The Design of the SHA1 Co-Processor

ECE 111 Final Project

Winter Quarter 2016

Date: March 14, 2016

Name: Alfredo Sauceda

A10482838

Introduction

The SHA-1 algorithm is a hash function used to produce a 160-bit value unique to the input message. It was creating by the United States government in 1993. It has two main uses in cryptography and data integrity. It has been suggested to deprecate the SHA-1 algorithm since around 2010 because of the ease of finding a hash collision on an insubstantial amount of money and computing time. However, SHA-1 still retains its uses in data integrity and revision control systems such as Git and Mercurial.

Description of the SHA-1 Algorithm

Psuedo-code (from the Wikipedia article)

```
Note 1: All variables are unsigned 32-bit quantities and wrap modulo 232 when calculating,
except for
        ml, the message length, which is a 64-bit quantity, and
        hh, the message digest, which is a 160-bit quantity.
Note 2: All constants in this pseudo code are in big endian.
        Within each word, the most significant byte is stored in the leftmost byte position

Initialize variables:

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

ml = message length in bits (always a multiple of the number of bits in a character).

Pre-processing:
append the bit '1' to the message e.g. by adding 0x80 if message length is a multiple of 8
bits.
append 0 ≤ k < 512 bits '0', such that the resulting message length in bits
   is congruent to −64 ≡ 448 (mod 512)
append ml, in a 64-bit big-endian integer. Thus, the total length is a multiple of 512 bits.

Process the message in successive 512-bit chunks:
break message into 512-bit chunks
for each chunk
    break chunk into sixteen 32-bit big-endian words w[i], 0 ≤ i ≤ 15

    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

    Add this chunk's hash to result so far:
    h0 = h0 + a
    h1 = h1 + b
    h2 = h2 + c
    h3 = h3 + d
    h4 = h4 + e

Produce the final hash value (big-endian) as a 160 bit number:
hh = (h0 leftshift 128) or (h1 leftshift 96) or (h2 leftshift 64) or (h3 leftshift 32) or h4
```

Logically, the pseudo code is simple to understand and break apart into English.

First, we initialize our starting hash values h0 - h4. These are a given at the start of every run of the SHA-1 algorithm. After the initial hash values are initialized, we take the length of the message in bits and then we begin pre-processing.

Pre-processing

- Append bit ‘1’ to the end of the message
- Append ‘k’ number of bits as ‘0’ after the ‘1’ to make the message length 64 bits less than some multiple of 512.
- Finally, append the message size (in bits) to the end of the message.

Once the pre-processing is done, we go into the algorithm itself

Algorithm

- Break the message into 512 bit chunks

- For every chunk:
    - Break the chunk into 16 32-bit words w[i] in big endian where 0 <= i <= 15
    - Now we extend the 16 32-bit words into 64 more words from 16<= 79 where
        - W[i] = (w[i-3] ^ w[i-8] ^ w[i-14] ^ w[i-16]) <<< 1 (where <<< denotes leftrotate)
    - Create the SHA-1 round variables A through E and initialize them to h0 through h4 respectively (A = h0, B = h1, etc)
    - Finally, for the main loop, the value of i determines the value of both the hash function ‘f’ and some constant ‘k’
    - For i from 0 to 79
        - if 0 <= i <= 19 then
            - f = (B & C) ^ (~B & D)
            - k = 0x5A827999
        - if 20 <= i <= 39 then
            - f = (B ^ C ^ D)
            - k = 0x6ED9EBA1
        - if 40 <= i <= 59 then
            - f = (B & C) ^ (B & D) ^ (C & D)
            - k = 0x8F1BBCDC
        - if 60 <= i <= 80 then
            - f = (B ^ C ^ D)
            - k = 0xCA62C1D6
        - After the value of ‘f’ and ‘k’ have been determined by ‘i’, then we update our A-E values
        - temp = (A <<< 5) + f + E + k + w[i]
        - E = D
        - D = C
        - C = B <<< 30
        - B = A
        - A = temp
    - Then finally, after 80 rounds of this, we find new intermediary values for our hash variables h0-h4.
    - h0 = h0 + A
    - h1 = h1 + B
    - h2 = h2 + C
    - h3 = h3 + D
    - h4 = h4 + E
    - After this we move on to the next message chunk and process everything for that chunk and we repeat until we’ve processed the final chunk.

- Finally to find the final hash value HH, we concatenate the bits of h0 - h4 as follows
- HH = {h0, h1, h2, h3}

And that is in English the SHA-1 algorithm.

Design Details

As opposed to using a state machine as I did in the RLE project, I decided to implement the design using multiple clocked always statements. I decided on this in the hopes of having a greater increase of F_{Max} than from using a state machine. There were two big struggles in my design, one was getting the timing of all the registers to line up correctly and the second is trying to deal with the massive amount of area taken up by the design is currently still the biggest problem with my design. My design is definitely targeted towards delay.

Summary of Results

- #ALUTs = 4,086
- #Registers = 2,976
- Area = 7,062
- Clock Period = $1/F_{Max}$ = 1/159.06 MHz = 6.2894 ns
- Delay = 6.287 ns * 247 cycles = 1552.87418 ns
- Area x Delay = $10.966 * 10^{-3}$

References

- https://en.wikipedia.org/wiki/SHA-1