

# Lab#B-SHA256

R09922190 王祥任

Github Repo: <https://github.com/allen880117/ACA-HLS-Lab-B-SHA256>

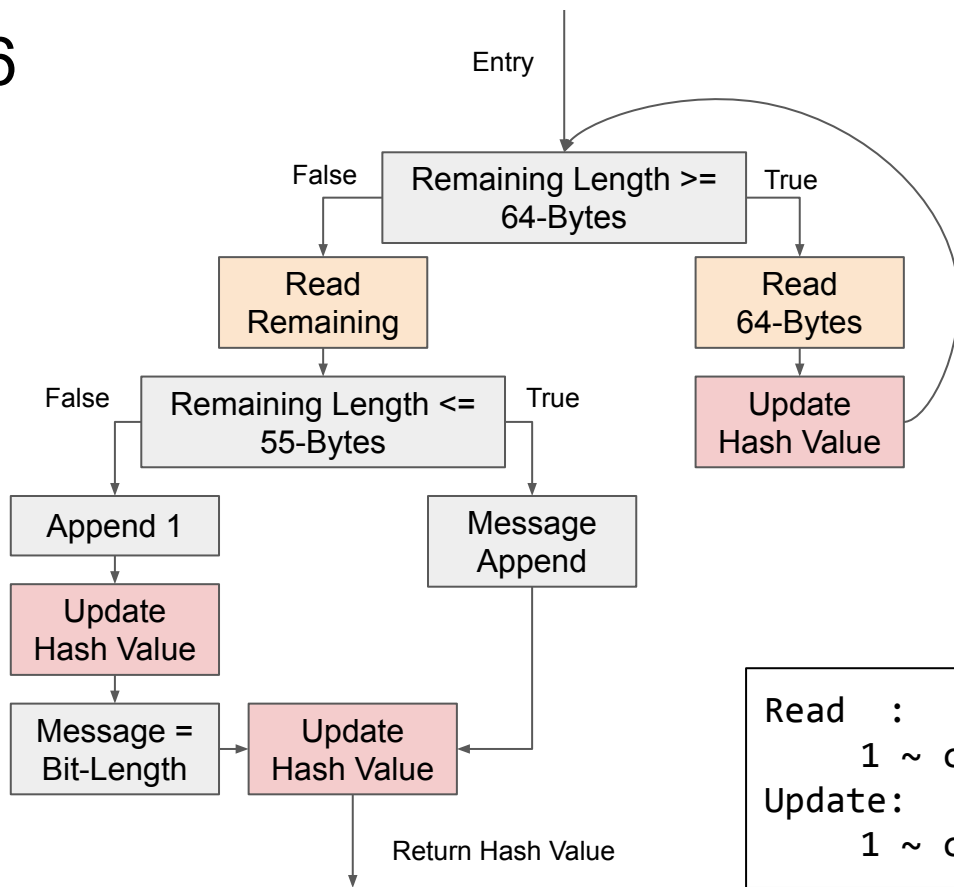
# Outline

- SHA256
- Implementation
- Comparison
- SILEXICA Tool
- On-Board
- Conclusion

# Outline

- **SHA256**
- Implementation
- Comparison
- SILEXICA Tool
- On-Board
- Conclusion

# SHA256

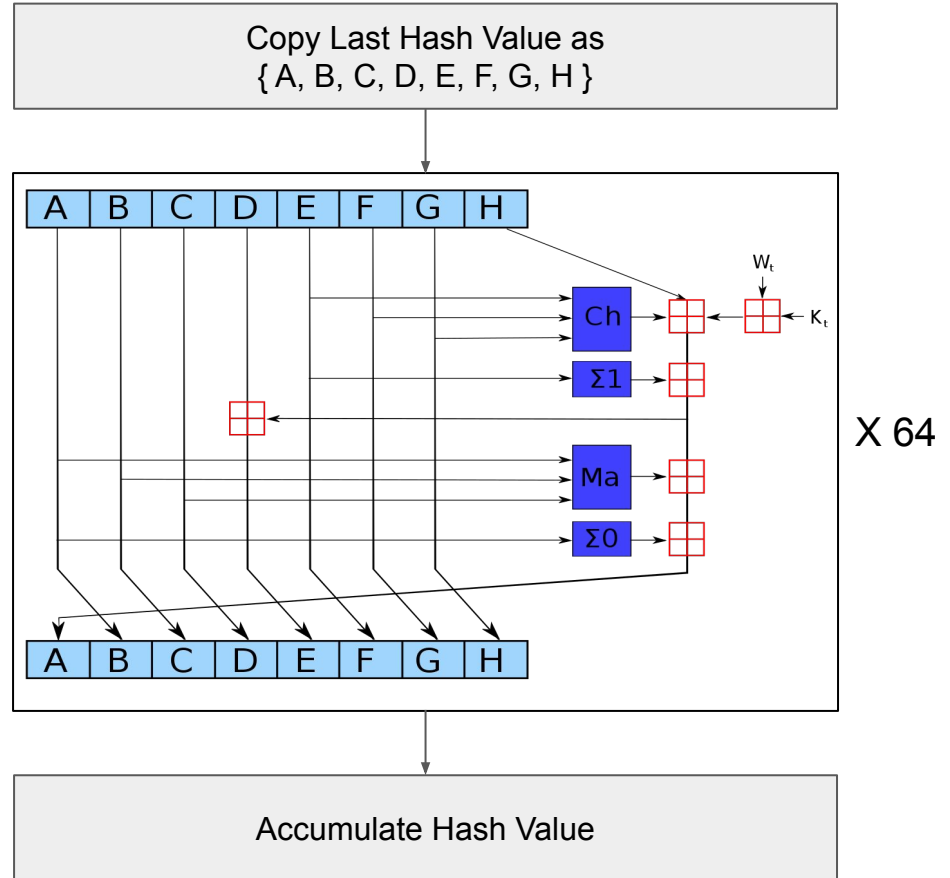


Read :  
1 ~  $\text{ceil}(\text{Length}(\text{bits}) / 512)$   
Update:  
1 ~  $\text{ceil}(\text{Length}(\text{bits}) / 512) + 1$

# SHA256 - Update

K[ 0~63] = Predefined Parameters

```
W[ 0~15] = Data( 511-i*32, 480-i*32 )
```

$$W[16 \sim 63] = \text{SIG1}(W[i-2]) + W[i-7] + \text{SIG0}(W[i-15]) + W[i-16]$$


X 64

# Outline

- SHA256
- **Implementation**
- Comparison
- SILEXICA Tool
- On-Board
- Conclusion

# Implementation - Interface

- Master-AXI
  - For Burst Read

```
u256_t sha256(volatile char* ctx_mem, u64_t ctx_len){  
#pragma HLS INTERFACE m_axi depth=512 port=ctx_mem offset=slave  
#pragma HLS INTERFACE s_axilite port=ctx_len  
#pragma HLS INTERFACE s_axilite port=return  
  
#if !DOUBLE_BUFFER  
    return sha256_main_basic(ctx_mem, ctx_len);  
#else  
    return sha256_main_DBUF(ctx_mem, ctx_len);  
#endif  
}
```

# Implementation - Read - Burst Read

```
void sha256_read(volatile char* ctx_mem, u512_t &data, u64_t offset,
                 u32_t byte_len) {
    READ:
        for (u32_t i = 0; i < byte_len; i++) {
            #pragma HLS LOOP_TRIPCOUNT min=0 max=64

            #if BASIC_OPT
            #pragma HLS PIPELINE
            #endif

            data(511 - i * 8, 504 - i * 8) = (u8_t)ctx_mem[offset + i];
        }
}
```

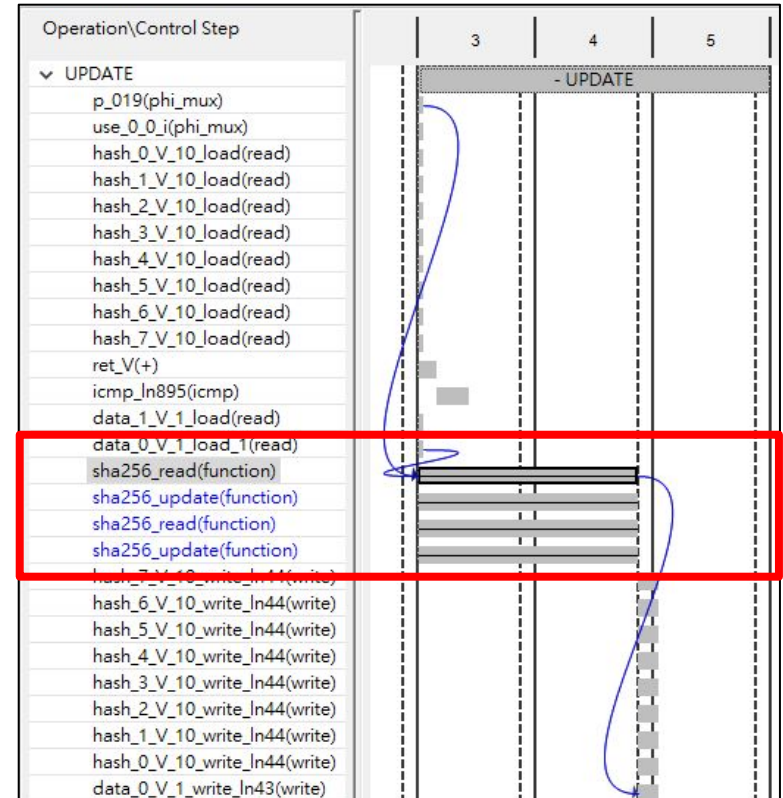
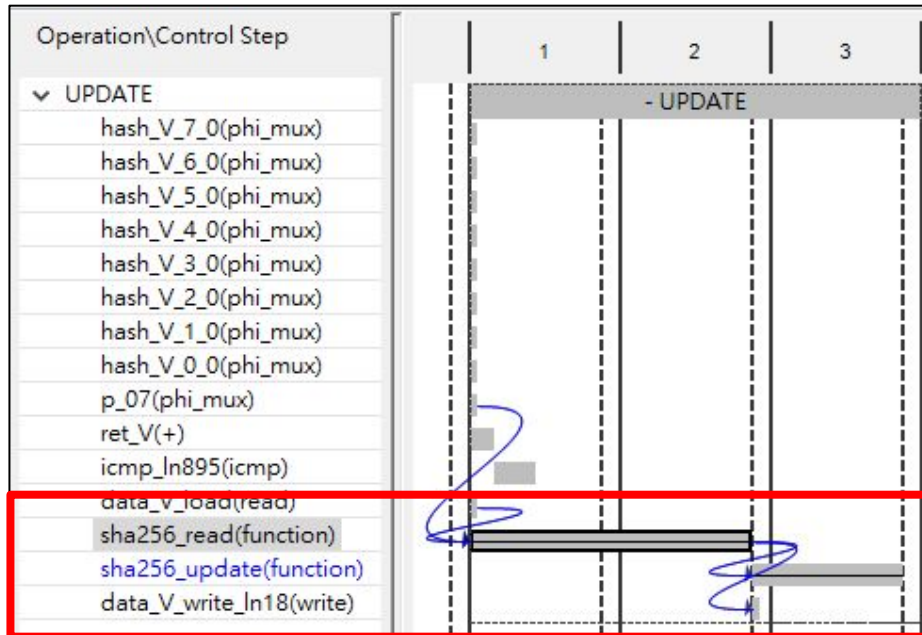


# Implementation - Read - Double Buffer

- Overlap “Read” and “Update”
- In my implementation
  - **Read** : about **74** cycles
  - **Update** : about **67** cycles

```
/* Proceed 512-bits (64-bytes) chunk and Update Hash Values*/  
if (use_0) {  
    sha256_read(ctx_mem, data_1, offset, 64);  
    sha256_update(data_0, hash);  
} else {  
    sha256_read(ctx_mem, data_0, offset, 64);  
    sha256_update(data_1, hash);  
}  
  
/* Ping-Pong */  
use_0 = (!use_0);
```

# Implementation - Read - Double Buffer



# Implementation - Update - Calculation of W[0~15]

There is no dependency of **First 16 W** between different iterations of the loop.

We can calculate all these **W in one cycle by unroll the loop completely.**

```
/* Assign First 16 W */  
ASSIGN_M_0_16:  
    for (u32_t i = 0; i<16; i++){  
        #if USE_HLS_PRAGMA  
        #pragma HLS UNROLL  
        #endif  
        u32_t tmp_wi = data(511-i*32, 480-i*32);  
        w[i]      = tmp_wi;  
        wsig0[i] = SIG0(tmp_wi);  
        wsig1[i] = SIG1(tmp_wi);  
    }
```

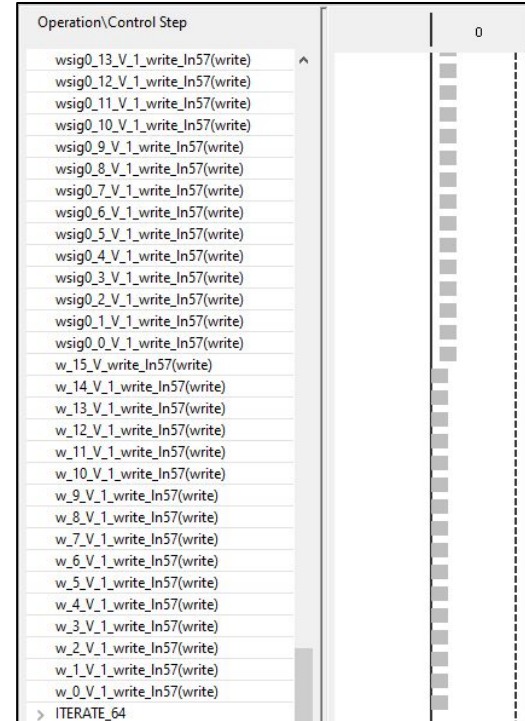
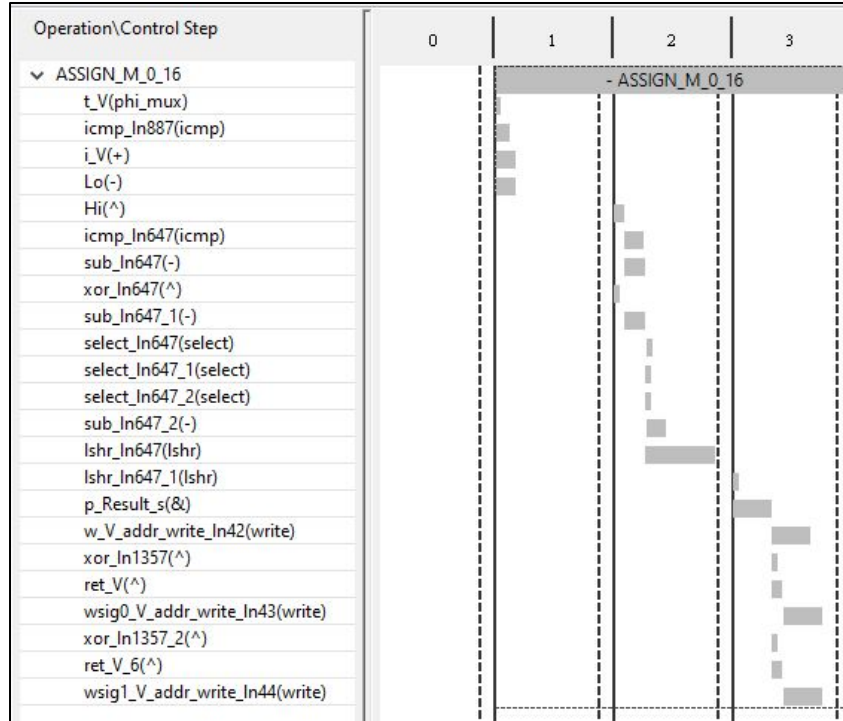
# Implementation - Update - Calculation of W[0~15] (cont.)

To reduce the critical path of the following calculation for W[16~63], we can calculate the **Wsig0** and **Wsig1** with **W** at same time.

```
/* Assign First 16 W */
ASSIGN_M_0_16:
    for (u32_t i = 0; i<16; i++){
        #if USE_HLS_PRAGMA
        #pragma HLS UNROLL
        #endif

        u32_t tmp_wi = data(511-i*32, 480-i*32);
        w[i] = tmp_wi;
        wsig0[i] = SIG0(tmp_wi);
        wsig1[i] = SIG1(tmp_wi);
    }
```

# Implementation - Update - Calculation of W[0~15] (cont.)



# Implementation - Update - Calculation of W[16~63]

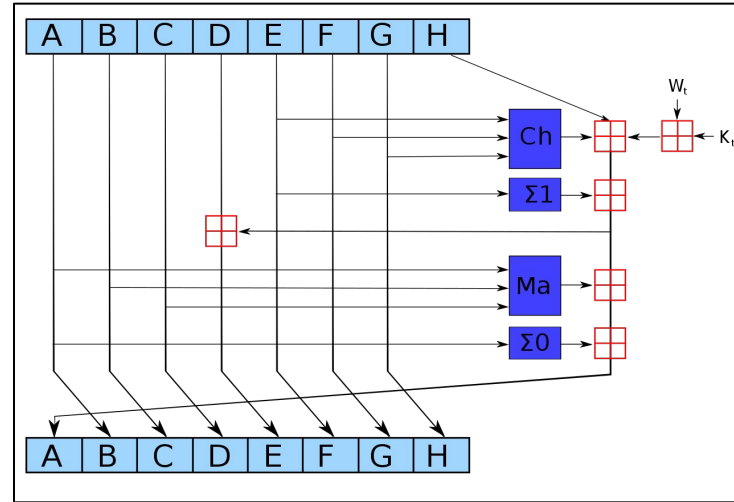
- There **exists dependency of different W between different iterations of the loop**, so we can't directly unroll the loop.
- To reduce the cycle for the whole process, we merge the calculation of W[16~63] into the following **64 iterations** manually.

```
W[0~15]  = Data( 511-i*32, 480-i*32 )
```

```
W[16~63] = SIG1( W[i- 2] ) + W[i- 7] +  
              SIG0( W[i-15] ) + W[i-16]
```

# Implementation - Update - 64 Iterations

- Since there **exists dependency of {a, b, c, d, e, f, g, h} between different iterations of the loop**, it's impossible to unroll the loop.
- Try to “**pipeline**” it.



# Implementation - Update - 64 Iterations - $W[16\sim 63]$

Each  $W[i]$  depends on :

1.  $SIG1( W[i-2] )$
2.  $W[i-7]$
3.  $SIG0( W[i-15] )$
4.  $W[i-16]$

Since we already have  $W[0\sim 15]$ , in the  $i$ -th iterations, we can calculate  $W[i+16]$  at first.

Then at  $(i+16)$ -th, we don't have to calculate  $W[i+16]$ .

```
/* Temporary Wi */
u32_t tmp_wi = w[i];

/* Forward Calculation
 * 1. For Reducing the Critical Path
 */
if (i < 64-16) {
    u32_t tmp_wi_16 = wsig1[i+14] + w[i+9] + wsig0[i+1] + tmp_wi;
    w[i+16] = tmp_wi_16;
    wsig0[i+16] = SIG0(tmp_wi_16);
    wsig1[i+16] = SIG1(tmp_wi_16);
}
```

**This work can be done parallely with the update iterations**



# Implementation - Update - 64 Iterations ( $II = 4$ )

The calculation of **a** and **e**,  
which is actually **the critical path**  
**of each iteration.**

With the code style show at right  
side, we can only achieve  $II = 4$ .

```
/* Temporal Summation */  
u32_t t1 = h + EP1(e) + CH(e,f,g) + k[i] + tmp_w1;  
u32_t t2 = EP0(a) + MAJ(a,b,c);
```

```
/* Swap Values*/  
h = g;  
g = f;  
f = e;
```

```
e = d + t1;
```

```
d = c;  
c = b;  
b = a;
```

```
a = t1 + t2;
```

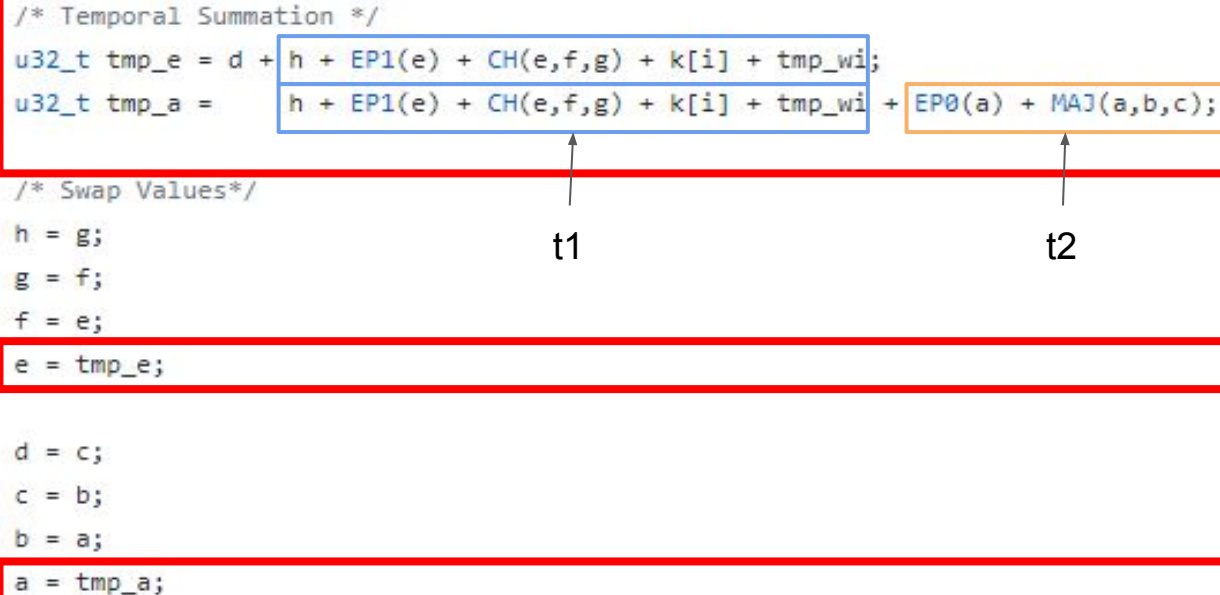
This work can be done parallelly with the calculation of  $W[16\sim63]$

# Implementation - Update - 64 Iterations ( $ll = 3$ )

```
/* Temporal Summation */
u32_t tmp_e = d + h + EP1(e) + CH(e,f,g) + k[i] + tmp_wi;
u32_t tmp_a = h + EP1(e) + CH(e,f,g) + k[i] + tmp_wi + EP0(a) + MAJ(a,b,c);

/* Swap Values*/
h = g;
g = f;
f = e;
e = tmp_e;

d = c;
c = b;
b = a;
a = tmp_a;
```



Cut the critical path by explicitly expanding the expression.

The result we get here is  $ll = 3$ .

# Implementation - Update - 64 Iterations ( $II = 2$ )

- Use double buffer
- Compute **part of i-th E and i-th A** at **(i-1)-th iteration**.
- The critical path will change to
  - Computation of SIG0
  - Computation of SIG1

```
/* Ping-Pong Calculation (Critical Path. will change to SIG0 and SIG1)*/
u32_t tmp_e, tmp_a;
if (use_0) {
    tmp_e = tmp_e_0 + EP1(e) + CH(e, f, g);
    tmp_a = tmp_a_0 + EP1(e) + CH(e, f, g) + EP0(a) + MAJ(a, b, c);
    tmp_e_1 = c + g + k_i_1 + w_i_1;
    tmp_a_1 = g + k_i_1 + w_i_1;
} else {
    tmp_e = tmp_e_1 + EP1(e) + CH(e, f, g);
    tmp_a = tmp_a_1 + EP1(e) + CH(e, f, g) + EP0(a) + MAJ(a, b, c);
    tmp_e_0 = c + g + k_i_1 + w_i_1;
    tmp_a_0 = g + k_i_1 + w_i_1;
}

/* Ping-Pong */
use_0 = (!use_0);
```

## Implementation - Update - 64 Iterations (II = 2) (cont.)

- Compute **WSIG0** and **WSIG1** later. (Delay 1 cycle)

```
/* Forward Calculation(For Reducing the Critical Path) */  
if (i < 64 - 16)  
    w[i + 16] = wsig1[i + 14] + w[i + 9] + wsig0[i + 1] + tmp_wi;  
  
/* SIG0 and SIG1 : Delay 1 Cycle (Reduce Criti. Path.) */  
if (i != 0 && i < 64 - 15) {  
    wsig0[i + 15] = SIG0(tmp_wi_back_1);  
    wsig1[i + 15] = SIG1(tmp_wi_back_1);  
}
```

Implementation - Update - 64 Iterations ( $l = 1$ )

- Change Target Frequency to 100 MHz .....

# Implementation - Update - Less Registers for W

- At **i-th** iteration, we use **W[i]** and calculate **W[i+16]**.
- However, **W[i]** **won't be used again after we get the W[i+16]**.
- So, we only have to store the **latest 16 W**.

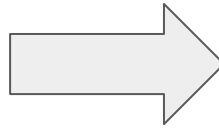
```
W[0~15] = Data( 511-i*32, 480-i*32 )
```

```
W[16~63] = SIG1( W[i- 2] ) + W[i- 7] +  
              SIG0( W[i-15] ) + W[i-16]
```

# Implementation - Update - Less Registers for W (cont.)

```
/* W-series Temporary Variables */  
u32_t w[64];  
u32_t wsig0[64];  
u32_t wsig1[64];
```

```
/* Temporary Wi */  
u32_t tmp_wi = w[i];
```

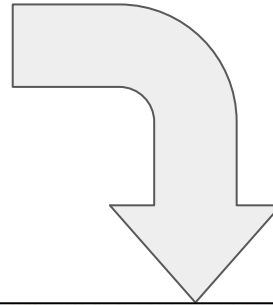


```
/* W-series Temporary Variables */  
u32_t w[16];  
u32_t wsig0[16];  
u32_t wsig1[16];
```

```
/* Temporary Wi */  
ap_uint<4> cur = i;  
u32_t tmp_wi = w[cur];
```

# Implementation - Update - Less Registers for W (cont.)

```
/* Forward Calculation(For Reducing the Critical Path) */  
if (i < 64 - 16)  
    w[i + 16] = wsig1[i + 14] + w[i + 9] + wsig0[i + 1] + tmp_wi;  
  
/* SIG0 and SIG1 : Delay 1 Cycle (Reduce Criti. Path.) */  
if (i != 0 && i < 64 - 15) {  
    wsig0[i + 15] = SIG0(tmp_wi_back_1);  
    wsig1[i + 15] = SIG1(tmp_wi_back_1);  
}
```



**Overwrite w[i]**

```
/* Forward Calculation(For Reducing the Critical Path) */  
if (i < 64 - 16)  
    w[cur] = wsig1[(ap_uint<4>)(cur + 14)] + w[(ap_uint<4>)(cur + 9)] +  
            wsig0[(ap_uint<4>)(cur + 1)] + tmp_wi;  
  
/* SIG0 and SIG1 : Delay 1 Cycle (Reduce Criti. Path.) */  
if (i != 0 && i < 64 - 15) {  
    wsig0[(ap_uint<4>)(cur - 1)] = SIG0(tmp_wi_back_1);  
    wsig1[(ap_uint<4>)(cur - 1)] = SIG1(tmp_wi_back_1);  
}
```



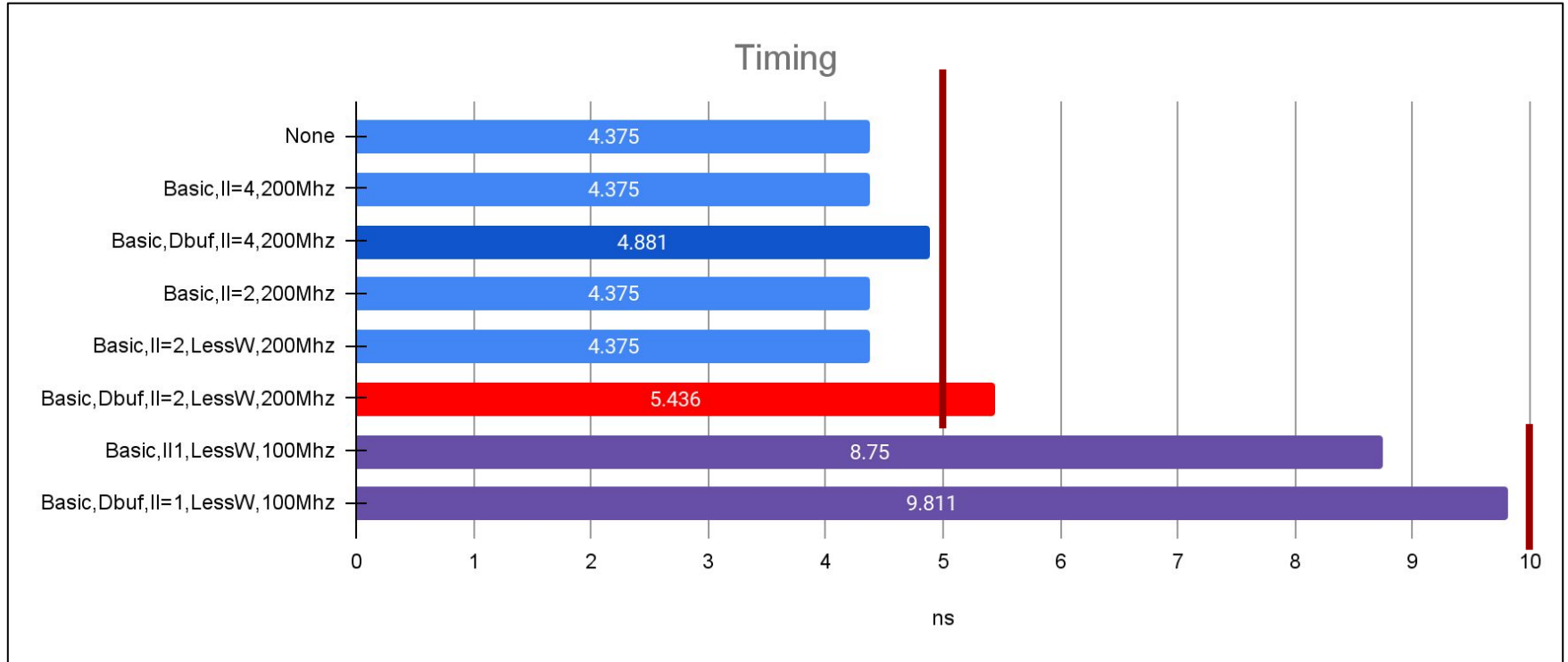
# Outline

- SHA256
- Implementation
- **Comparison**
- SILEXICA Tool
- On-Board
- Conclusion

# Comparison - Timing

Basic	Double Buffer	II = ?	Less W	Mhz = ?	Timing
					4.375
V		4		200	4.375
V	V	4		200	4.881
V		2		200	4.375
V		2	V	200	4.375
V	V	2	V	200	5.436
V		1	V	100	8.750
V	V	1	V	100	9.811

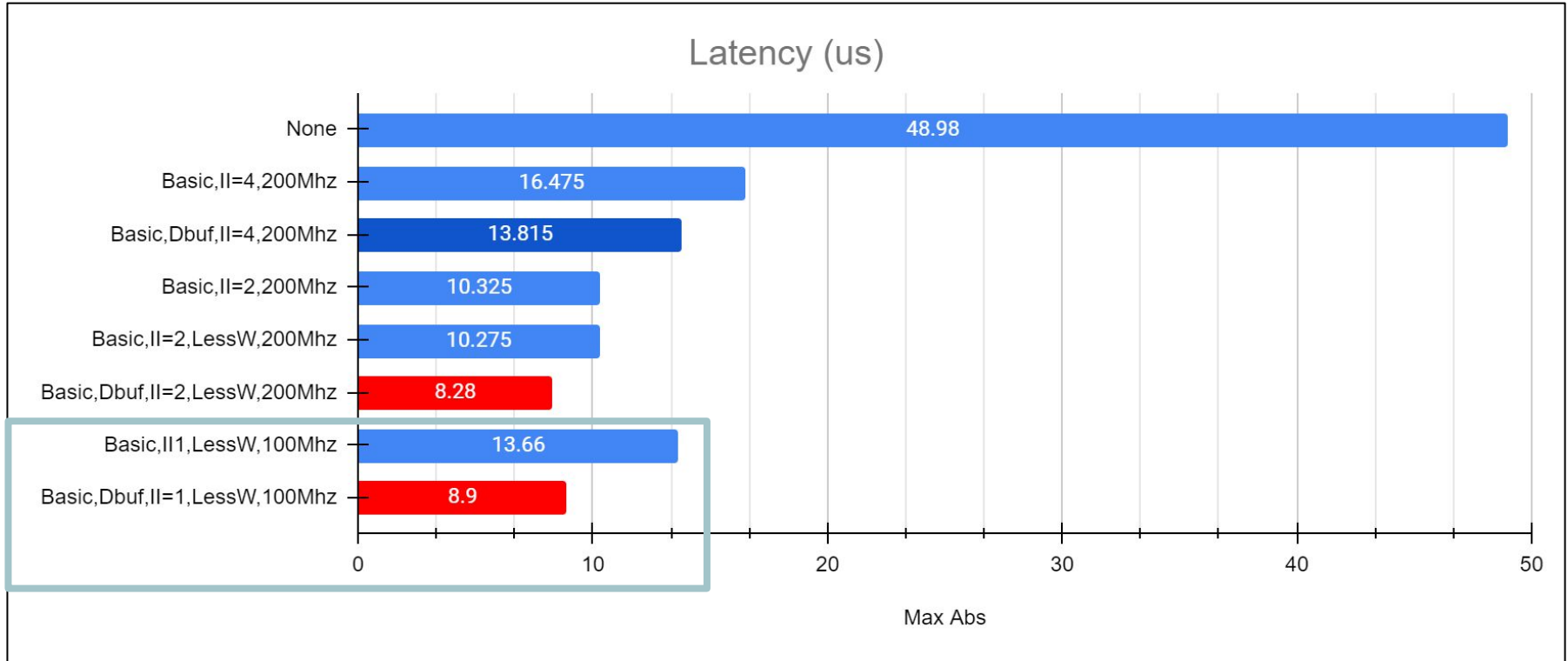
# Comparison - Timing (cont.)



# Comparison - Latency

Basic	Double Buffer	II = ?	Less W	Mhz = ?	Min Cycle	Max Cycle	Min Abs (us)	Max Abs (us)
					989	9796	4.945	48.980
V		4		200	547	3295	2.735	16.475
V	V	4		200	278	2763	1.390	13.815
V		2		200	301	2065	1.505	10.325
V		2	V	200	299	2055	1.495	10.275
V	V	2	V	200	154	1523	0.837	8.280
V		1	V	100	162	1366	1.620	13.660
V	V	1	V	100	84	890	0.840	8.900

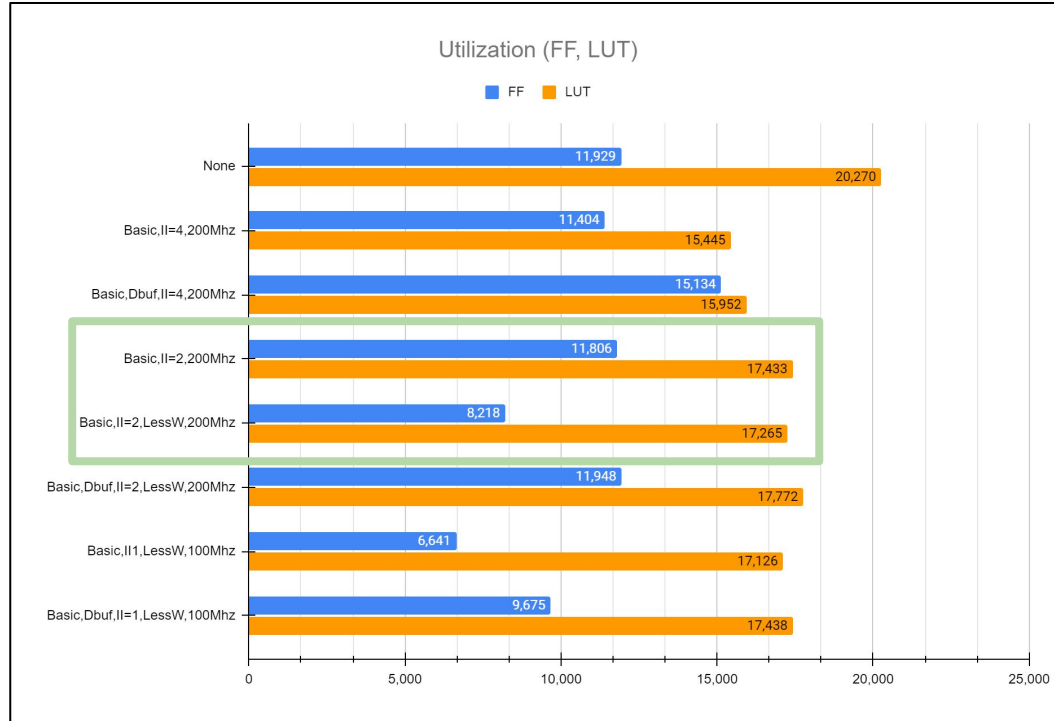
# Comparison - Latency (cont.)



# Comparison - Utilization

Basic	Double Buffer	II = ?	Less W	Mhz = ?	BRAM 18K	FF	LUT
					6	11,929	20,270
V		4		200	3	11,404	15,445
V	V	4		200	3	15,134	15,952
V		2		200	3	11,806	17,433
V		2	V	200	3	8,218	17,265
V	V	2	V	200	3	11,948	17,772
V		1	V	100	3	6,641	17,126
V	V	1	V	100	3	9,675	17,438

# Comparison - Utilization (cont.)



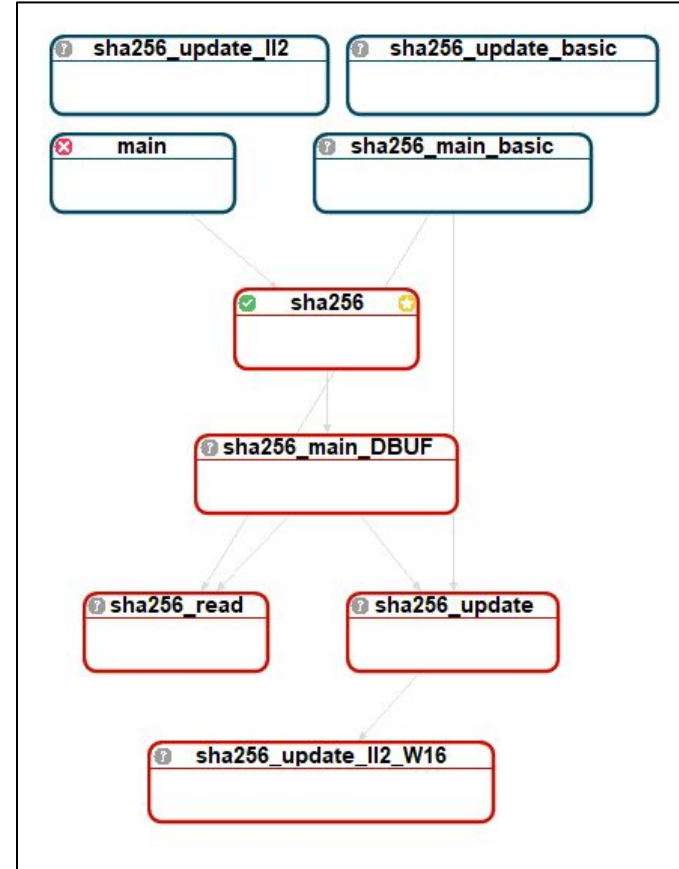
# Outline

- SHA256
- Implementation
- Comparison
- **SILEXICA Tool**
- On-Board
- Conclusion



# SLX

- You must provide main function
  - For profiling ?
- By profiling
  - find possible parallelism
- Insert Pragma for Optimization
  - Loop unrolling
  - Loop pipelining
  - Array partitioning and interface design
  - Function inlining
  - Trip-counts



# SLX (cont.)

- Data Level Parallelism
- Pipeline Level Parallelism

READ [sha256\_read.cpp:6]

Unrolling

Enable ☐

Unroll Factor 1

Pragma #pragma HLS unroll

Pipelining

Enable ☒

Initiation Interval 1

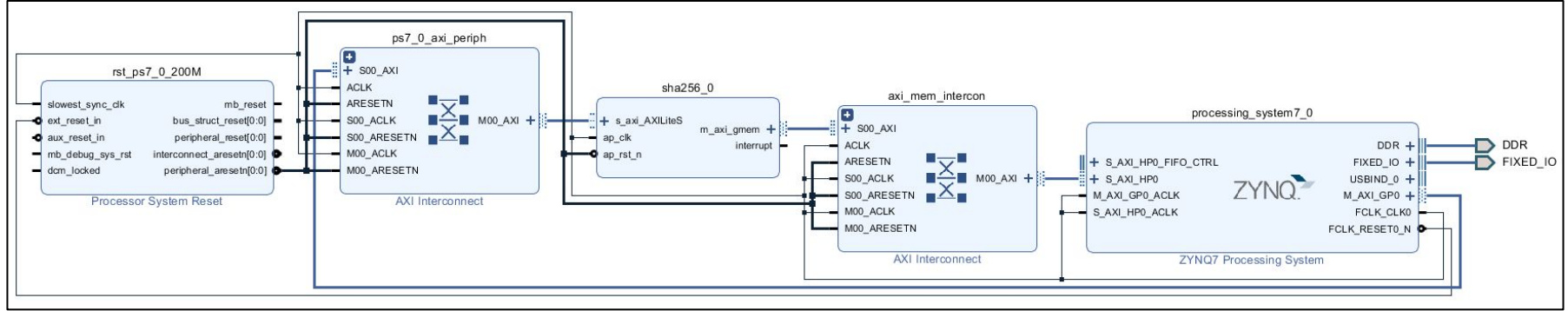
Pragma #pragma HLS pipeline

Name	Status	Location	Description	CPU Tot...	FPGA Total Cost(%)	Help
12 Loop		..\hls_lab8_sha256\src\main.cpp			0.0	
13 sha256		..\hls_lab8_sha256\src\sha256.c...			100.0	
14 sha256_main_DBUF		..\hls_lab8_sha256\src\sha256.c...			99.99	
15 Loop		..\hls_lab8_sha256\src\sha256.c...			67.67	
16 DLP	✗					
17 PARTITIONING		..\hls_lab8_sha256\src\sha256.c...	The loop carries dependencies that can be ignored.			?
18 PARTITIONING		..\hls_lab8_sha256\src\sha256.c...	Induction variable: offset			?
19 PARTITIONING		..\hls_lab8_sha256\src\sha256.c...	Unroll factor(s) 7 will be considered.			?
20 PARTITIONING		..\hls_lab8_sha256\src\sha256.c...	The loop carries dependencies.			?
21 PARTITIONING		..\hls_lab8_sha256\src\sha256.c...	Variable: data_0 [RAW]			?
22 PARTITIONING		..\hls_lab8_sha256\src\sha256.c...	Variable: data_1 [RAW]			?
23 PARTITIONING		..\hls_lab8_sha256\src\sha256.c...	Variable: use_0 [RAW]			?
24 PARTITIONING		..\hls_lab8_sha256\src\sha256.c...	Variable: hash [RAW]			?
25 PLP	✓					
26 PARTITIONING		..\hls_lab8_sha256\src\sha256.c...	Pipelining will be considered.			?
27 Loop		..\hls_lab8_sha256\src\sha256.c...			0.05	
28 sha256_read		..\hls_lab8_sha256\src\sha256_r...			3.2	
29 Loop		..\hls_lab8_sha256\src\sha256_r...			3.17	
30 DLP	✗					
31 PARTITIONING		..\hls_lab8_sha256\src\sha256_r...	The loop carries dependencies.			?
32 PARTITIONING		..\hls_lab8_sha256\src\sha256.c...	Variable: data_0 [RAW]			?
33 PARTITIONING		..\hls_lab8_sha256\src\sha256.c...	Variable: data_1 [RAW]			?
34 PARTITIONING		..\hls_lab8_sha256\src\sha256_r...	The loop carries dependencies that can be ignored.			?
35 PARTITIONING		..\hls_lab8_sha256\src\sha256_r...	Induction variable: i			?
36 PARTITIONING		..\hls_lab8_sha256\src\sha256_r...	Considered private: unnamed			?
37 PARTITIONING		..\hls_lab8_sha256\src\sha256_r...	Unroll factor(s) 64 will be considered.			?
38 PLP	✓					
39 PARTITIONING		..\hls_lab8_sha256\src\sha256_r...	Pipelining will be considered.			?
40 sha256_update		..\hls_lab8_sha256\src\sha256.c...			85.2	
41 sha256_update_IJ2...		..\hls_lab8_sha256\src\sha256.c...			85.19	
42 Loop		..\hls_lab8_sha256\src\sha256.c...			0.99	
43 Loop		..\hls_lab8_sha256\src\sha256.c...			0.99	
44 Loop		..\hls_lab8_sha256\src\sha256.c...			83.78	

# Outline

- SHA256
- Implementation
- Comparison
- SILEXICA Tool
- **On-Board**
- Conclusion

# Block Diagram & Utilization



WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP
							0	0	0.0	0	0
0.306	0.000	0.018	0.000	0.000	1.521	0	9919	10651	1.0	0	0

Report from HLS :  
LUT: 17438, FF: 9675

# Result

- I/O-Bound

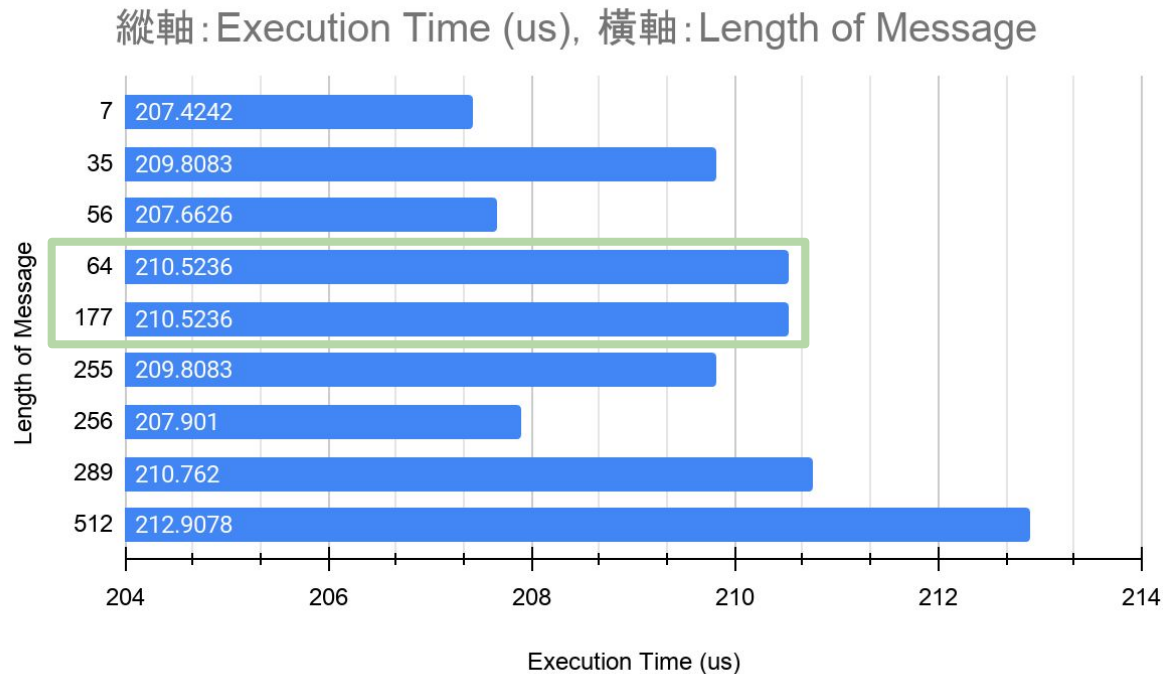
```
Kernel execution time: 0.0002079010009765625 s
RESULT: fdb005a0bb6ea5a5ae2a3ba98a90f8b24a52283d5c7264605bd82501e3a1fd28
ANSWER: fdb005a0bb6ea5a5ae2a3ba98a90f8b24a52283d5c7264605bd82501e3a1fd28
PASS : True

Kernel execution time: 0.000209808349609375 s
RESULT: c640f1d6cb4d10609c2b4125f799c608eddc03a037e213ac2c80b3bb59773bf6
ANSWER: c640f1d6cb4d10609c2b4125f799c608eddc03a037e213ac2c80b3bb59773bf6
PASS : True

Kernel execution time: 0.000209808349609375 s
RESULT: ae9b4aa2144d08da65a4878e0db091702fea3cb2d1b897e587f68f44aba841f6
ANSWER: ae9b4aa2144d08da65a4878e0db091702fea3cb2d1b897e587f68f44aba841f6
PASS : True

Kernel execution time: 0.0002120077911376953 s
RESULT: 836e3aedb84f837dc8f44ef138eae5b7febfcde1688cba6a2cf5c63d644b36e6
ANSWER: 836e3aedb84f837dc8f44ef138eae5b7febfcde1688cba6a2cf5c63d644b36e6
PASS : True

=====
ALL PASS : True
TOTAL TIME: 0.0018873214721679688 s
=====
Exit process
```



# Conclusion

- Apply optimization on SHA256 for HLS
  - Use Double Buffer and Burst Read to reduce cycle for read data
  - Use Loop unrolling for one-cycle calculation of W[0~15]
  - Use Double Buffer and Cycle-Delay for reducing the II of pipeline
- SLX is good for profiling and finding the parallism in HLS
  - But it can't do some aggressive code transforming.
  - Double Buffer and Cycle-Delay needs programmer to explicit implement the control-flow and it's hard to do it by inserting pragma only.

# Questions

- Why **unrolling** but not **pipelining** the calculation of  $W[0\sim 15]$ ?
  - Considering the throughput and area.
- To do burst read, I may use **memcpy to copy 64 byte once**, but there exists a possible risk related to the host memory. What it could be?
  - Assume Xilinx didn't provide any standard or model for host memory.