CS577: SHA256 Project Report

Group Number:	11
Name of the top modules:	sha256
Link for GitHub Repo:	https://github.com/cs577vlsi

Group Members	Roll Numbers
Pratik kulkar	204101031
Anuj Kumar	204101011
Ram Avtar Sahu	204101044
Mukul Chaturvedi	204101035
Aditya Tandon	204101003

INTRODUCTION

SHA-256: A cryptographic hash (also known as 'digest') is a kind of 'signature' for a text or a data file. SHA-256 generates an almost-unique 256-bit (32-byte) signature for a text. SHA-256 is one of the successor hash functions to SHA-1 and is one of the strongest hash functions available.

Algorithm:

- ❏ All variables are 32 bit unsigned integers and addition is calculated modulo 232
- ❏ For each round, there is one round constant k[i] and one entry in the message schedule array w[i], 0 ≤ i ≤ 63
- ❏ The compression function uses 8 working variables, a through h
- ❏ Big-endian convention is used when expressing the constants in this pseudocode, and when parsing message block data from bytes to words, for example, the first word of the input message "abc" after padding is 0x61626380

1. [Initialize hash values]
   first 32 bits of the fractional parts of the square roots of the first 8 primes 2..19.
2. [Initialize array of round constants]
   first 32 bits of the fractional parts of the cube roots of the first 64 primes 2..311.
3. [Pre-processing (Padding)]:
   begin with the original message of length L bits
   append a single '1' bit
   append K '0' bits, where K is the minimum number >= 0 such that L + 1 + K + 64 is a multiple of 512
   append L as a 64-bit big-endian integer, making the total post-processed length a multiple of 512 bits
4. (Process the message in successive 512-bit chunks)
   break message into 512-bit chunks
   for each chunk do:
   - create a 64-entry message schedule array w[0..63] of 32-bit words
   - copy chunk into first 16 words w[0..15] of the message schedule array

   for i = 16 to 63 do: ***RCS:right circular shift***
   - s0 := (w[i-15] RCS 7) xor (w[i-15] RCS 18) xor (w[i-15] RCS 3)
   - s1 := (w[i- 2] RCS 17) xor (w[i- 2] RCS 19) xor (w[i- 2] RCS 10)
   - w[i] := w[i-16] + s0 + w[i-7] + s1

   Initialize working variables(a-h) to current hash value(h0-h7).

for i = 0 to 63 do:

- S1 := (e RCS 6) xor (e RCS 11) xor (e RCS 25)
- choice := (e and f) xor ((not e) and g)
- t1 := h + S1 + choice+ k[i] + w[i]
- S0 := (a RCS 2) xor (a RCS 13) xor (a RCS 22)
- majority := (a and b) xor (a and c) xor (b and c)
- t2 := S0 + majority
- h = g
- g = f
- f = e
- e = d + t1
- d = c
- c = b
- b = a
- a = t1+ t2

(Update the current hash value)

h0 := h0 + a

h1 := h1 + b

h2 := h2 + c

h3 := h3 + d

h4 := h4 + e

h5 := h5 + f

h6 := h6 + g

h7 := h7 + h

5. Final hash value:

   hash = concat(h0,h1,h2,h3,h4,h5,h6,h7)

PHASE-1

1. Running the algorithm

1.1 Simulation output:

The top function module was simulated successfully. Here's the screenshot attached for the same.

```
   Generating csim.exe
Data to hash: ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTU
VWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUV

Test 0, base: 0 length 0:
SHA256_SW: E3 B0 C4 42 98 FC 1C 14 9A FB F4 C8 99 6F B9 24 27 AE 41 E4 64 9B 93 4C A4 95 99 1B 78 52 B8 55
SHA256_HW: E3 B0 C4 42 98 FC 1C 14 9A FB F4 C8 99 6F B9 24 27 AE 41 E4 64 9B 93 4C A4 95 99 1B 78 52 B8 55
PASSED

Test 1, base: 1 length 1:
SHA256_SW: DF 7E 70 E5 02 15 44 F4 83 4B BE E6 4A 9E 37 89 FE BC 4B E8 14 70 DF 62 9C AD 6D DB 03 32 0A 5C
SHA256_HW: DF 7E 70 E5 02 15 44 F4 83 4B BE E6 4A 9E 37 89 FE BC 4B E8 14 70 DF 62 9C AD 6D DB 03 32 0A 5C
PASSED

Test 2, base: 0 length 64:
SHA256_SW: 9B 0C 35 67 68 3F 6D 51 98 D7 14 B0 0C 40 DF 30 39 F7 71 7B A5 CA 65 C2 B6 61 76 88 B6 3B 42 FA
SHA256_HW: 9B 0C 35 67 68 3F 6D 51 98 D7 14 B0 0C 40 DF 30 39 F7 71 7B A5 CA 65 C2 B6 61 76 88 B6 3B 42 FA
PASSED

Test 3, base: 3 length 127:
SHA256_SW: A2 23 36 C2 73 7D D7 47 79 DF DB 12 3C 07 3A 04 47 5B 9C 4F 5E F2 E0 F7 A5 CB 27 09 DF 85 31 CE
SHA256_HW: A2 23 36 C2 73 7D D7 47 79 DF DB 12 3C 07 3A 04 47 5B 9C 4F 5E F2 E0 F7 A5 CB 27 09 DF 85 31 CE
PASSED

Test 4, base: 0 length 256:
SHA256_SW: 6C 50 76 06 1B 0C C3 1F 39 87 76 7C 06 2C D1 33 AB 13 07 34 A0 B8 18 4C 65 D0 65 88 18 23 B9 92
SHA256_HW: 6C 50 76 06 1B 0C C3 1F 39 87 76 7C 06 2C D1 33 AB 13 07 34 A0 B8 18 4C 65 D0 65 88 18 23 B9 92
PASSED

INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

1.2 Synthesis output:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.699	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
461	18187	461	18187	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	295	-
FIFO	-	-	-	-	-
Instance	4	-	24119	49656	-
Memory	5	-	0	0	0
Multiplexer	-	-	-	845	-
Register	-	-	1037	-	-
Total	9	0	25156	50796	0
Available	720	1152	290880	145440	0
Utilization (%)	1	0	8	34	0

1.3 C/RTL co-simulation output:

Cosimulation Report for 'sha256'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	1282	1890	2885	1283	1642	2096

Export the report(.html) using the Export Wizard

2. Result:

FPGA Part	Name of Top Module	FF	LUT	BRAM	Latency Minimum	Latency Maximum
ZYNQ-7 ZC702 Evaluation Board	sha256	4410	6303	12	862	58126

2.1 Explanation of result:

In this project our top function is SH256, which computes the hash of given input

- In the top function, a total of 4410 flip-flops are used as a register.
- The 6303 Lookup table is used in which 5466 is used for instance, 241 is used for Expressions, Multiplexer uses 596 LUT.
- 10 BRAM is used as memory to store instances of modules such as sha256 and remaining are used to store data and temporary hash values.
- Latency of the top function is of at least 862 cycles and can go upto 58126.

2.2 Problems and its solution

We addressed following issues while running and optimizing the code:

- undefined latency

  When we ran the c-synthesis for the first time. The latency was undefined. To resolve this issue we had to find each loop with variable size and we added loop_tripcount pragmas with different set of values and finally we were able to get the latency of **58126** with FF **4410** & LUT **6303** This is the maximum latency we got so far. Now after we have this high latency value, our target is to reduce this value as low as possible by applying correct pragmas at different places.

- Pipeline led to increase in resource utilization:

  In sha_final function, for static size of loop we tried to introduce a pragma pipeline with iteration value 2. It resulted in increasing the hardware utilization drastically.

  Solution:

  We have tried to change the board with ‘Kintex UltraScale’. This resulted in about 40% decrease in FF utilization and 25% decrease in latency.

PHASE-2

The target FPGA board is **Kintex UltraScale board**

Benchmarks	Type	Resource Utilization				Latency		Major Optimization
		LUT	FF	DSP	BRAM	Clock Cycles	Clock Period	
Baseline	Baseline	4410	6303	0	12	58126	8.522	No Optimization
Optimization 1	Latency	97867	33600	0	9	18085	8.75	Latency
Optimization 2	Area	50796	25156	0	9	18187	8.69	LUT,FF

Table 1

Explanation for optimization-1:

- **Consider the following code snippet**

```
   for (i=0,j=0; i < 16; ++i, j += 4){
#pragma HLS array_partition variable=data cyclic factor=4 dim=1
      m[i] = (data[j] << 24) | (data[j+1] << 16) | (data[j+2] << 8) | (data[j+3]);
   }
```

In the sha256_transform module in sha_impl.c while making a block of 32 bit the array was accessed in the following format data[j], data[j+1],data[j+1],data[j+2],data[j+3]. So, we have partitioned the array with factor = 4 in a cyclic fashion where dimension is 1. Hence we can each element in a particular element which takes only 1 cycle instead of 4 to access 4 elements.

- **Consider the following code snippet**

```
   for ( ; i < 64; ++i){
#pragma HLS pipeline II=1
      m[i] = SIG1(m[i-2]) + m[i-7] + SIG0(m[i-15]) + m[i-16];
   }
```

In the above code we can see that we can do one operation after the other like in a pipeline and we estimated that there won't be a stall because each memory fetch operation will take similar time and we have adders available with us.

- Since the remaining optimization are on loops the reason they works is same as the above

optimization and remaining code snippets where optimization applied is as follows:

```
   for (i = 0; i < 64; ++i) {
#pragma HLS pipeline II=1
      t1 = h + EP1(e) + CH(e,f,g) + k[i] + m[i];
      t2 = EP0(a) + MAJ(a,b,c);
      h = g;
      g = f;
      f = e;
      e = d + t1;
      d = c;
      c = b;
      b = a;
      a = t1 + t2;
   }
```

```
void sha256_final(SHA256_CTX *ctx, uchar hash[])
{
    uint i;

    i = ctx->datalen;
#pragma HLS pipeline II=1
    // Pad whatever data is left in the buffer.
    if (ctx->datalen < 56) {
        ctx->data[i++] = 0x80;
        while (i < 56){
#pragma HLS loop_tripcount min=0 max=56
            ctx->data[i++] = 0x00;
        }
    }
    else {
        ctx->data[i++] = 0x80;
        while (i < 64){
#pragma HLS loop_tripcount min=0 max=8
            ctx->data[i++] = 0x00;
        }
        sha256_transform(ctx,ctx->data);
        for( i = 0; i < 56; i++ ) {

            ctx->data[i] = 0x00;
        }
    }
```

```
   // reverse all the bytes when copying the final state to the output hash.
   for (i=0; i < 4; ++i) {
#pragma HLS pipeline II=1
      hash[i]    = (ctx->state[0] >> (24-i*8)) & 0x000000ff;
      hash[i+4]  = (ctx->state[1] >> (24-i*8)) & 0x000000ff;
      hash[i+8]  = (ctx->state[2] >> (24-i*8)) & 0x000000ff;
      hash[i+12] = (ctx->state[3] >> (24-i*8)) & 0x000000ff;
      hash[i+16] = (ctx->state[4] >> (24-i*8)) & 0x000000ff;
      hash[i+20] = (ctx->state[5] >> (24-i*8)) & 0x000000ff;
      hash[i+24] = (ctx->state[6] >> (24-i*8)) & 0x000000ff;
      hash[i+28] = (ctx->state[7] >> (24-i*8)) & 0x000000ff;
   }
```

- **Consider the following code snippet**

```
void sha256_init(SHA256_CTX *ctx)
{
#pragma HLS inline
   ctx->datalen = 0;
   ctx->bitlen[0] = 0;
   ctx->bitlen[1] = 0;
   ctx->state[0] = 0x6a09e667;
   ctx->state[1] = 0xbb67ae85;
   ctx->state[2] = 0x3c6ef372;
   ctx->state[3] = 0xa54ff53a;
   ctx->state[4] = 0x510e527f;
   ctx->state[5] = 0x9b05688c;
   ctx->state[6] = 0x1f83d9ab;
   ctx->state[7] = 0x5be0cd19;
}
```

  here sha_init module is called multiple times so we have made it an inline function so that time taken by switching the module part will be saved.

Explanation for optimization-2:

- Originally in the program they have used macros for operations used in the SHA-2 algorithm but when we have used functions instead of macros we had found a drastic reduction in the number of resources. So, we replaced all the macros with the function which gave the reduction of nearly 47071 for LUTs and 8444 for FF.
- The reason behind this is, the file is preprocessed by these macros before compilation and an expanded source file is given to the compiler for further execution whereas in case of function it creates a separate module for it but only once and the process will switch to that module whenever necessary hence no operation is executed before compilation.
- As we can see in Table 1 without hardware optimization our latency is 18085 and with hardware optimization latency is 18187. So, our latency increases by a very small factor and since the number of hardware components decreases drastically we can bear this much amount of increase in latency.

Conclusion:

Initially the program was taking around 60000 cycles and the board was ZYNQ-7 ZC702 Evaluation Board but while optimizing latency we felt a need to change the board. So we changed the board to **Kintex UltraScale board** but once we have to optimize the area the resources were so less that the program can now run on ZYNQ-7 ZC702. But changing to ZYNQ-7 ZC702 has increased the latency to around 24000, so we decided to go with **Kintex UltraScale board.**

Outcomes:

Final Latency : 18187

Final LUTs : 50796

Final FF : 25156

Final BRAM: 9