial proposal. We spent the first week discussing how to implement a rainbow table, just to realize that the overhead would take to much time to implement. This week could have been used to work on the project instead of deciding what and how to do it. Lesson learned, come prepared!

The multi core system worked in simulations but we did not see the performance increasing when running it on the FPGA. Four MD5 cores performed the same as only one, if this is due to some optimizations we missed or other constrains we didn't know about remains unknown.

Thanks to our experience in hardware development we did not experience any major setbacks while making the bruteforce IP-core. However the VGA controller proved to be a hassle. Even after several different implementations we were not successful in displaying the desired text. Since other projects have had a working VGA controller we can only blame ourself for not asking for help in time.

5 Contributions

Most of the development where done in cooperation with each other. Some blocks in the bruteforcer was mainly written by one person and is listed below:

Niklas Aldén wrote the pre processor, MUX/DEMUX, part of MD5 core and drew the schematics.

Gabriel Jönsson wrote the comparator, VGA controllers and part of the MD5 core.

Jonathan Sönnerup wrote the string generator, main controllers (FSL and bruteforce) and the software.

The person writing a block also developed the test bench(es) for it. This may not be the best practice since one can overlook flaws and bugs.

A User manual

- Open the project `Brutus_multi/Xilinx/system.xmp` in XPS, synthesize it and export it to SDK. Choose the workspace `Brutus_multi/software`, build the software and upload it using *Adept*.
- Connect to the system through the serial port (`COMXX`). Find a hash you want to crack, enter it in the terminal when asked.
- Wait for the system to crack the password.
- The result will printed in the terminal when found.

References

[1] Wikipedia. MD5. `http://en.wikipedia.org/wiki/MD5`, 11:35, September 26 2014.

Niklas Aldén
Gabriel Jönnson
Jonathan Sönnerup