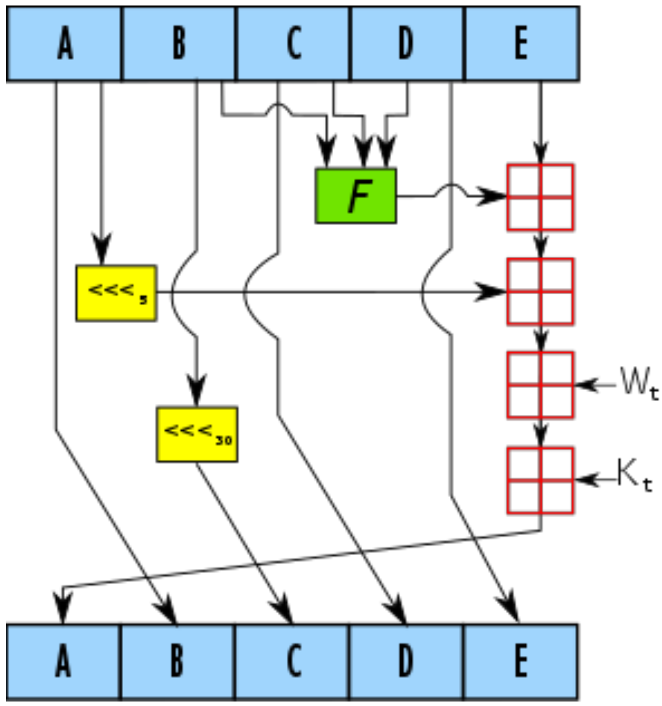
Program – 9

AIM: To implement a program to calculate hash by SHA1.

Introduction and Theory

In computer cryptography, a popular message compress standard is utilized known as Secure Hash Algorithm (SHA). Its enhanced version is called SHA-1. It has the ability to compress a fairly lengthy message and create a short message abstract in response. The algorithm can be utilized along various protocols to ensure security of the applied algorithm, particularly for Digital Signature Standard (DSS). The algorithm offers five separate hash functions which were created by National Security Agency (NSA) and were issued by the National Institute of Standards and Technology (NIST).



According to SHA-1 standard, a message digest is evaluated utilizing padded message. The evaluation utilizes two buffers, each comprises of five 32 bit words and a sequence of eighty 32 bit words. The words of the first five-word buffer are labelled as A, B, C, D and E. The words of the second five-word buffer are labelled as H0, H1, H2, H3 and H4. The words of the eighty-word sequence are labelled as W0, W1, W2 to W79. SHA1 operates blocks of 512 bits, when evaluating a message digest. The entire extent lengthwise of message digest shall be multiple of 512. A novel architecture of SHA-1 for enhanced throughput and decreased area, in which at the same time diverse acceleration techniques are exerted like pre-computation, loop unfolding and pipelining. Hash function requires a set of operations that an input of diversifying length and create a stable length string which is known as the hash value or message digest.

SHA-1 hash architecture has been occupied utilizing Visual Hardware Description Language (VHDL) and executed in Xilinx 13.2. It utilizes transformed Carry Save Adder so as to achieve enhanced throughput and decreased area. The recommended pipelined architecture has achieved a throughput of 8.6 Gbps and 1230 slices, with the integration of diverse acceleration techniques. When compared with prior work, it has been observed that the recommended execution shows 17% improved throughput as well as 25% additional dense architecture. Loop unfolding is a technique which exploits the combinational logic to execute several rounds in only one clock cycle. Pipelining is a technique in which the architecture is break into 'n' number of steps in which independent estimations are executed. Pre-computation technique is utilized to produce definite intermediate signals of the critical path and reserve them in a register, which can be utilized in the computation of values of next step. For a message possessing a maximum length of 264, SHA-1 constructs a 160 bit message digest. `
160 bit dedicated hash function is incorporated in SHA-1 originate in the design principle of MD4, which is an algorithm utilized to certify data integrity through the formation of a 128 bit message digest from data input that is declared to be as distinctive to that particular data as a

Program – 9

fingerprint is to the particular individual. It implements the Merkle-Damgard paradigm to a dedicated compression function. The input message is padded and break into ‘k’ 512 bit message blocks. At every iteration of the compression function ‘h’, a 160 bit chaining variable Ht is upgraded utilizing one message block Mt+1, that is Ht+1 = h(Ht, Mt+1). The beginning value H0 is established in advance and Hk is the out-turn of the hash function. SHA-1 compression function is constructed upon the Davis Meyer construction. It utilizes a function ‘E’ as a block cipher with Ht for the message input and Mt+1 for the key input.
The SHA-1 is implicit easily. It is as secure as anything in opposition to reimaged attacks, although it is effortless to calculate, which means it is uncomplicated to mount a brute force or dictionary attack. It is a well-known cryptographic primitive which ensures the integrity and reliability of original message.

Code

Table 1 SHA.h

```
#ifndef SHA1_HPP
#define SHA1_HPP


#include <cstdint>
#include <iostream>
#include <string>
#include <stdint.h>

class SHA1
{
public:
    SHA1();
    void update(const std::string &s);
    void update(std::istream &is);
    std::string final();
    static std::string from_file(const std::string &filename);

private:
    uint32_t digest[5];
    std::string buffer;
    uint64_t transforms;
};


#endif /* SHA1_HPP */
```

Table 2 SHA.cpp

```
#include "sha1.h"
#include <sstream>
#include <iomanip>
#include <fstream>


static const size_t BLOCK_INTS = 16;  /* number of 32bit integers
per SHA1 block */
static const size_t BLOCK_BYTES = BLOCK_INTS * 4;
```

Program – 9

```


static void reset(uint32_t digest[], std::string &buffer, uint64_t
&transforms)
{
    /* SHA1 initialization constants */
    digest[0] = 0x67452301;
    digest[1] = 0xefcdab89;
    digest[2] = 0x98badcfe;
    digest[3] = 0x10325476;
    digest[4] = 0xc3d2e1f0;

    /* Reset counters */
    buffer = "";
    transforms = 0;
}


static uint32_t rol(const uint32_t value, const size_t bits)
{
    return (value << bits) | (value >> (32 - bits));
}


static uint32_t blk(const uint32_t block[BLOCK_INTS], const size_t
i)
{
    return rol(block[(i+13)&15] ^ block[(i+8)&15] ^ block[(i+2)&15]
^ block[i], 1);
}


/*
 * (R0+R1), R2, R3, R4 are the different operations used in SHA1
 */

static void R0(const uint32_t block[BLOCK_INTS], const uint32_t v,
uint32_t &w, const uint32_t x, const uint32_t y, uint32_t &z, const
size_t i)
{
    z += ((w&(x^y))^y) + block[i] + 0x5a827999 + rol(v, 5);
    w = rol(w, 30);
}
static void R1(uint32_t block[BLOCK_INTS], const uint32_t v,
uint32_t &w, const uint32_t x, const uint32_t y, uint32_t &z, const
size_t i)
{
    block[i] = blk(block, i);
    z += ((w&(x^y))^y) + block[i] + 0x5a827999 + rol(v, 5);
    w = rol(w, 30);
}
static void R2(uint32_t block[BLOCK_INTS], const uint32_t v,
uint32_t &w, const uint32_t x, const uint32_t y, uint32_t &z, const
size_t i)
{
    block[i] = blk(block, i);
    z += (w^x^y) + block[i] + 0x6ed9eba1 + rol(v, 5);
```

Program – 9

```
    w = rol(w, 30);
}
static void R3(uint32_t block[BLOCK_INTS], const uint32_t v,
uint32_t &w, const uint32_t x, const uint32_t y, uint32_t &z, const
size_t i)
{
    block[i] = blk(block, i);
    z += (((w|x)&y)|(w&x)) + block[i] + 0x8f1bbcdc + rol(v, 5);
    w = rol(w, 30);
}
static void R4(uint32_t block[BLOCK_INTS], const uint32_t v,
uint32_t &w, const uint32_t x, const uint32_t y, uint32_t &z, const
size_t i)
{
    block[i] = blk(block, i);
    z += (w^x^y) + block[i] + 0xca62c1d6 + rol(v, 5);
    w = rol(w, 30);
}
/*
 * Hash a single 512-bit block. This is the core of the algorithm.
 */
static void transform(uint32_t digest[], uint32_t block[BLOCK_INTS],
uint64_t &transforms)
{
    /* Copy digest[] to working vars */
    uint32_t a = digest[0];
    uint32_t b = digest[1];
    uint32_t c = digest[2];
    uint32_t d = digest[3];
    uint32_t e = digest[4];
    /* 4 rounds of 20 operations each. Loop unrolled. */
    R0(block, a, b, c, d, e,  0);
    R0(block, e, a, b, c, d,  1);
    R0(block, d, e, a, b, c,  2);
    R0(block, c, d, e, a, b,  3);
    R0(block, b, c, d, e, a,  4);
    R0(block, a, b, c, d, e,  5);
    R0(block, e, a, b, c, d,  6);
    R0(block, d, e, a, b, c,  7);
    R0(block, c, d, e, a, b,  8);
    R0(block, b, c, d, e, a,  9);
    R0(block, a, b, c, d, e, 10);
    R0(block, e, a, b, c, d, 11);
    R0(block, d, e, a, b, c, 12);
    R0(block, c, d, e, a, b, 13);
    R0(block, b, c, d, e, a, 14);
    R0(block, a, b, c, d, e, 15);
    R1(block, e, a, b, c, d,  0);
    R1(block, d, e, a, b, c,  1);
    R1(block, c, d, e, a, b,  2);
    R1(block, b, c, d, e, a,  3);
    R2(block, a, b, c, d, e,  4);
    R2(block, e, a, b, c, d,  5);
    R2(block, d, e, a, b, c,  6);
    R2(block, c, d, e, a, b,  7);
    R2(block, b, c, d, e, a,  8);
    R2(block, a, b, c, d, e,  9);
```

Program – 9

```
    R2(block, e, a, b, c, d, 10);
    R2(block, d, e, a, b, c, 11);
    R2(block, c, d, e, a, b, 12);
    R2(block, b, c, d, e, a, 13);
    R2(block, a, b, c, d, e, 14);
    R2(block, e, a, b, c, d, 15);
    R2(block, d, e, a, b, c,  0);
    R2(block, c, d, e, a, b,  1);
    R2(block, b, c, d, e, a,  2);
    R2(block, a, b, c, d, e,  3);
    R2(block, e, a, b, c, d,  4);
    R2(block, d, e, a, b, c,  5);
    R2(block, c, d, e, a, b,  6);
    R2(block, b, c, d, e, a,  7);
    R3(block, a, b, c, d, e,  8);
    R3(block, e, a, b, c, d,  9);
    R3(block, d, e, a, b, c, 10);
    R3(block, c, d, e, a, b, 11);
    R3(block, b, c, d, e, a, 12);
    R3(block, a, b, c, d, e, 13);
    R3(block, e, a, b, c, d, 14);
    R3(block, d, e, a, b, c, 15);
    R3(block, c, d, e, a, b,  0);
    R3(block, b, c, d, e, a,  1);
    R3(block, a, b, c, d, e,  2);
    R3(block, e, a, b, c, d,  3);
    R3(block, d, e, a, b, c,  4);
    R3(block, c, d, e, a, b,  5);
    R3(block, b, c, d, e, a,  6);
    R3(block, a, b, c, d, e,  7);
    R3(block, e, a, b, c, d,  8);
    R3(block, d, e, a, b, c,  9);
    R3(block, c, d, e, a, b, 10);
    R3(block, b, c, d, e, a, 11);
    R4(block, a, b, c, d, e, 12);
    R4(block, e, a, b, c, d, 13);
    R4(block, d, e, a, b, c, 14);
    R4(block, c, d, e, a, b, 15);
    R4(block, b, c, d, e, a,  0);
    R4(block, a, b, c, d, e,  1);
    R4(block, e, a, b, c, d,  2);
    R4(block, d, e, a, b, c,  3);
    R4(block, c, d, e, a, b,  4);
    R4(block, b, c, d, e, a,  5);
    R4(block, a, b, c, d, e,  6);
    R4(block, e, a, b, c, d,  7);
    R4(block, d, e, a, b, c,  8);
    R4(block, c, d, e, a, b,  9);
    R4(block, b, c, d, e, a, 10);
    R4(block, a, b, c, d, e, 11);
    R4(block, e, a, b, c, d, 12);
    R4(block, d, e, a, b, c, 13);
    R4(block, c, d, e, a, b, 14);
    R4(block, b, c, d, e, a, 15);

    /* Add the working vars back into digest[] */
    digest[0] += a;
```

Program – 9

```
    digest[1] += b;
    digest[2] += c;
    digest[3] += d;
    digest[4] += e;

    /* Count the number of transformations */
    transforms++;
}
static void buffer_to_block(const std::string &buffer, uint32_t
block[BLOCK_INTS])
{
    /* Convert the std::string (byte buffer) to a uint32_t array
(MSB) */
    for (size_t i = 0; i < BLOCK_INTS; i++)
    {
        block[i] = (buffer[4*i+3] & 0xff)
                   | (buffer[4*i+2] & 0xff)<<8
                   | (buffer[4*i+1] & 0xff)<<16
                   | (buffer[4*i+0] & 0xff)<<24;
    }
}
SHA1::SHA1()
{
    reset(digest, buffer, transforms);
}
void SHA1::update(const std::string &s)
{
    std::istringstream is(s);
    update(is);
}
void SHA1::update(std::istream &is)
{
    while (true)
    {
        char sbuf[BLOCK_BYTES];
        is.read(sbuf, BLOCK_BYTES - buffer.size());
        buffer.append(sbuf, (std::size_t)is.gcount());
        if (buffer.size() != BLOCK_BYTES)
        {
            return;
        }
        uint32_t block[BLOCK_INTS];
        buffer_to_block(buffer, block);
        transform(digest, block, transforms);
        buffer.clear();
    }
}


/*
 * Add padding and return the message digest.
 */

std::string SHA1::final()
{
    /* Total number of hashed bits */

```

Program – 9

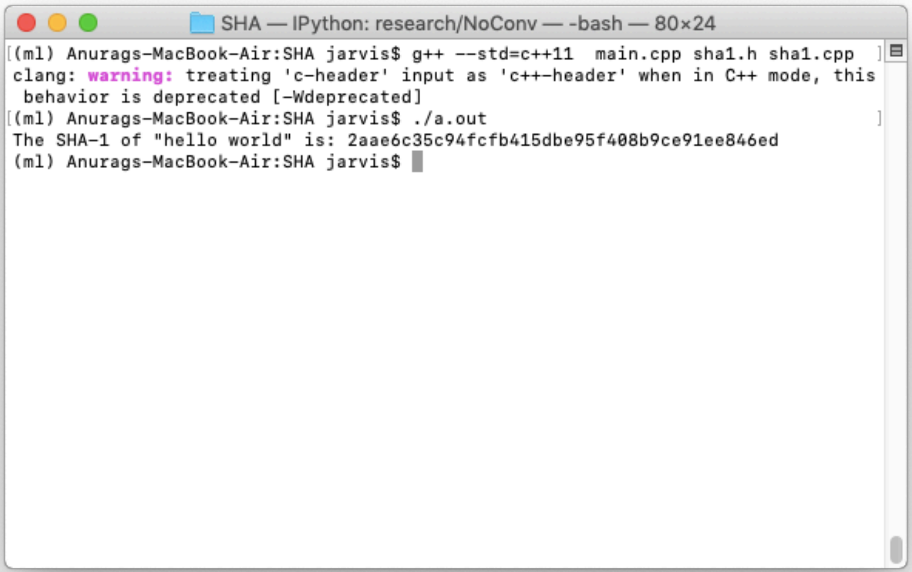
```
    uint64_t total_bits = (transforms*BLOCK_BYTES + buffer.size()) *
8;
    /* Padding */
    buffer += (char)0x80;
    size_t orig_size = buffer.size();
    while (buffer.size() < BLOCK_BYTES)
    {
        buffer += (char)0x00;
    }

    uint32_t block[BLOCK_INTS];
    buffer_to_block(buffer, block);

    if (orig_size > BLOCK_BYTES - 8)
    {
        transform(digest, block, transforms);
        for (size_t i = 0; i < BLOCK_INTS - 2; i++)
        {
            block[i] = 0;
        }
    }

    /* Append total_bits, split this uint64_t into two uint32_t */
    block[BLOCK_INTS - 1] = (uint32_t)total_bits;
    block[BLOCK_INTS - 2] = (uint32_t)(total_bits >> 32);
    transform(digest, block, transforms);

    /* Hex std::string */
    std::ostringstream result;
    for (size_t i = 0; i < sizeof(digest) / sizeof(digest[0]); i++)
    {
        result << std::hex << std::setfill('0') << std::setw(8);
        result << digest[i];
    }

    /* Reset for next run */
    reset(digest, buffer, transforms);

    return result.str();
}


std::string SHA1::from_file(const std::string &filename)
{
    std::ifstream stream(filename.c_str(), std::ios::binary);
    SHA1 checksum;
    checksum.update(stream);
    return checksum.final();
}
```

Program – 9

Table 3 main.cpp

```cpp
#include "sha1.h"
#include <string>
#include <iostream>
using std::string;
using std::cout;
using std::endl;

int main(int /* argc */, const char ** /* argv */)
{
    const string input = "hello world";

    SHA1 checksum;
    checksum.update(input);
    const string hash = checksum.final();

    cout << "The SHA-1 of \"" << input << "\" is: " << hash << endl;

    return 0;
}
```

Results and Outputs:



Findings and Learnings:

1. We have implemented SHA1 hash.