

System Verilog Simulation of SHA-256 on Memristive Crossbars (ECE 590)

Content added by Manikandeshwar Sasidhar

1. PROBLEM STATEMENT

- The main goal for the project is to study memristive technology and use it to implement SHA 256 algorithm using SystemVerilog and optimize the design to improve the timing and power.
- In order to achieve the best performance in term of delay and low power, this paper applies several variants of timing optimization technique to implement the SHA-256 algorithm. Then the performance of each approach is evaluated using simulation.

2. INTRODUCTION

Computing SHA-256 hash is the most important security and reliability aspect of the bitcoin system. Using just a CPU to calculate the SHA-256 hash can be a slow process. Since the hash calculation happens so frequently in the bitcoin mining system, the performance of the hash computation can have a big impact on the response time of the bitcoin mining. A faster SHA-256 hash computation can reduce the time required to mine new bitcoin blocks. The recent attempts to increase the speed of SHA computation have forced the SHA hardware design philosophy to move away from conventional CPU based designs to dedicated GPUs, and then to FPGAs and to ASICs fully optimized for bitcoin mining. While the CMOS circuit is approaching fundamental limits, the alternate new devices and microarchitectures are explored to address the growing need of lower power and faster speed. New nanotechnologies, such as memristors, emerge. Memristors can be used to perform stateful logic with nanowire crossbars which allow for the implementation of very large binary logic networks that can be easily reconfigured. Comparing to the CMOS technology, the memristor can hold values with the lower area and higher density, which make it a candidate for implementing cryptography algorithms. In this paper, we focus on SHA256. For designing the SHA256 architecture we have combined the memristor technology with the CMOS technology to save the die area and utilize more logic on the die.

This preliminary paper discusses several variants of memristive designs of SHA-256. To verify the design, it will have to be simulated with a Verilog model we proposed, together with the Spice simulation for the crossbar. Also, an FPGA version of SHA-256 has been developed and will be used as a reference to evaluate the performance of the memristor version.

3. BACKGROUND

3.1 FUNDAMENTAL CONCEPTS of SHA-2 FAMILY and SHA-256

SHA2 is the acronym for the Secure Hash Algorithm-2. It is a combination of cryptographic hash functions (mathematical operations used for security purposes). It was designed by NSA (National Security Agency). The integrity of the data can be determined by computing the hash from the execution of the cryptographic algorithm and comparing it with a known value of the hash function. If the comparison doesn't match, then we know that the file has been tampered with. The main advantage of the cryptographic hash function is its collision resistance. It is computed with the use of 32-bit words. The application of SHA2 includes security protocols, cryptocurrencies and cryptographic algorithms. **SHA256** is one of the hash functions in the SHA2 family. Figure 1 explains the basic idea of Hash architectures. The word Zebra is hashed to number 2,415,919,104.

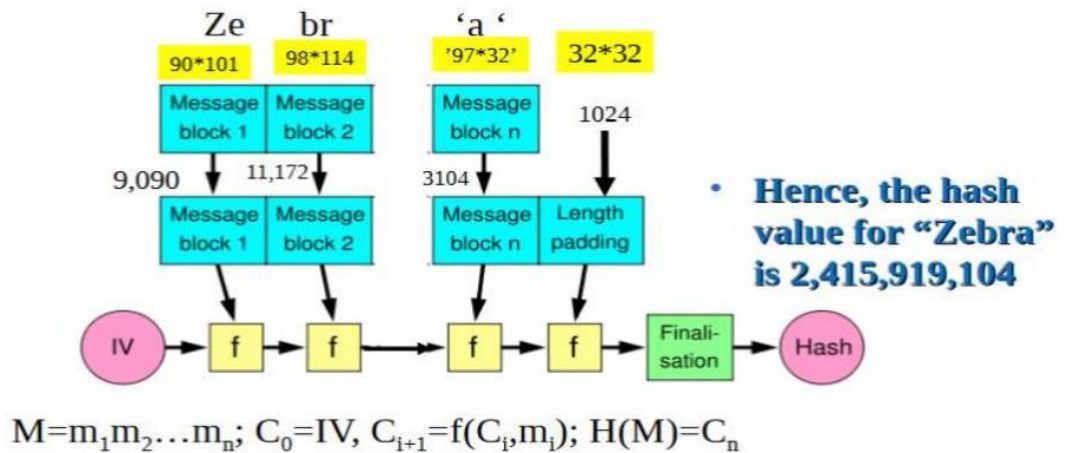


Fig. 1. Message - hash generation using SHA - 256

The hardware implementation of SHA256 is a cryptographic system that withstands any brute force attacks that occur on, to detect the secret message that is encrypted using Merkle Damgard based cryptographic functions. The two ways to get the encrypted message back is to try to get a hash which was already generated for a particular message, hence we can trace back to the actual message, the other way is that if an attacker is given a message, it should be infeasible for the attacker to manipulate the message and yet get the digest just as the original message digest. The statistics show that in order to break SHA-256, it takes an incredible amount of time that exponentially grows, the more and more the complexity to detect the message adds up. It is designed by using flags to notify changes in the state of the execution or processing. SHA-256 has two major components: the scheduler function and compression function. There is a control logic that directs the execution of the scheduler function, compression function and finally updates the digest. The scheduler function and compression function will be implemented in hardware with several memristors based crossbar arrays. The Pre-processing stage has been executed on the software side for ease of execution.

3.2 SHA256 Architecture Introduction.

Cryptographic Hash Functions

A Cryptographic Hash Function takes any string as an input to produce a fixed size output. It is a one way function and non-reversible. It is efficiently computable in a short amount of time. The SHA256 algorithm takes an input that has a length of less than 2^{64} bits. It has a block size of 512 bits, as seen on top of Figure 2. Blocks are represented as a sequence of sixteen 32-bit words. This 512 bit block enters a function called the message compression function in words of 32 bits (Wt) through a block called the message scheduler.

SHA-256 architecture is basically divided into 2 main parts:

- Compression unit
- Message Scheduler

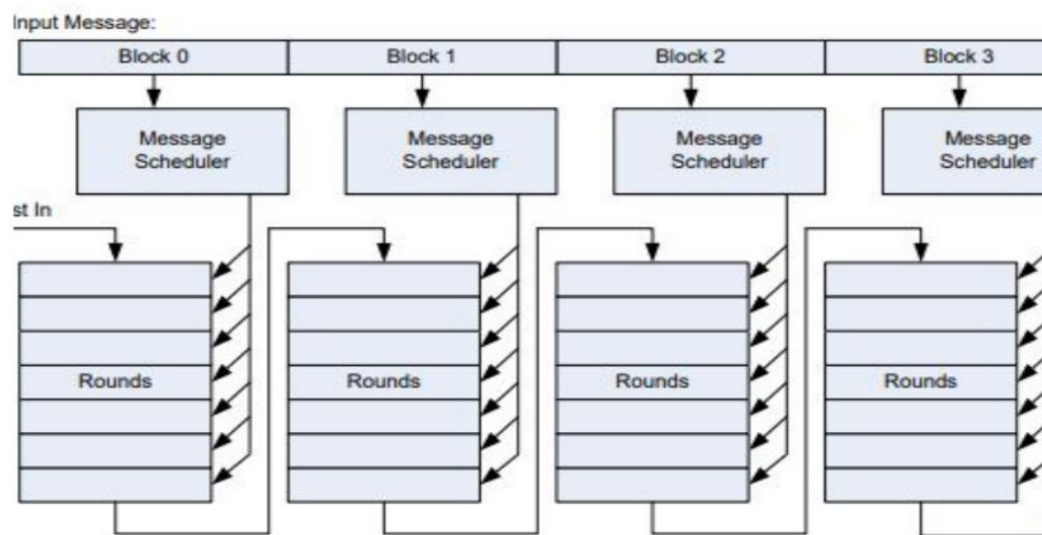


Fig. 2. Overall architecture of SHA256.

The **message scheduler** expands the 512-bit message block into 256-bit data. The operations inside the SHA256 hashing algorithm are performed on words that are 32-bit in length using eight working variables names as A, B, C, D, E, F, G and H. The word length of the SHA256 algorithm is of 32 bits. The values for these working variables are computed at every round. This process continues till 64 rounds have been completed (only 7 rounds are shown in Figure 1 for simplification). All additions in the SHA256 hashing algorithm are performed modulo 2^{32} .

An intermediate message digest that is obtained at the end of the first 64 rounds serves as the input (W) for the second message block (four consecutive message blocks are shown in Figure 1). The SHA256 hash function is built using the Davies-Meyer construction where the variable W is added to the output at the end of 64 rounds. Thus, after 64 rounds of the **message compression function** and addition of the W, the algorithm produces an intermediate message digest of 256 bits. After the entire message blocks have been hashed, a value on 256 bits is obtained that is the final message digest of the input message.

In stateful IMPLY gates, the resistance of the memristor represents the logic state; where R_{off} or High Resistance State (HRS) is considered as logic '0' and R_{on} or Low Resistance State (LRS) as logic '1'. In $a \rightarrow b$, the IMPLY yields for all combinations of the two variables a and b , the value of '1', except for $a = '1'$ and $b = '0'$. The result will be saved in b , i.e., b loses its initial value. IMPLY logic can be implemented with the memristors as it is shown in Figure 5, in which the inputs are the initial states of memristors a and b . These two are connected to a resistor R_G . The output is the final state of the memristor b after applying the fixed voltages V_{SET} and V_{COND} , respectively. The basic conditions for building the IMPLY gates are as follows: $R_{on} \ll R_G \ll R_{off}$, $V_{COND} < V_C < V_{SET}$, and $V_{SET} - V_{COND} < V_C$, where V_C is the critical voltage (i.e., memristor threshold voltage), under which the memristor holds its initial state. This determines the voltages that should be applied to the memristor. We call the memristor a source memristor and b target memristor, when executing a operation $a \rightarrow b$.

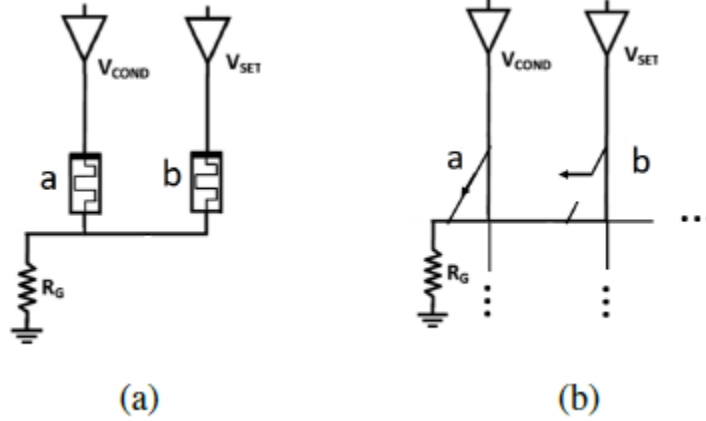


Fig. 5. (a) Circuit implementation of IMPLY logic gate (b) Correspondent switch representation with implementation in a crossbar array structure.

A cross bar with memristor at any crossing node are used as the basic architecture of memristive computing. By placing above two memristor in a crossbar, A typical Crossbar of memristors is shown in Fig.6. By applying V_{SET} , V_{COND} and ground to the different ends of the crossbar, we can select any two memristor cells as the target as the source of the ImPLY gate logic operation. We call one set of V_{SET} , V_C and ground the control signal for one gate. A series of logic operations can be performed on any memristors through a serial control signal produced by pulse a pulse generator.

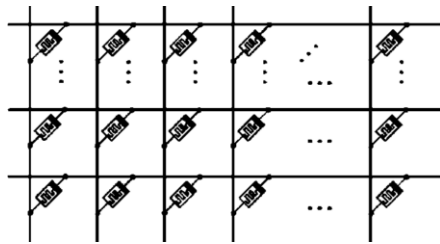


Fig. 6. A typical crossbar with memristor.

A hybrid memristive FPGA architecture was proposed by Kamela C. Rahman. A Finite State Machine with Datapath is proposed for the pulse generator. The instruction of each gate and its connectivity are stored in a memristive memory. A simple counter is used to generate the reading address for the memory. The controller reads the instruction and converts it to a set of signals that select rows and columns of the crossbar. By synthesizing the Boolean function into a chain of ImPLY gates operation and converting them to control instruction stored in the **RERAM** (memristive memory), any Boolean function can be programmed and realized with this MsFPGA architecture.

Strategy to implement a single row of memristors

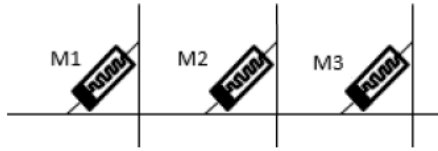


Fig. 9. A row of 3 memristors.

Given below is the design of said single row of memristors.

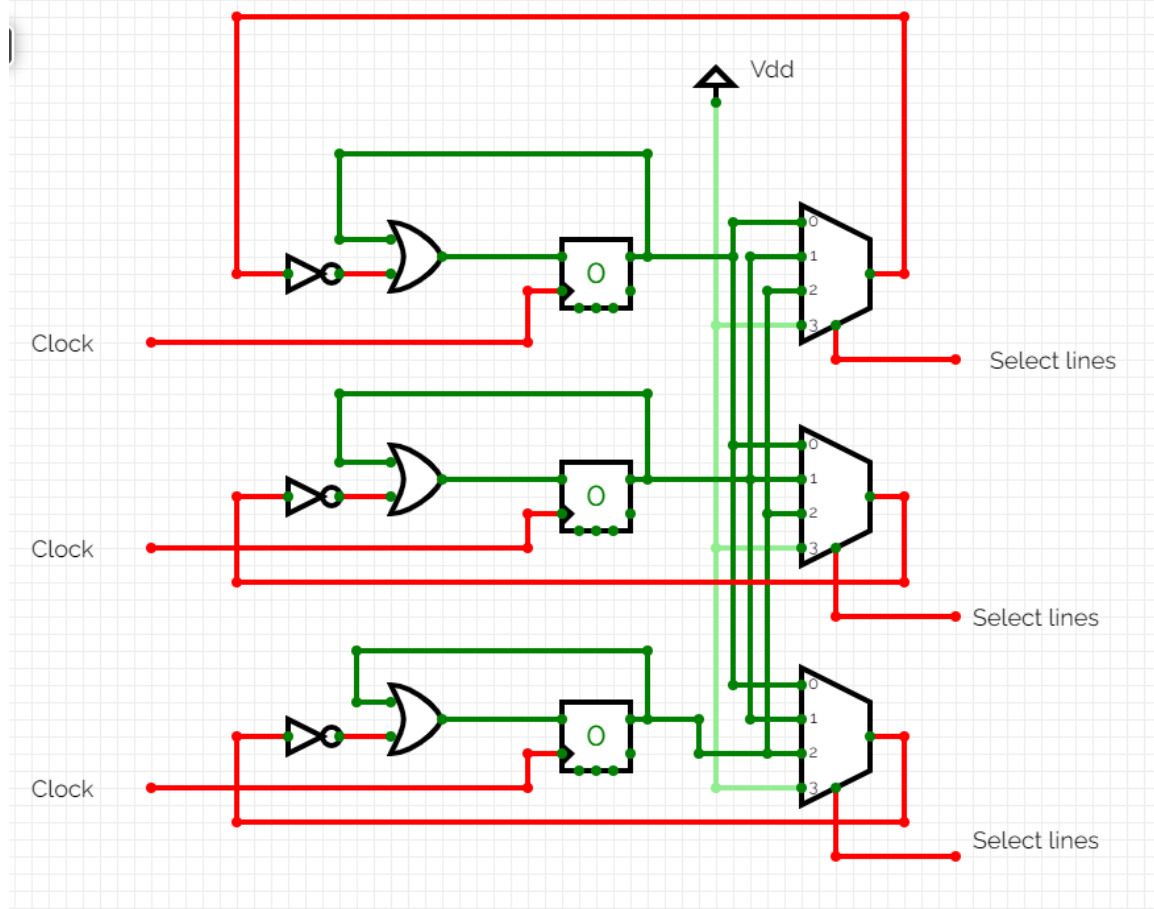


Fig. 10. Logic gate design of the aforementioned 3 memristors.

The logic diagram is equivalent to the row of 3 memristors in the figure above it. Here each of the D flipflops act as a memristor, with the MUX acting as the arbiter for selecting the appropriate memristor for IMPLY and FALSE.

MUX Signals in this case are:

- 00 (select FF 0)
- 01 (select FF 1)
- 10 (select FF 2)
- 11 (No select, retain old values)

For example, if we want to do a simple 2 input NAND gate, we do:

1. IMPLY (2,0) where sel = (11,11,01)
2. IMPLY (1,0) where sel = (11,11,00)

This is assuming that FF2 was 0 before the start of the operation. Otherwise, we must perform a FALSE operation (basically a clear) for which the signal is not shown in the diagram. For the construction of a 2D array of memristors, I simply replicated a single flipflop, N times horizontally and vertically.

SOURCE CODE:

```
module memristor #(parameter N=3)(input logic [N-1:0][N-1:0] clear, input logic [N][N][ $\lceil \log_2((N*N)+1) \rceil$ 
sel, input logic clock, output logic [N-1:0][N-1:0] Q);
```

```
logic [N-1:0][N-1:0] mem;
logic [N-1:0][N-1:0] Y;
```

SIGNALS:

clear is the signal for performing FALSE operation. Setting clear to 1 for a particular memristor sets it to 0. Therefore, by setting $clear = \{N*N\{1'b1\}\}$, the entire crossbar is erased. The value of the clear array is independent of the values of the select lines.

Mem is the 2D array of registers representing the crossbar. Y is the matrix representing the output of each of the flipflops. Q is not exactly required, but I assume that the crossbar is connected to a bus on the output, and Q is the signal that feeds into it.

To perform this in SystemVerilog,


```
task imply(input int a,b); //A and B are the memristor numbers in the array.
clear=0;foreach(sel[i,j]) sel[i][j]=N*N;
sel[a/N][a%N]=b;
@(negedge clock); endtask
```

For example if we have a 8x8 crossbar, performing imply(8,0) does an imply operation of the first 2 memristors in the first column.

1.1. Main logic blocks of SHA-256 architecture.

We have to design using memristors the following logic blocks.

In this paper, we will use the “*Imply Sequence Diagram*” (ISD) notation for the application of analyzing and synthesizing the memristor circuits. In this notation, horizontal lines represent physical memristors,

while this symbol  represents a pulse applied to it. The top side of this symbol is the negated input. The left side is the non-negated input and the value of memristor before the pulse. The right side is the value of the memristor after the pulse. A 0 in the square indicates an additional pulse required to reset the input state of the memristor to 0. Horizontal lines are memristor, vertical dash line are time.

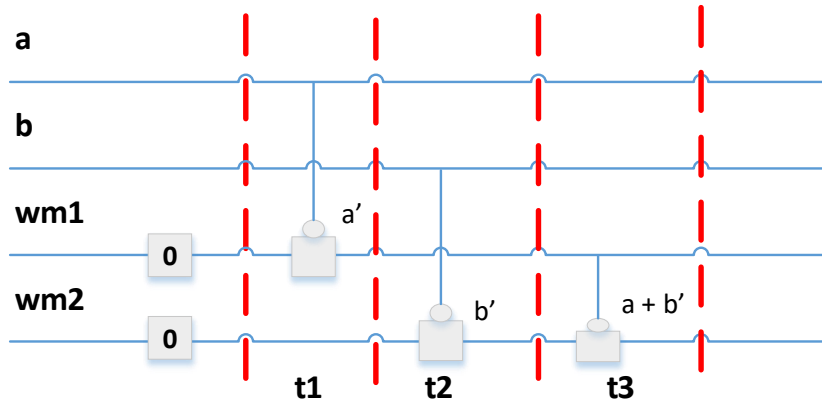


Fig.11 one pulse corresponds to one imply gate in that given pulse period.

Before t1, wm1 and wm2 are reset to 0. At t1, the wm1 become a', because $wm1 = a \rightarrow wm1 = 0 + a'$.

Basic Logic Gates in ISD Notation

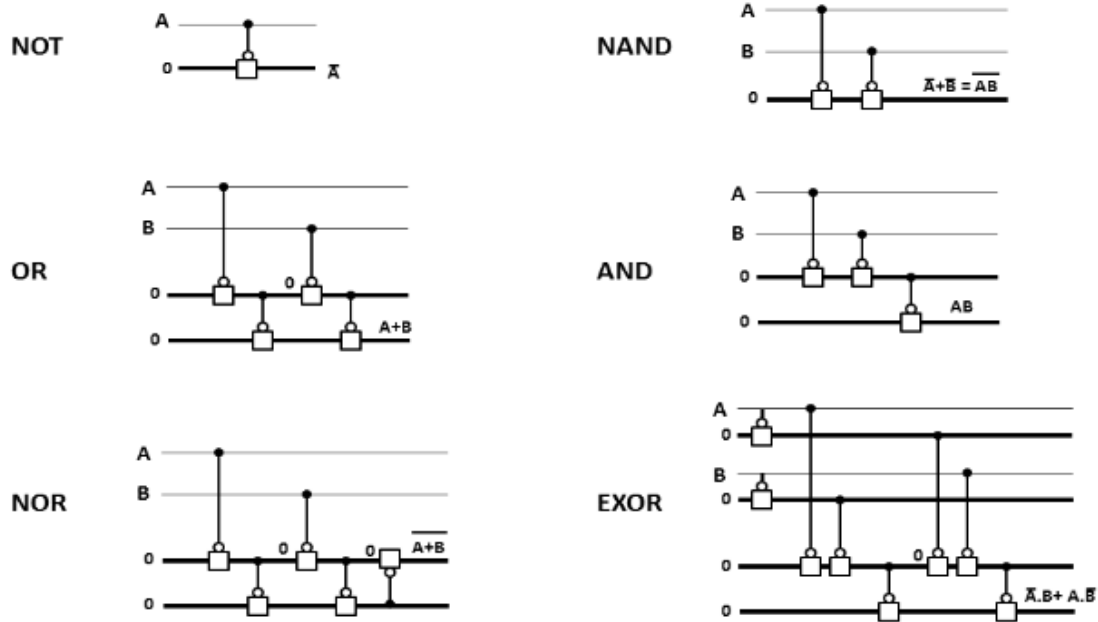


Fig. 12. ISD notation of the basic gates

Implementation of logic operation with Imply gates in IDS forms (Imply Sequence Diagram):

Note: Some of the designs destroy the input data. Just keep it in mind when design upper-level block, after the operation the data in input line is corrupted. For example in the full adder, after the operation original input line A and B and Cin are modified, so the input data is destroyed. When you use this kind of operation, you must make sure the operations in the future will not need those data anymore.

4. Implementation of SHA with memristor.

Strategy to implement the SHA 256 algorithm in memristor crossbar

We implemented a whole SHA 256 algorithm using gates built in memristive crossbar. Our design approach involved replacing every basic logic block in the original SHA256 architecture with an implementation in memristor. Then we used a bottom-up approach of building SHA 256 architecture using these memristive blocks.

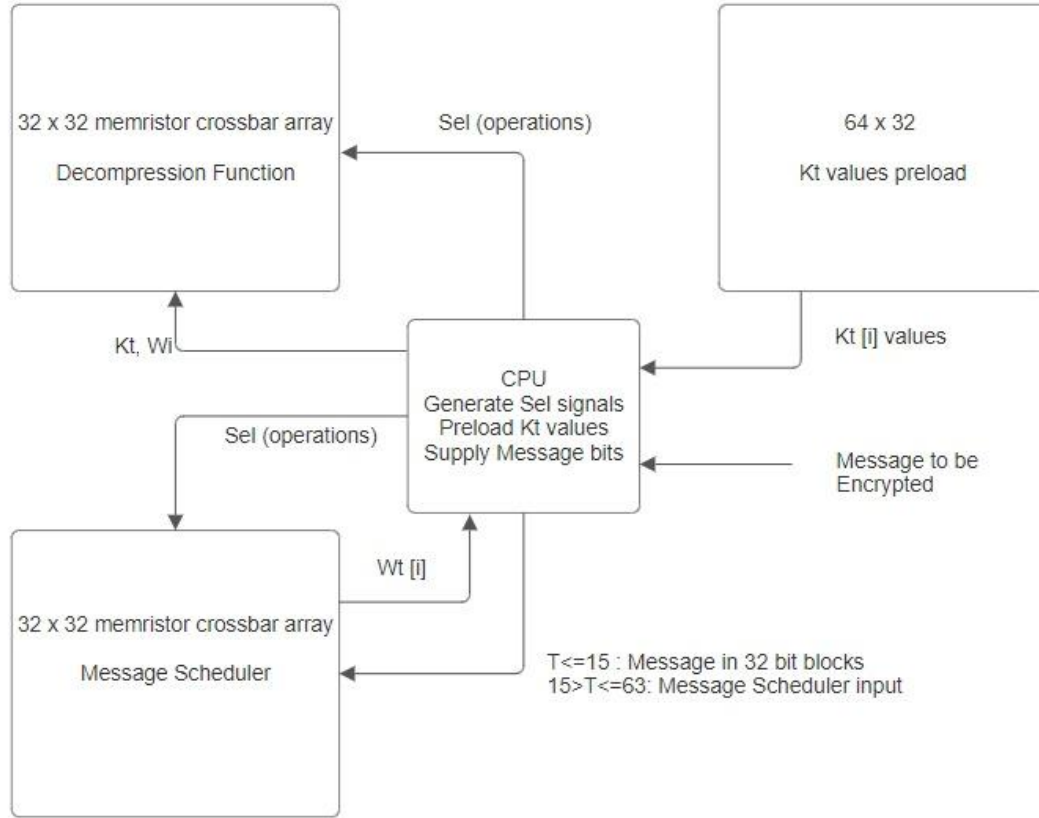


Fig. 13. High level block diagram of the SHA-256 implementation.

4.1. Starting from the bottom, we implement each single bit logic block with a series of Imply gates in the single crossbar. We first identify all the operations we need in Fig. 9 and then generate a control pulse for each block.

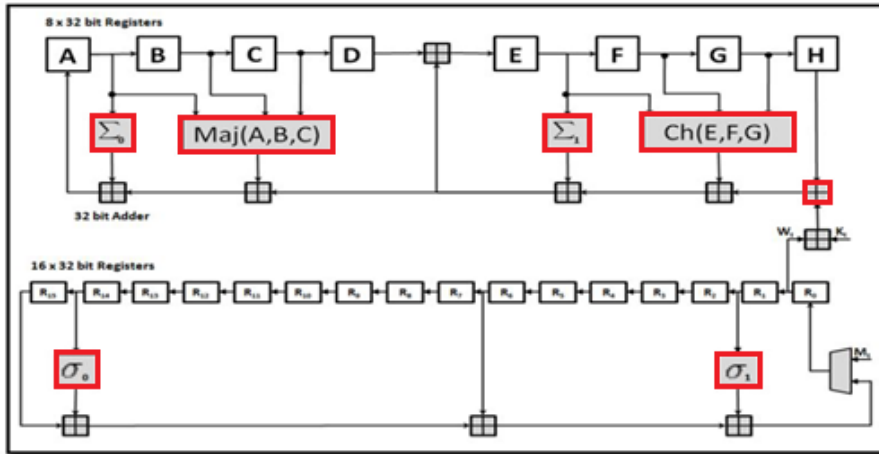


Fig. 14. Connection between the compression unit and message scheduler

The red outlined blocks of the compression unit and message scheduler are all the operations that we need to perform. These operations will need to be performed using IMPLY and FALSE operations that we just talked about. The ISD notation for these operations is discussed soon below.

Main logic blocks of SHA-256 architecture.

1. Modulo Full Adder.
2. CH operation: The SHA256 formal specification defines the XOR gate, but it can be replaced with OR: $Ch(a, b, c) = a.b + !a.c$. (same as mux).
3. Maj operation: The SHA256 formal specification defines the XOR gate, but it can be replaced with OR: $Maj(a, b, c) = ab + bc + ac$.
4. Σ_0 operation: $\Sigma_0(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$. $ROTR^n$ means Bitwise Rotate Right by n bits. A combinational implementation of rotation operation is using concatenation. $\Sigma_0 = (\{x[1:0], x[31:2]\} \oplus \{x[12:0], x[31:13]\} \oplus \{x[21:0], x[31:22]\})$.
5. Σ_1 operation: $\Sigma_1(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$. $\Sigma_1 = (\{x[5:0], x[31:6]\} \oplus \{x[10:0], x[31:11]\} \oplus \{x[24:0], x[31:25]\})$.
6. σ_0 operation: $\sigma_0(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$. SHR^n is Bitwise Shift Right n bit. A combinational implementation of shift operation is using concatenation and a temporary register with 0. $temp = 32'h0$, $\sigma_0 = (\{x[6:0], x[31:7]\} \oplus \{x[17:0], x[31:18]\} \oplus \{temp[2:0], x[31:3]\})$.
7. σ_1 operation: $\sigma_1(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$. $temp = 32'h0$, $\sigma_1 = (\{x[16:0], x[31:17]\} \oplus \{x[18:0], x[31:19]\} \oplus \{temp[9:0], x[31:10]\})$.

Full adder

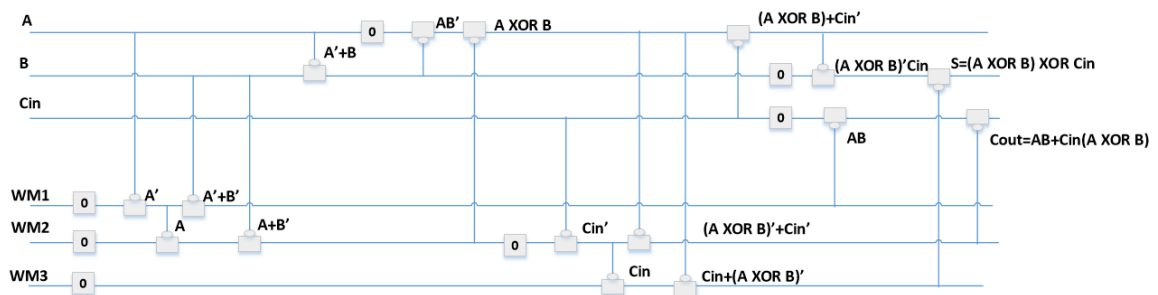
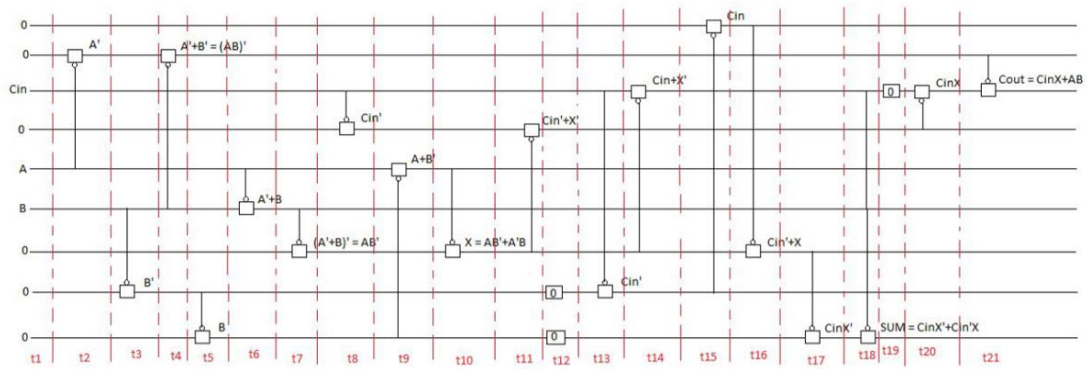


Fig. 15. ISD notation of a single bit full adder

CH using imply gates

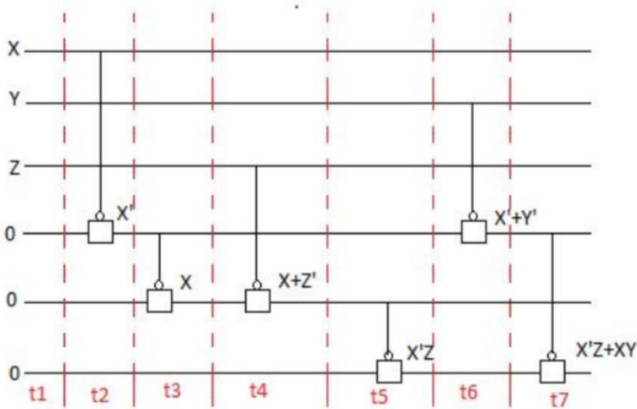


Fig. 16. ISD notation of the choose function

Majority using imply gates

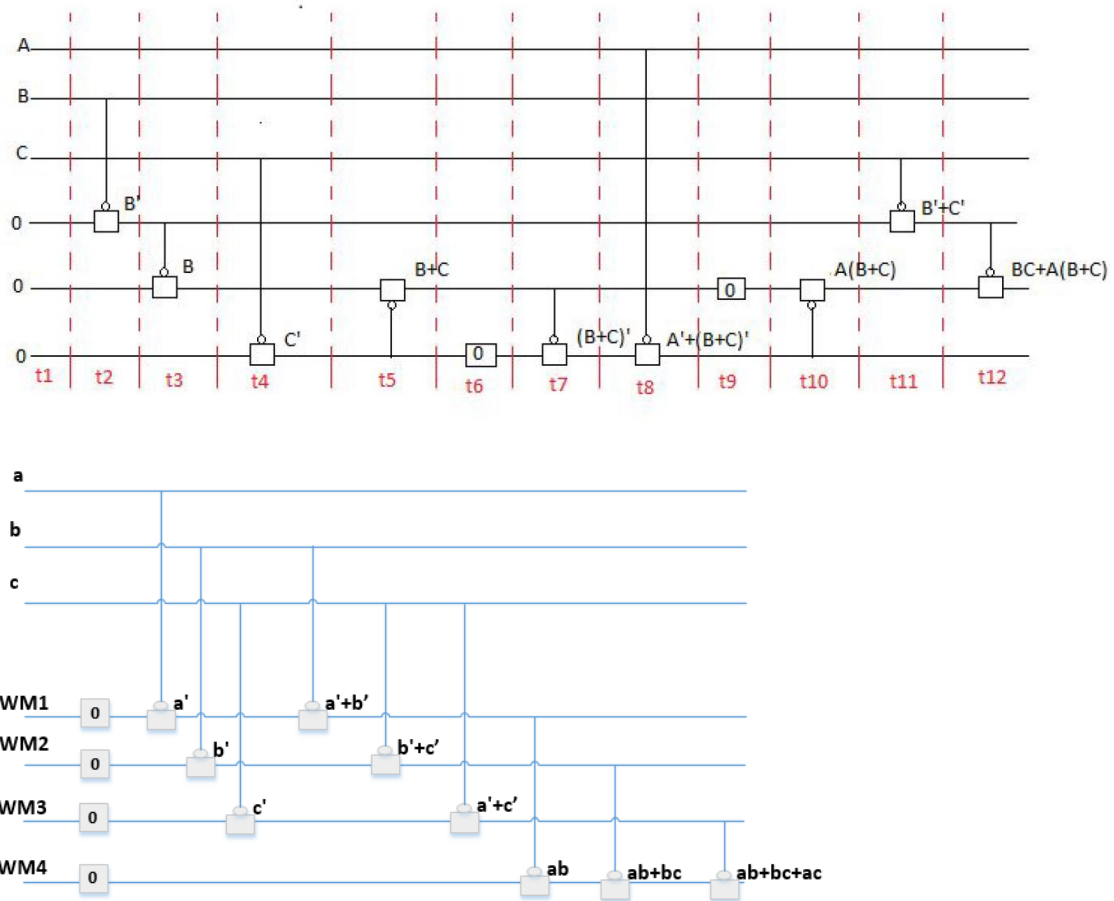


Fig. 17. ISD notation of the majority function

$$A \oplus B \oplus C = a'b'c + a'bc' + ab'c' + abc$$

3 input EX-OR using imply gates

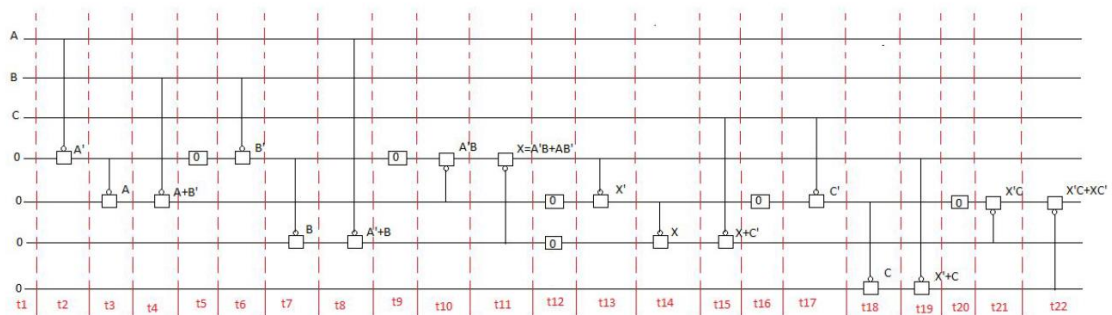
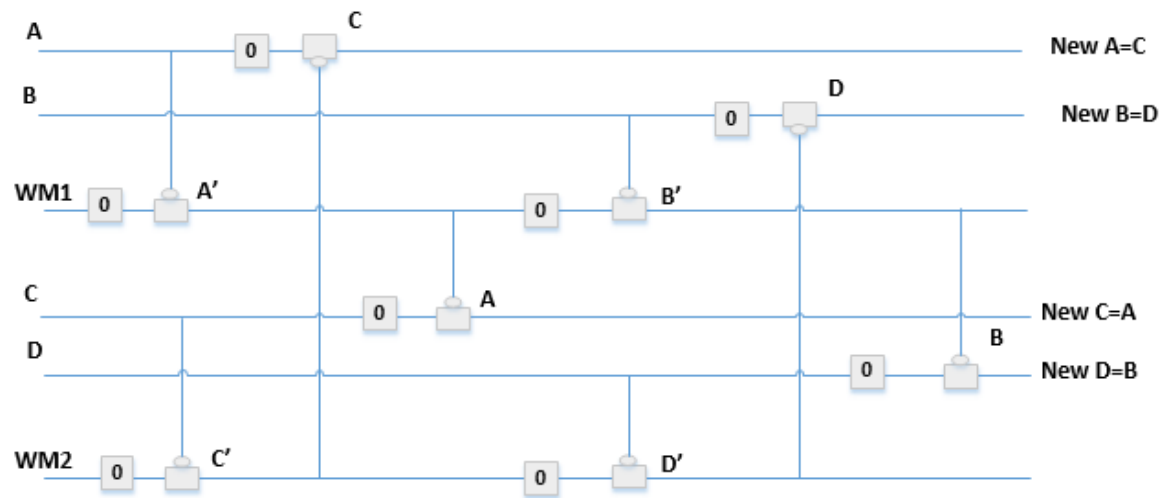


Fig. 18. ISD notation of a 3 input XOR

1. rotate 2 bit: New A = C, New B = D, New C = A, New D = B



2. rotate 3 bit: New A = B, New B = C, New C = D, New D = A

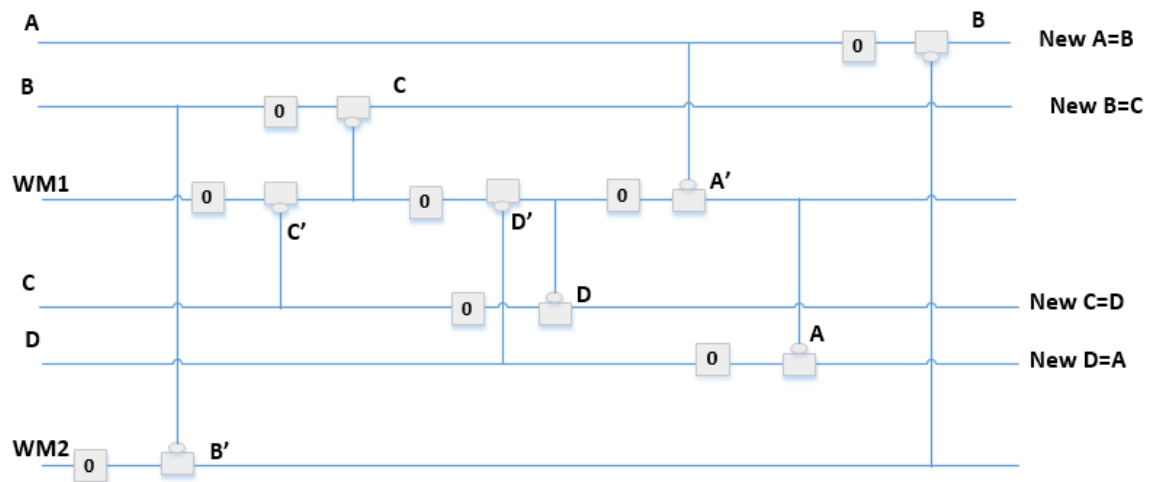


Fig. 19. ISD notation of a 2 and 3 bit cyclic shift (refer further below for more shifts)

4.2. Since each register in the compressor is 32 bits, we then extend the single bit operation to 32-bit operation and decide how to execute the multiple-bit operation. In some operation, all the bits can execute simultaneously, but some have a dependency between each bit.

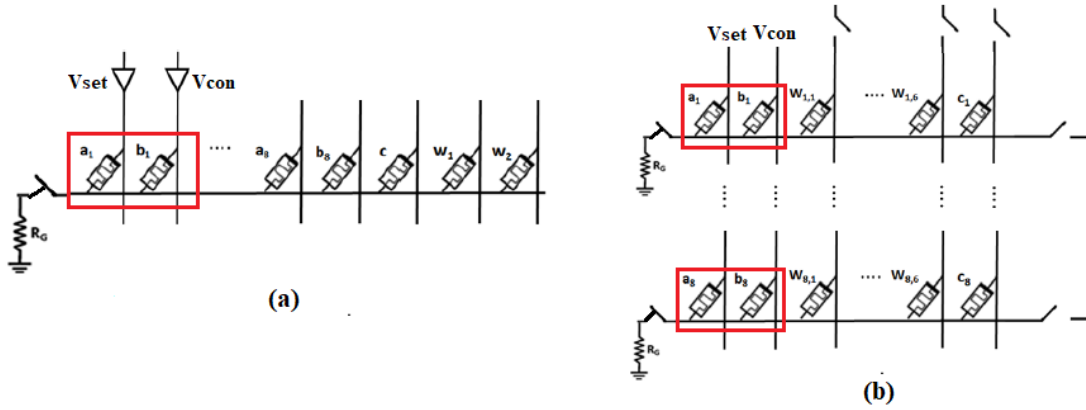


Fig 20. (a) shows ImPLY gate in only one line (b) shows multiple rows can be executed in parallel if each row is executing the same instruction.

The basic algorithm for executing a row or column wise instruction is to manipulate the select lines. By this example shown below, the entire row or column is able to perform an IMPLY parallelly, thereby saving clock cycles. The column numbers are represented by a and b as discussed earlier.

```
for(int i=0;i<N;i++) begin
    sel[i][a]=b+i*N;
end
```

We review all the single bit blocks to see if they can be run in parallel.

1. Ch (a,b,c) and majority (a,b,c) are two logic block can be parallelized. For bitwise Ch and maj operation, each bit a1~a32 and b1~b32 can be executed simultaneously with the same imply instruction. To perform this we can basically set the select signals of each row with respect to the other memristors of the same row. Even better actually, because the value of the select signals of each row will simply be a,b,c + i*N where a,b and c are column numbers, i is the row number and N is the total number of rows.

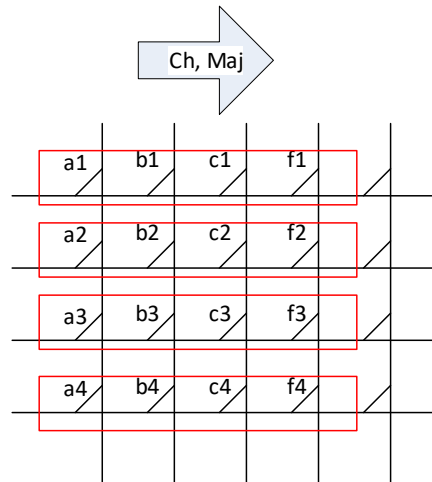


Fig. 21. Red rectangle covers the input and the output memristor for Ch and Maj operation, the Working Memristor might be needed for these operations, but they are not shown in the figure.

If we look at the code for this,

```
y_false(wm1);y_false(wm2);y_false(out);
```

```

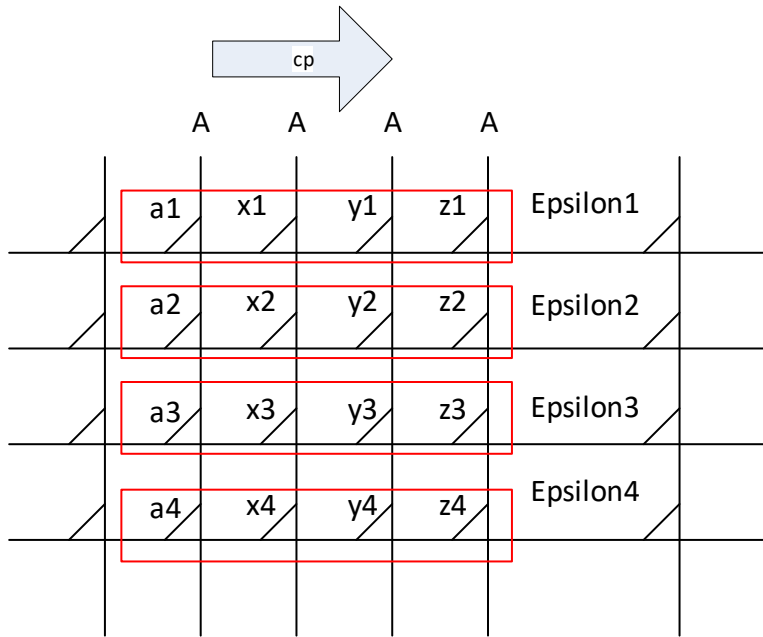
x_imply(wm1,in1);
x_imply(wm2,w1);
x_imply(wm2,in3);
x_imply(out,w2);
x_imply(w1,in2);
x_imply(out,w1);

```

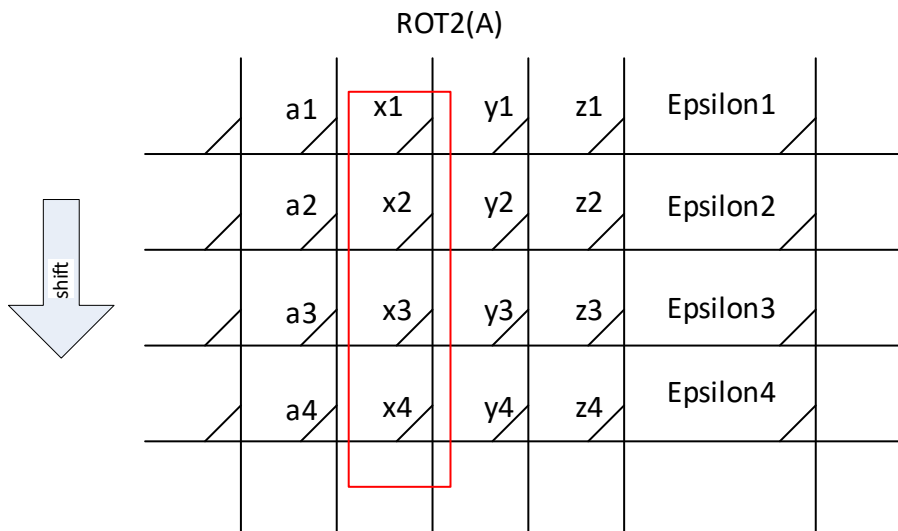
where the x and y suffix basically refer to a whole row or column respectively.

2. Σ and σ are the operations require both vertical and horizontal computation in the crossbar. The following shows an example of $\Sigma 0$.

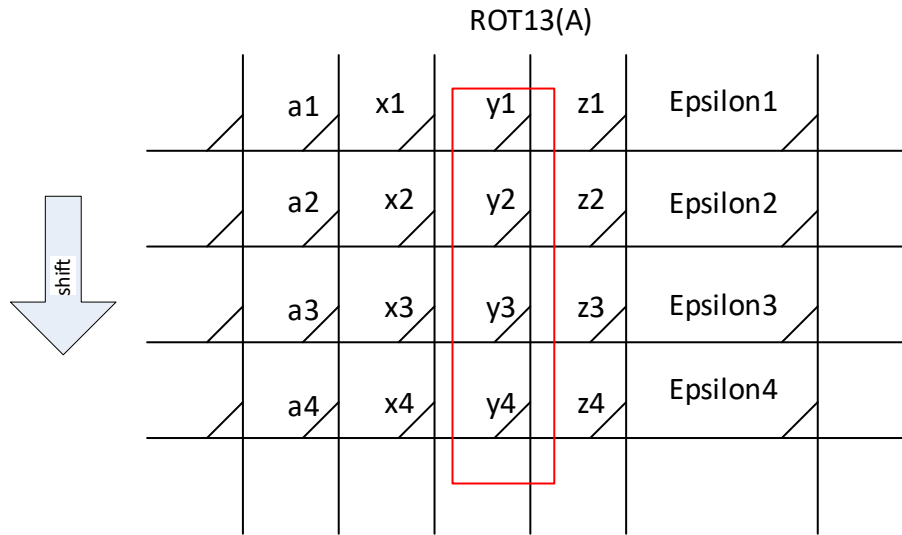
a) Copy the input A to first three more memristor cells in each line.



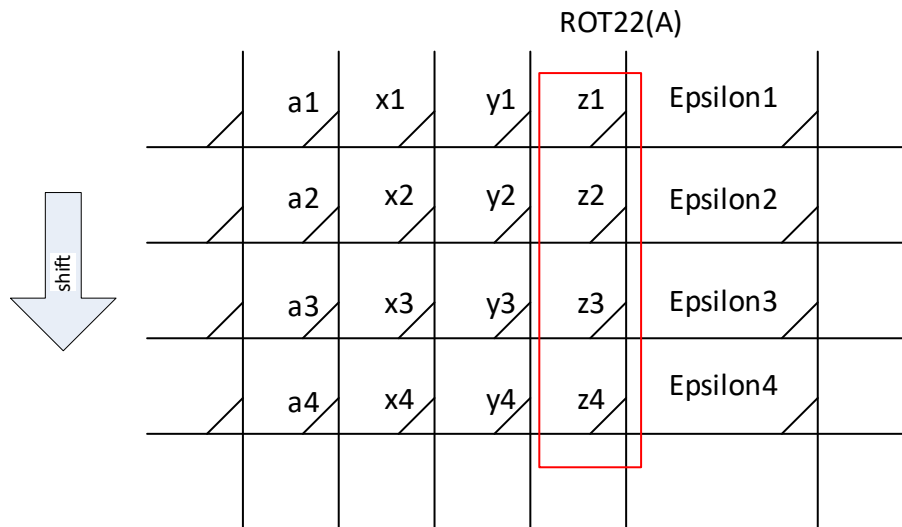
(b) Perform combinational 2 bit rotation on the first copy of A.



(b) Perform combinational 13-bit rotation on the second copy of A.



(c) Perform combinational 22-bit rotation on the third copy of A.



(d) Perform xor on the three rotation result x, y, and z in parallel.

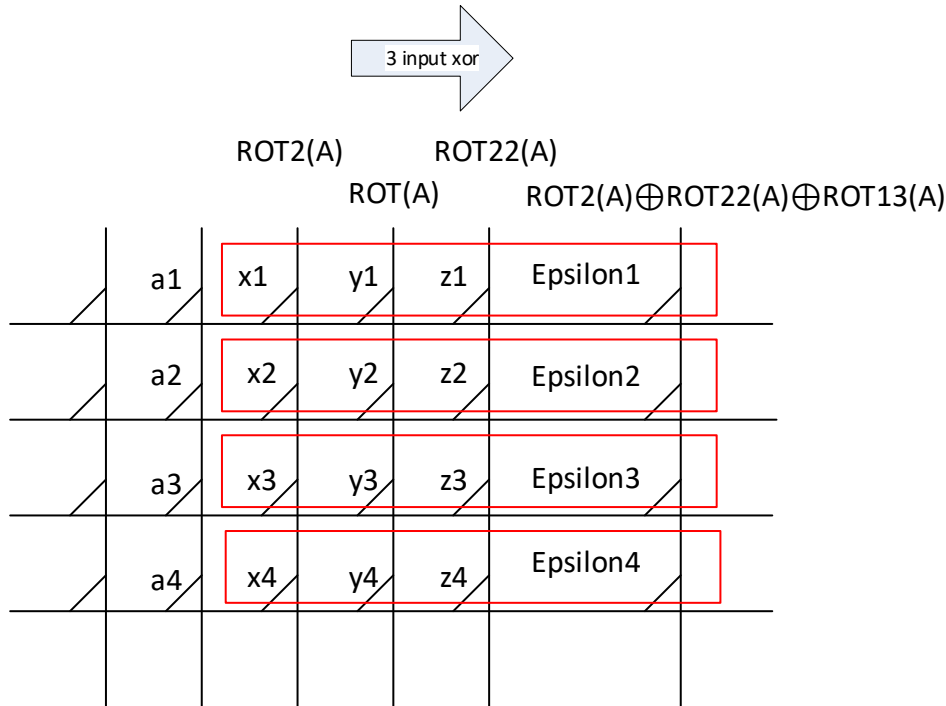


Fig. 22. Shows the layout operations to obtain E

NOTE: to perform a copy operation, we IMPLY the value to be copied into a different row/column/memristor and IMPLY it again to the required memristor. For example, $A = B$, we do $\text{IMPLY}(C, A) > \text{FALSE}(B) > \text{IMPLY}(B, C)$. We CAN NOT directly copy values between memristors.

ROTATION instructions:

```
clear=0;foreach(sel[i,j]) sel[i][j]=N*N;y_false(b);
for(int i=N-1;i>=0;i--) begin
    if(i-z>=0) sel[i-z][b]=a+i*N;
    else sel[N+i-z][b]=a+i*N;
end
```

We basically perform $\text{IMPLY}(b+z, a)$ in a loop where b is the working memristor, a is the column to be rotated and z is the number of rotations.

3. 32-bit Adder

For our basic SHA256 design, the longest data path, or critical path, is present in the calculation of the variable A for each round. This involves a total of 7 operands, H, Wt, Kt, $\square \Sigma 0$, $\square \Sigma 1$, Ch(E, F, G, and Maj(A, B, C)), and performing seven mod 2^{32} additions. This means we see the most delay during this process. So, the first optimization we employ is a method of using Carry Save Adders (CSA) which minimizes the delay caused by the carry propagation in normal adders. CSA's take in three inputs and separate the sum and carry paths as the two outputs.

A Carry Save Adder is essentially just a full adder with repurposed inputs and outputs as shown in the Figure below. For a series of adders, instead of the carry bit being propagated through to the next adder, the carry operation is saved to the very end.

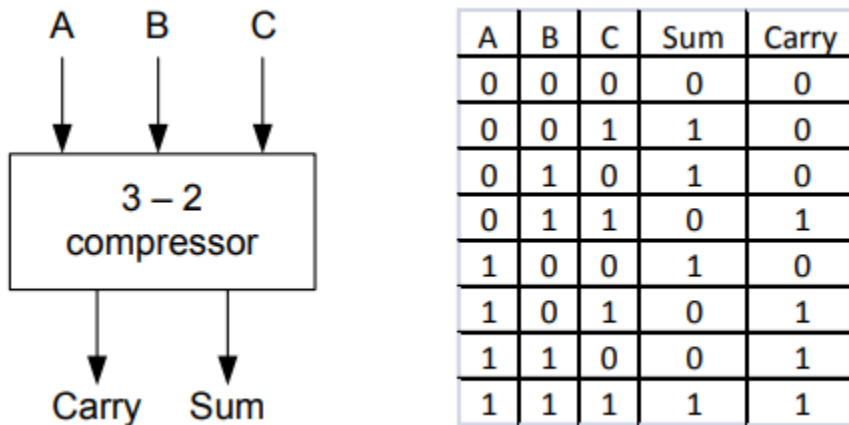


Fig. 23. Carry-Save Adders approach also called 3:2 compressors

Example

	2^4	2^3	2^2	2^1	2^0
x	0	1	0	1	0
y	1	1	0	1	1
z	1	0	1	1	1
<hr/>					
s	0	0	1	1	0
c	1	1	0	1	1

$$x+y+z = s + c$$

An example shows how a carry save adder are used to calculate a sum of three inputs. The sum and carry can be calculated in separate and then add together at the very end. When calculating multiple operands addition, CSA tree is the most effective method. A 7-input addition is implemented with a CSA tree shown in the Figure, for each CSA the critical delay is always 1 full adder. At the end, one carry propagation adder is needed.

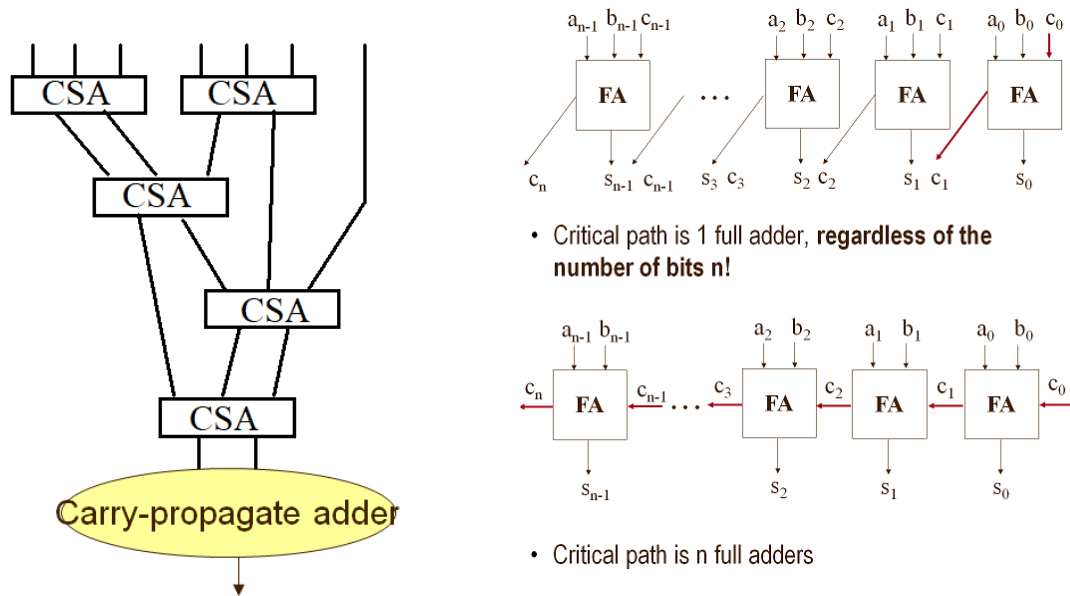


Fig. 24 shows the CSA component of a more efficient adder

A 4-bit CSA + CPS example:

- Carry save adder can be executed parallelly to calculate the sum and carry separately.
- A simple carry ripple adder is implemented by executing one full adder a time and copy the carry to next line. Then a second full adder is executed.
- $T = T_{fa} + NT_{fa} + 2NT_{cp}$

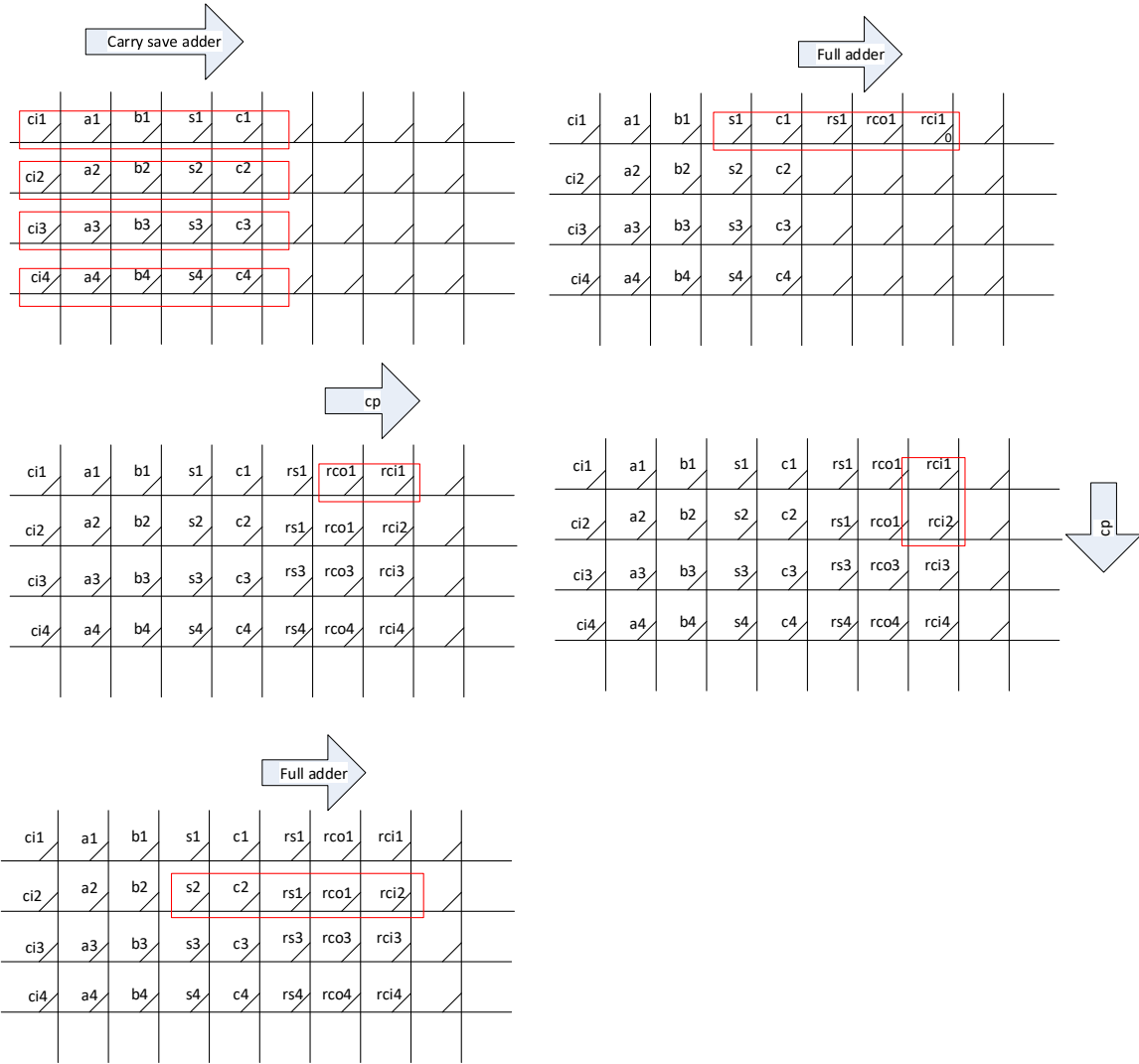


Fig. 26. shows an example of a 4-bit addition using CSAs

After deriving the delay of each block of multiple bit operations, we can build up the whole SHA256 architecture from these blocks and perform timing evaluation and optimization. The optimization strategy we consider includes “one-clock”, unrolling, parallel, pipeline, etc. After the logic block has been finalized, we can estimate the time of each block, by assigning the time delay of each logic block, we can re-time the logic-block order to optimize timing. The final sample output without timing constraints is as shown:

```

00010110001111011000110100111100
00010110000111011010110100111110
00010011110111011010111010010010
00010101110111011011110001010010
00010110010110110111000100111010
00010011011110110110101010011000
00010111110110110100100000010010
00010111101110110101000000010100
00010000101110110101101011110100
00010000111110110111001011110000
00010110010110110101000100111010
00010100001111011001111101111100
00010110110110110111100000110010
00010111010111011001010100011010
00010010011110110111000010111000
00010001010111011000011111011010
0001011111110110111100000010000
00010010000111011011010110111110
00010001001111011001011111011100
00010011100010111111111110010110
00010011010111101001011110011011
00010010011111101001110010111001
00010101100111101011010101010111
00010101100111100100000101000111
00010010001111100100000110101101
00010010000111101010010110111111
00010011000111100110101110011111
00010101110111101000110001010011
00010001100111101000111111010111
00010010010111101011110110111011
00010011010111100101101110011011
00010111011000101001110000011001

```

The first 8 columns from the right represent A – H after a single round of compression. The next 4 columns from the right represent the previous rounds ch, maj and E functions. The 2 columns after that represent the previous kt and W injected by the CPU. Refer to the Git repo for the code and it's working.

Compressor function block design

The SHA-256 compression function operates on a 512-bit message block and a 256-bit intermediate hash value. It encrypts the intermediate hash value using the message block as the key. The SHA-256 message compression is shown in Figure 5 below where the symbol “*cross in square*” denotes a mod 232 adder. There are five adders in this architecture.

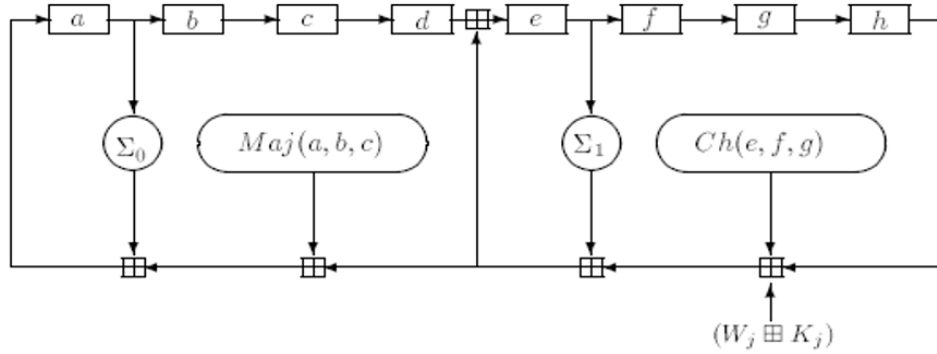


Fig. 27. the j^{th} internal step of the SHA256 compression function. Observe that in this diagram the order of adders is shown in the way to simplify flow in the figure. In reality, the adders can be used to create various trees, depending on the chip layout and delays of blocks that precede them.

SHA-256 ARCHITECTURE OPTIMIZATIONS BASED ON ADDERS

The first step in designing and implementing optimized hardware for the SHA256 architecture is identifying the parts of the design where improvement is needed the most and identifying what the objective for your improved design is. In our case, we want to employ one or more methods to reduce the delay of the core algorithm and increase the speed of the main calculations, i.e. increase the throughput of the design.

The number of levels in the CSA tree determines the basic cycle time of the addition process. In our case, our CSA tree from Figure 10 has a total of 7 operands and 6 levels of addition including the final propagation phase CPA. By minimizing the tree the designer can obtain a faster circuit by organizing the adder tree with the goal of reducing the operation time.

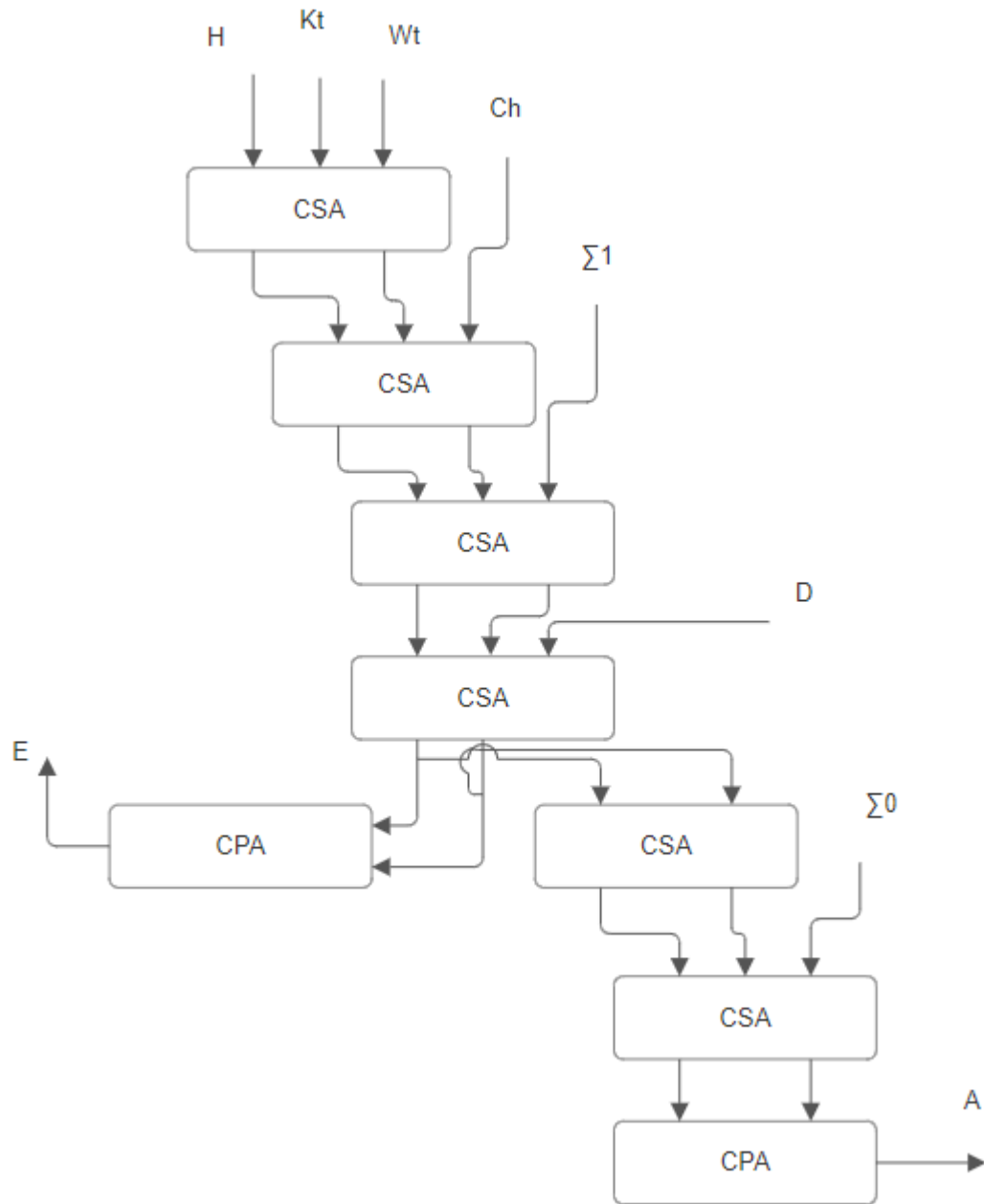


Fig. 28. Shows the flow and timing diagram of the compression unit.

Then we follow the tree and use a timing block diagram that is similar to ISD to demonstrate the timing order of each block in series. We only show the working memristor that will be used to store the result, some of the block might need more working memristors for temporary values.

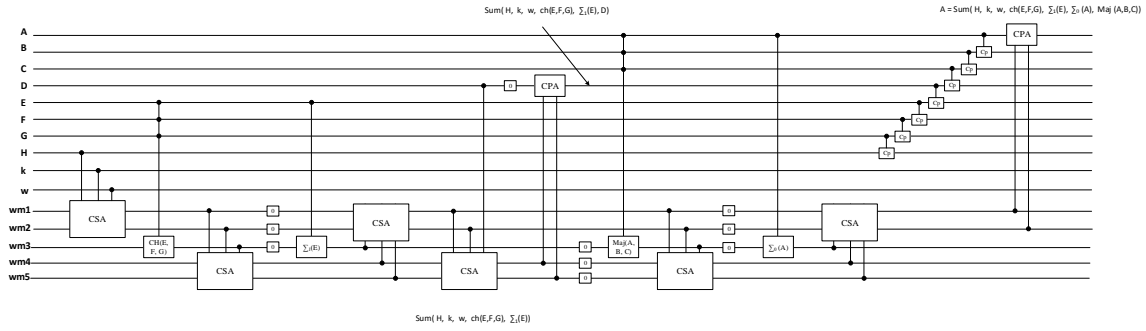


Fig. 29. Above timing block diagram showing an all-serial implementation of the compression function.

MESSAGE SCHEDULER

Message scheduler in SHA-256 creates a message schedule from the padded message which is later used to iteratively generate a series of hash values. The SHA-256 algorithm uses a message schedule of sixty-four 32-bit words. After preprocessing is completed, each message block (512 bits long), M_1, M_2, \dots, M_n , is processed in order. (Where M represents blocks generated from input message). Each message block has 512 bits represented as sixteen 32-bit words. In our design, we use 16 registers to store these values. Look to Figure 3 to find these 16 registers denoted by R_0 to R_{15} from right to left on top. As we see, they are connected to create kind of a cyclic shift register of registers with serial input from M on the bottom right and a serial output w_t on top right. The output of the adder on top right goes to the Compression part. For the first 16 cycles, $w_t = m_t$ (m_t represents 32-bit words from message block (M)). The multiplexer, shown at bottom right of Figure 3, is used to either select word directly from the message block (for the first 16 rounds) or to make computations inside the scheduler (for the last 48 rounds). The SHA256 standard is illustrated in Figure 3.

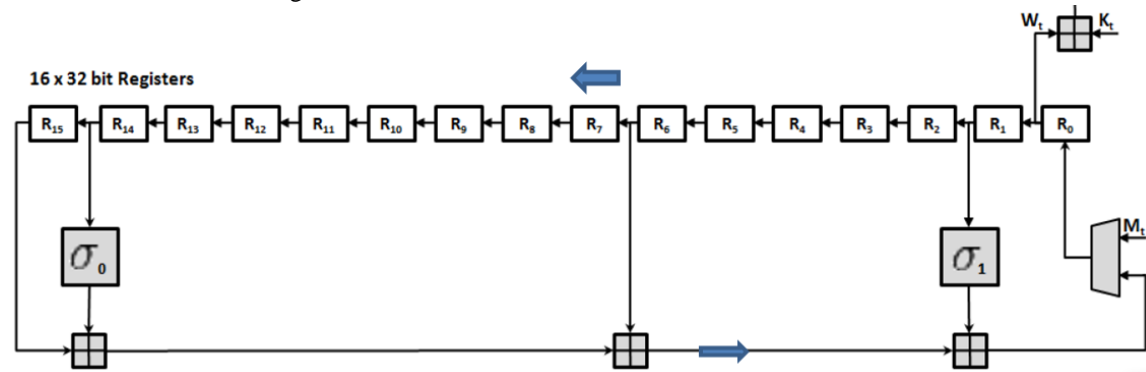


Fig. 30. Shows the first 16 cycles of scheduling.



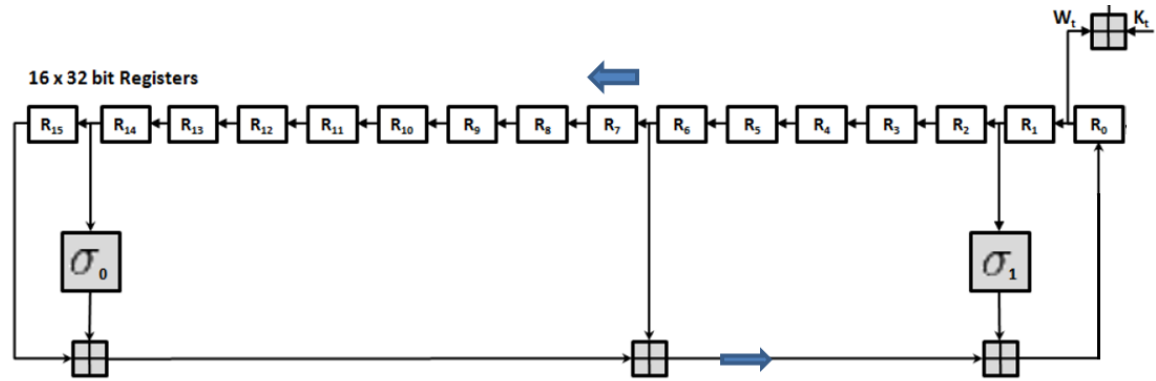


Fig. 31. Shows the last 48 cycles of scheduling.

σ_1 and σ_0 involve cyclic rotation with bitwise right shifts. The right shifts can be performed by performing a cyclic shift and then clearing 'y' bits from the MSB. The code is as follows

```
task shrx(input int a,b,z);
  rotrx(a,b,z);
  for(int i=N-1;i>=N-z;i--) begin
    false(a+i*N);end
endtask
```

Here a is the word to be rotated, b is a working memristor and z is the number of rotations to be performed.

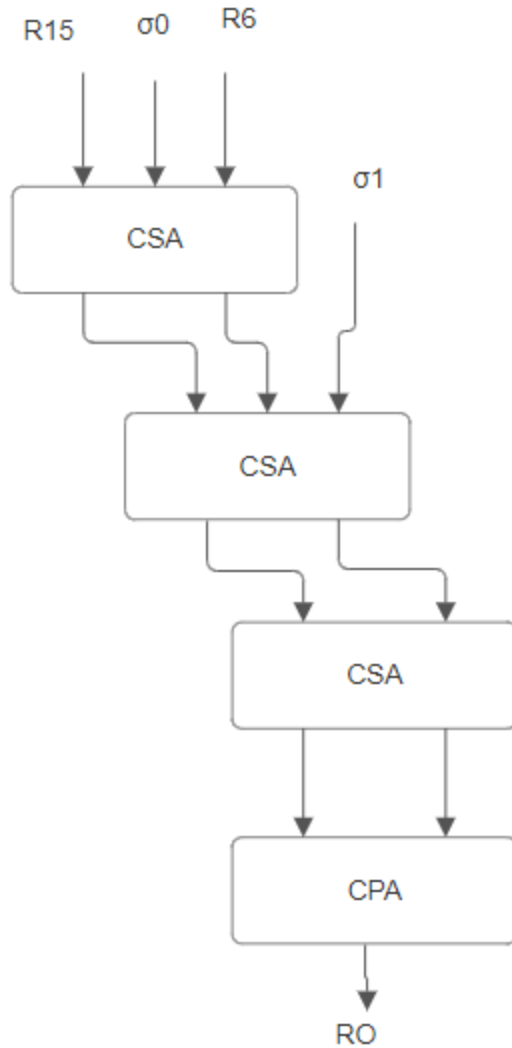


Fig. 32. Timing diagram of the scheduler from $16 < t < 64$.

Since the connectivity and the logic gate in memristive crossbar are all programmable, and the mux is only controlled by time, the mux can be removed. Each register can be implemented with one memristor in the array. At the first 16 cycles, the crossbar is just programmed as a shifter. After 16 cycles, the sigma and addition operations are added, which can be achieved by changing the control sequence.

Another approach is to reduce the operation of moving data, instead of moving the data, we can move the location of operation by changing the control signal. This approach might complicate the control but save some pulse (delay).

Assigning and placement: We must assign each memristor line to each register in the logic block and place them in the crossbar. For example, in the follow crossbar, you will have to assign the area and working memristor for two carry save adders and one carry propagation adder. Each block in the timing block diagram must be assigned to an area in the crossbar; the area can be reused after one operation.

Given below is a simple example. For the actual timing diagrams, refer the excel spreadsheets

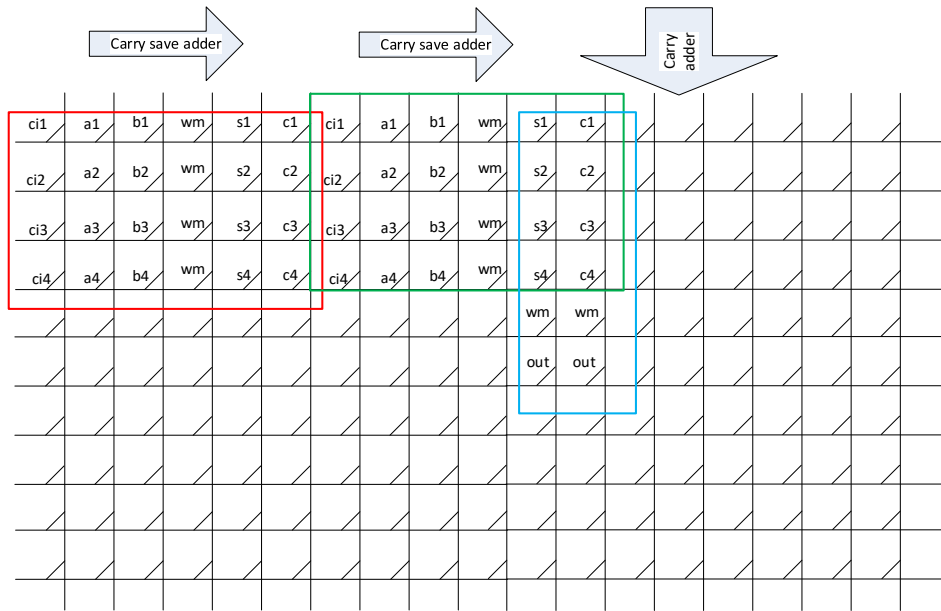


Fig. 33. An example of layout on the crossbar.

FUTURE WORK

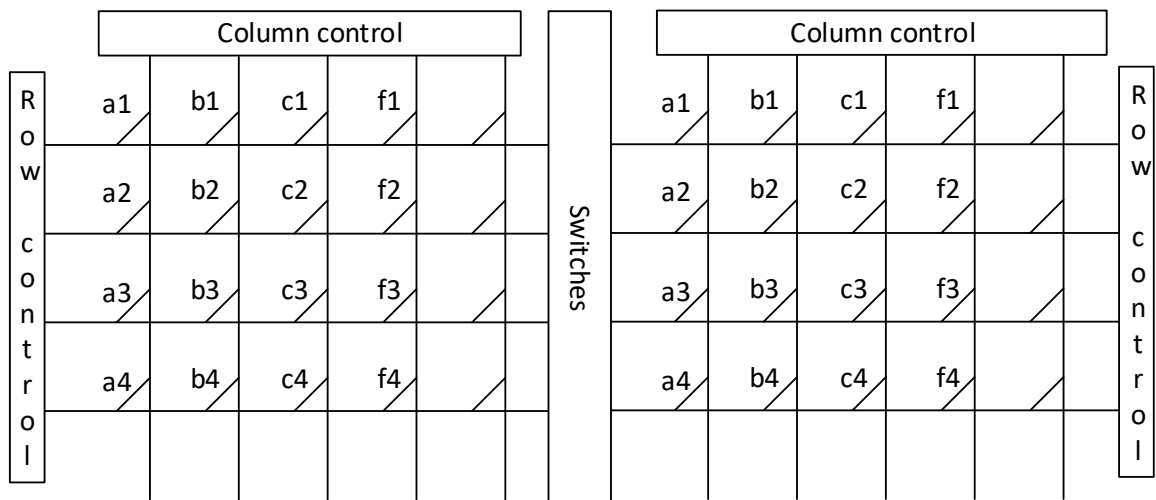


Fig. 34. Connection between 2 crossbars via switches.

From the above diagram, the CMOS Pulse controller looks like in the same level as crossbar, in fact, they are not, following figure shows how the real placement of the CMOS layer of the controller and the crossbar is realized. The CMOS controllers are in another layer.

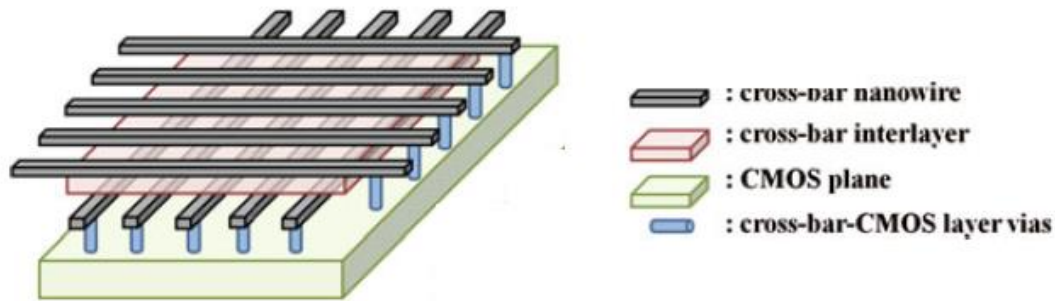


Fig. 35. Visualization of the actual connection between the memristive and CMOS components.

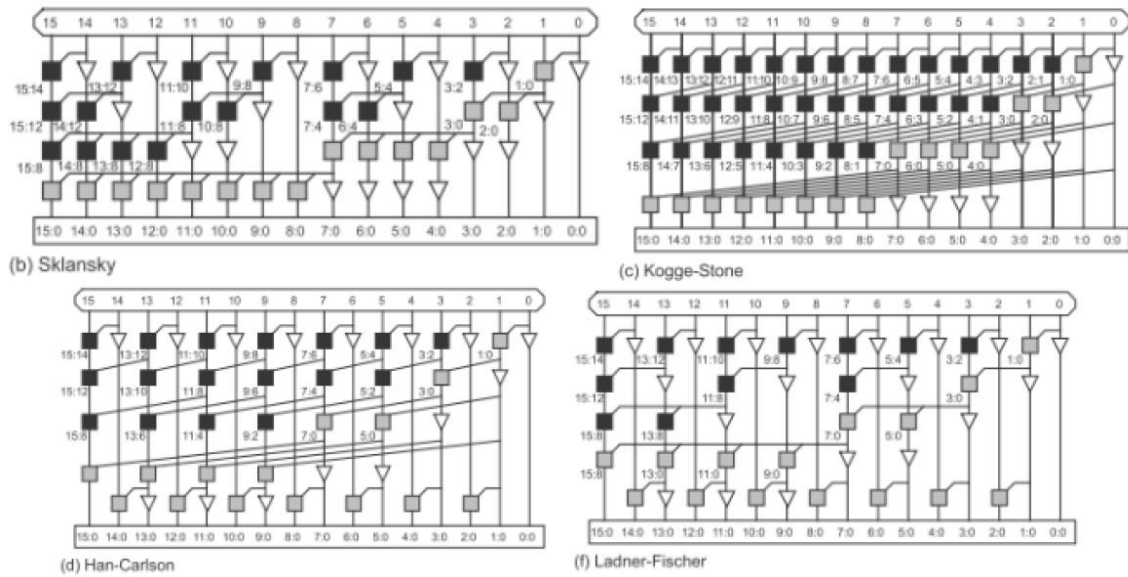
Visualization of the implementation concept for the proposed hybrid crossbar architecture. Memristors are fabricated directly above the CMOS plane. The logic units in the CMOS plane control the voltage supply to each terminal in the memristive crossbar through vias.

After this model is fully developed, we can separately test out each part of the circuit and clearly derive the number of pulses we need and the size of the memory we need to store that instruction. Also, by examining the instructions we may be able to find some patterns such that it can be used to modify our pulse generator by replacing simple counter to a more complicated program counter.

Remaining problems:

1. Pulse generators timing and synchronization. Multiple clock domain design (maybe).
2. Some optimization ideas: Copy operation with imply is negation copy, it will take one more cycle to restore. Hence, if the negation is built in the operation, we can skip 1 cycle and 1 memristor everytime.
3. The carry propagation adder can be replaced with another faster adder implemented with memristors. In this approach we just need to map the fastest adder implementation like CLA Adder, pre-fix adder into memristor crossbar. And then allocate a different memristive array for each branch.
5. Majority of the time bottleneck occurs in the Compression unit, especially during t from 0 to 16 where kt is generated just by shifting.
6. Need to test the CPU for the circuit.

Prefix adder structures:



32-bit CLA with 4-bit blocks

