ARRAY

TECHNICAL DETAIL

FOR THE IMPLEMENTATION OF:

A COMPUTE-IN-MEMORY ARITHMETIC CORE WITH
+85% EFFICIENCY AND THROUGHPUT GAINS
USING STANDARD CMOS TECHNOLOGY

www.arrayarchitecture.com

[ARRAY ARCHITECTURE, CORP.]

Part I Analyzes Traditional Adders. Ripple-carry adders are simple but suffer inherent carry-propagation delay. State-of-the-art parallel adders (carry-lookahead, prefix trees) trade this for circuit complexity, deeper gate depth, and poor scalability — still falling short of the performance demanded by operation-intensive workloads like matrix multiplication (AI/ML/neural networks) and Bitcoin mining.

Part II Addresses the Root Cause of the Efficiency Problem: Von Neumann Bottleneck. The vast majority of energy is spent shuttling data between memory hierarchies and processing cores. Solving this has proven difficult because the regular grid topology of memory arrays is fundamentally incompatible with the irregular topology of parallel adders. Proposed remedies — exotic memory technologies (ReRAM, FeFET, MRAM) for compute-in-memory, or near-memory approaches like HBM — remain commercially unviable due to prohibitive fab upgrade costs, design complexity, and hardwired single-task architectures that demand complete redesign for any algorithmic change.

Part III Introduces a New Mathematical Definition of Addition. We detail a Finite State Machine that co-locates logic gates and memory cells bit-by-bit. This peer-reviewed and published method eliminates carry propagation and data movement. We provide the full RTL block diagram of the Simple and Linear Fast Adder Core, VHDL and SystemVerilog Simulations, and illustrative EPWave examples. Our patented finite-state machine redefines addition as an iterative transformation of two input columns into two new output columns, reusing the same memory registers. It converges in logarithmic time, uses exactly one Half Adder per significant bit, and routes as a perfectly regular Manhattan grid. It is Compute-In-Memory without exotic materials, without analog noise, and without dense 3D packaging. Just standard edge-triggered flip-flops and logic gates (DFFs + AND/XOR gates).

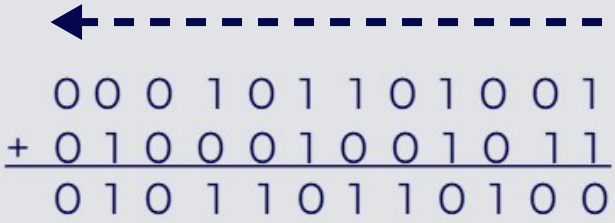
Part IV Demonstrates a Complete, Practical Application. The full SHA-256 compression function executed entirely in-situ within a 32-bit edge-triggered memory array using our multi-input adder design. All 64 rounds run inside the memory array, eliminating data migration costs 64 times over while simultaneously reducing gate depth.

Index

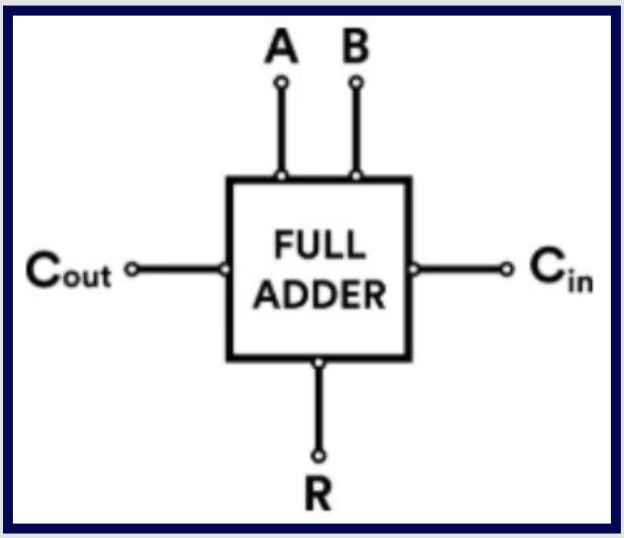
Part I: Traditional Adders

Ripple-Carry Adders

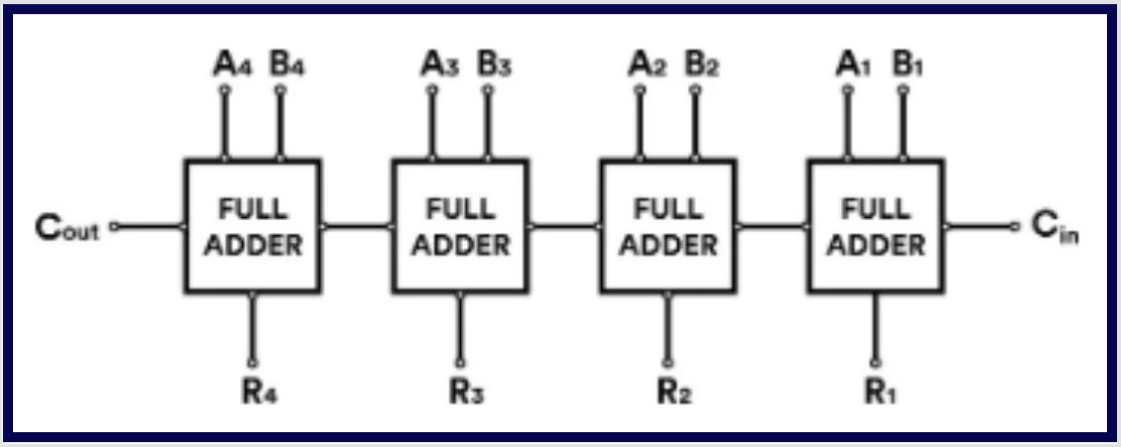
Ripple-Carry Adders are characterized by having a linear design achieved by connecting several subunits in series. It is a simple design and we can scale it for more bit capacity simply by connecting more subunits in series. These first generation adders are called Ripple-Carry Adders because **they are basically executing the grade-school algorithm for addition, whereby the output is calculated bit-by-bit, from right to left.** To add two binary numbers A= 000101101001, and B=010001001011 we start by adding the first digit of A with the first digit of B and write the result in the third row. Then, move to the next bit to the left. The name of this method comes from this "ripple effect" propagating from right to left one bit at a time.



We need special units called "Full Adders" to perform this one-bit addition. Each FA has the two usual inputs, plus a third input for the carry-over bit from the FA to the right. Addition is done bit by bit, one bit at a time starting on the right, and moving left.



To execute this algorithm with a digital circuit, we form a linear structure of FAs connected in series. Each FA executes one-bit addition and sends the carry-over bit to the next significant bit on the left. **For *n*-bit addition we need *n*-many FAs connected in series.** Each FA adds the two corresponding bits of input and the carry-over bit, then writes the result in it's corresponding Register (R1, R2, R3, R4,...) and sends a carry-over bit signal to the FA on it's left. **Below we can see the linearly scalable structure of the *n*-bit adder, with a behavioral diagram of a four-bit adder.**



The circuit is simple enough, but the downside is obvious. A simple Ripple-Carry Adder is slow because each bit must wait for the carry signal to "ripple" from the previous bit. Each full-adder must wait for its predecessor's carry-in before it can produce a correct sum and carry-out. **This is not a limitation of the hardware, it is a limitation of the algorithm itself.** The mathematical method dictates that every significant bit has to wait for the carry over signal from the previous significant bit before computing its output. You must compute the outputs one at a time, taking a full clock cycle on each iteration. **This leads to a huge time delay, that only gets bigger as the number of bits grows.**

Parallel Adders

Instead of waiting for a carry to ripple through the chain, **the Carry-Lookahead Adder uses a separate, fast circuit (the Lookahead Carry Unit) to compute all carry bits simultaneously by looking at the initial inputs.** This trades more complex hardware for a significant gain in speed. This principle of computing carries in parallel is the foundation for most high-speed adders, even though modern designs use more advanced techniques (like prefix trees e.g., Kogge-Stone, Brent-Kung) to make this concept scalable to 64 or 128 bits.

But, operation intensive workloads like mining or matrix multiplication consume massive amounts of energy. **Even with super-fast parallel adders, traditional processors hit fundamental walls. Parallel adders are not fast enough, they are difficult to scale, they have a large area and power demand, among other major setbacks.**

Memory Wall

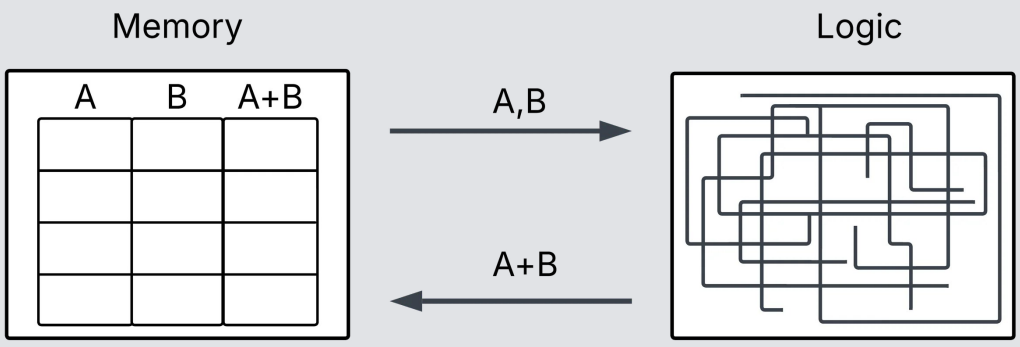
The Problem: Processor speed (GHz) has vastly outpaced main memory (DRAM) speed. A CPU can perform hundreds of operations in the time it takes to fetch a single piece of data from memory.

Adder Context: A parallel adder can calculate a 64-bit sum in one clock cycle. But if it has to wait 100+ cycles for the operands to arrive from memory, its speed is irrelevant. The adder is idle most of the time.

Power Wall

The Problem: Data movement is incredibly expensive from an energy perspective. Moving a single 64-bit word from memory to the processing cores consumes many times more energy than the actual operation itself.

Adder Context: You are spending +95% of your energy just to deliver the numbers to the adder. This is astronomically inefficient for data-center workloads.



Part II: Problems with Bottleneck Solutions

Even though there is plenty of potential for increasing the adder's efficiency, it is no longer the primary bottleneck. The problem also comes from everything around it. That is why CIM and other solutions are being researched.

Over the decades there have been a number of proposed solutions aiming to break down the power and memory walls, but each faces significant technical and economic challenges.

Near-Memory Computing. Placing processing units closer to memory (on the same package) to reduce data travel distance is one strategy.

Challenges:

- Immense heat dissipation. Memory is low-power logic; processors are high-power.
- Putting them together creates "hotspots" that are difficult to cool.
- Extremely high packaging costs (e.g., Silicon Interposers), lower yields, and a more complex supply chain.

Computing-in-Memory (CIM) / Processing-in-Memory (PIM). Perform computation directly inside the memory array, eliminating data movement.

Challenges:

- DRAM/Flash cells are not designed for computation.
- Modifying them is slow, unreliable, and severely limits the complexity of possible operations.
- It often requires new, non-standard memory technologies (e.g., ReRAM, MRAM).
- The entire semiconductor industry and software ecosystem are built around the CPU-Memory separation. Adopting CIM requires a fundamental redesign of both hardware and software, a monumental economic undertaking.

High-Bandwidth Memory (HBM). One solution is to provide a massive, fast pipeline of data to the processor.

Challenges:

- Extremely high cost. HBM uses complex 3D-stacking and advanced packaging, making it significantly more expensive than traditional GDDR memory.
- It's not a general-purpose solution but a premium one reserved for high-margin products like GPUs and AI accelerators.
- Data still has to move from the HBM stack to the CPU/GPU cores, consuming energy. It mitigates, but does not solve the power wall.

Neuromorphic Computing. Mimic the brain's architecture for ultra-low-power, event-driven, analog computation.

Challenges:

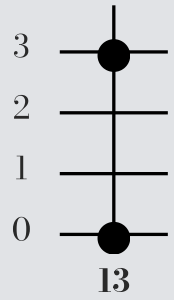
- Radical paradigm shift. It's often analog, asynchronous, and uses non-von Neumann principles. This makes it highly specialized for specific tasks and terrible for general-purpose computing.
- Precision is also a challenge.
- There is no established software ecosystem. Programming these systems is fundamentally different, requiring new languages and tools. The manufacturing process is also non-standard, keeping costs high.

Part III: Simple and Linear Fast Adder

Addition Algorithm

Adders, in any technology platform, are working within the paradigm of the carry-over algorithm or some modification/variant thereof, carrying out a fast version of the carry-over algorithm, computing certain parts in parallel, or dividing the inputs, computing the smaller additions and putting back together the results.

We define addition by means of a completely new method in terms of a Finite State Machine. **A single column with a given configuration of particles represents a number.** Each column has a configuration of at most one particle per level. A particle on the base level '0', is worth a single unit. A particle in level '1', is worth two units. A particle in level '2' is worth four units and a particle in level '3' is worth eight units, etc. The representation of numbers is the usual binary form. We have presented this in a column interpretation with particles on different levels because it will be illustrative for how the operation is defined. For example, to represent the number $13=2^0+2^2+2^3$, we have the column configuration below.



To execute addition of two numbers, we define a Finite State Machine. A *state* is determined by two columns, and each column has a number of levels. Each level is for one bit of accuracy; to carry out *n*-bit addition we require *n* levels. The system evolves discretely in time. To pass from one state to the next, two new columns are formed using two simple rules. These rules tell us how to form two new output configurations, provided two input columns.

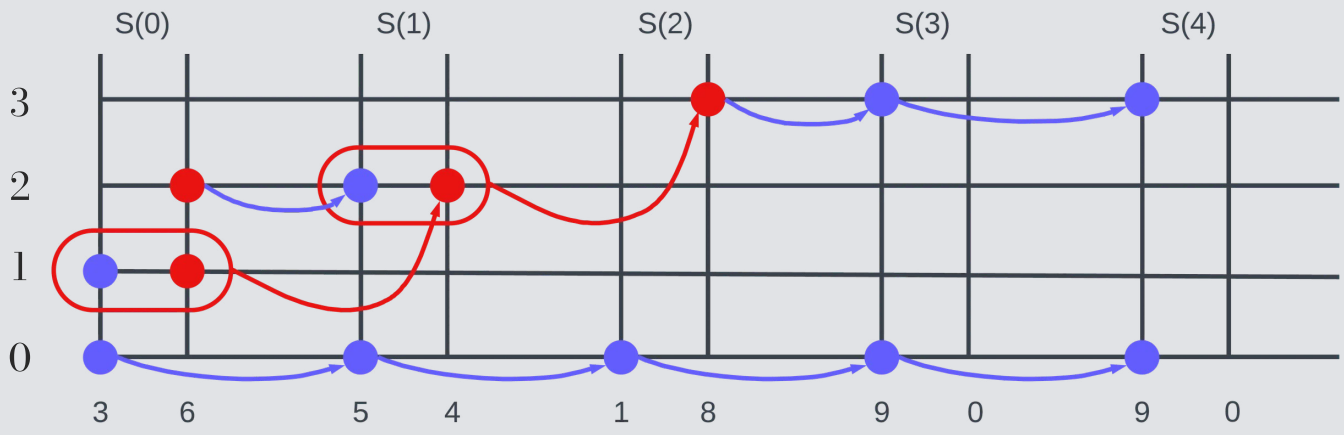
- ***Rule #1 for Symmetric Difference:*** *In any level where there is exactly one particle in the current state, we will place one (blue) particle in that level on the left column of the next state.*

- ***Rule #2 for Intersection:*** *In any level where there are exactly two particles in the current state, we will place one (red) particle* one level up, *on the right column of the next state.*

We have a fixed and well defined set of rules that take in two inputs, and produce two new numbers. Let us illustrate the rules with a 4-bit example.Suppose we wish to add $3=2^0+2^1$ and $6=2^1+2^2$. The initial inputs, (3,6), are **the initial state S(0), given by the leftmost pair of columns.** The left column represents the number '3' with particles in level '0' and '1'. The right column represents the number '6', that is why it has particles in levels '1' and '2'. To find the addition of these two numbers, 3+6, **we will apply the two rules to form two new columns.** The first rule indicates that the left column of **S(1)** must have particles in levels '0' and '2' because these are the levels tha have exactly one particle in **S(0).** The second rule indicates to place a particle in level '2' of the right column of **S(1)** because level '1' of the initial state **S(0)** has two particles. Thus, our two new columns are formed: $5=2^0+2^2$ on the left and $4=2^2$ on the right. **We have simply reduced the initial inputs (3,6), and turned them into two new columns (5,4).**

We will apply the same process to these new inputs (5,4). The first rule tells us that the left column of **S(2)** will have a particle in level '0'. The second rule tells us the right column of **S(2)** will have a particle in level '3'. The new columns represent the numbers (1,8). Applying the rules to these new inputs (1,8) gives us two new columns (9,0). If we apply the rules again to the inputs (9,0) something interesting happens. The output obtained will be the same configuration (9,0). This means **our FSM has reached a stable state. This stable state happens to give us the result to the addition 3+6=9.** Determining if a state is stable is easy. The right column is empty.

There are several things to point out from this method. First, **it is not a prefix method like the carry-over algorithm.** We can simultaneously work on every level (each level is a significant bit) in parallel. We do not have to calculate the bottom levels first and then move upwards one level at a time, as in the case of the carry-over propagation. Next, we notice that **the time for addition will be determined by the number of iterations before the system stabilizes.** In our example it took three iterations for the system to stabilize. **Executing *n*-bit addition takes $\log_2 n$ iterations to stabilize, on average, and at most *n+1* iterations.** This is important because logarithmic time is the fastest theoretical time complexity we can hope for in any addition algorithm.



Bit-wise Logic

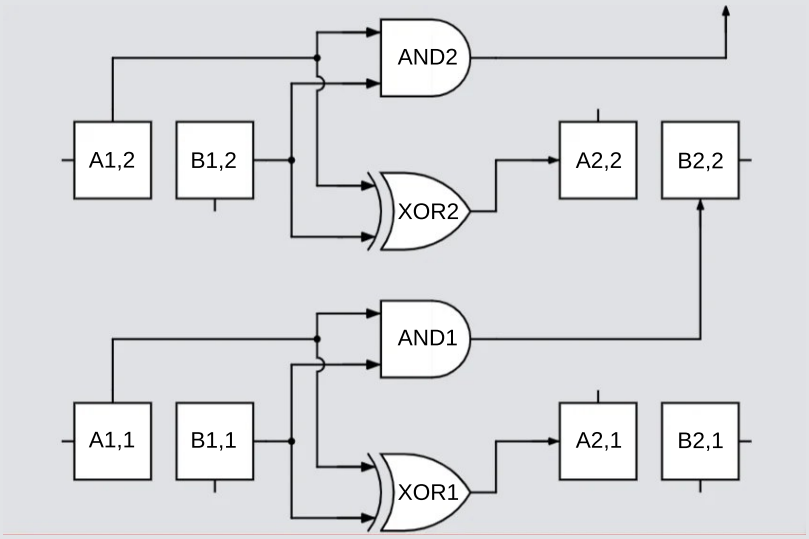
Given the theoretical time complexity of the algorithm, which we know is optimal, we now focus on the logic and structure of a circuit to execute the algorithm. We start with two columns of memory (two *n*-bit registers). **On any given level we need a Half Adder that will receive two inputs (signal A from the left column and signal B from the right column) and will compute two output signals (one AND output and one XOR output).** The AND output is directed to the new right column (with a bit shift up), and the XOR output is directed to the new left column (in the same significant bit). The possible cases are

- **Both inputs are '0':** In this case the unit will output '0' to the right and left columns. That is, both AND/XOR gates will output '0'.

- **One of the inputs is '0' and the other input is '1':** In the case that our inputs are (0,1) or (1,0), then the signal to the left column (XOR output) is '1' and the signal to the right column (AND output) is '0'. This is the part of the circuit equivalent to the first rule.

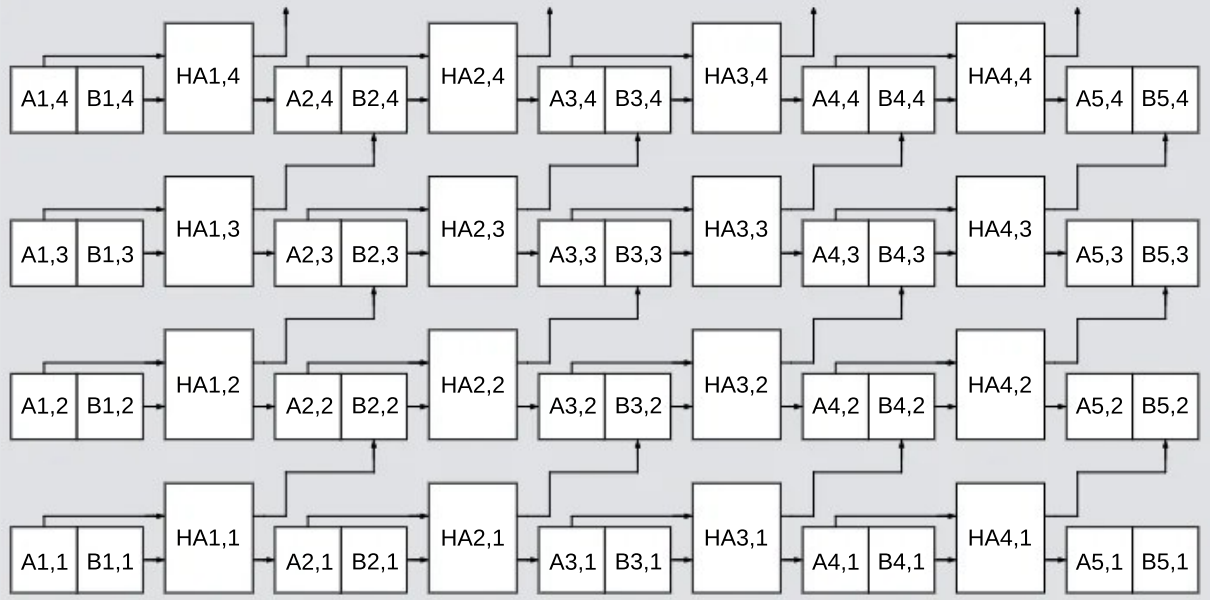
- **Both inputs are '1':** In the case that both input bits are '1', the XOR gate outputs a '0' to the left column, and the AND gate outputs '1' to the right column, with a bit shift up. This is the circuit equivalent of the second rule.

A Half Adder (HA) executes the functions we need on each level. The outputs we need for the new right column can be calculated using AND gates (outputs '1' if and only if both inputs are '1'). The outputs we need for the new left column can be calculated using XOR gates (outputs '1' only when exactly one of the inputs is '1'). **This is calculated bit-wise, so that we only need one HA per bit of input.** The output of the XOR gate is connected to the new left column on the same bit level, and the output of the AND gate is connected to the new right column, one bit level up.

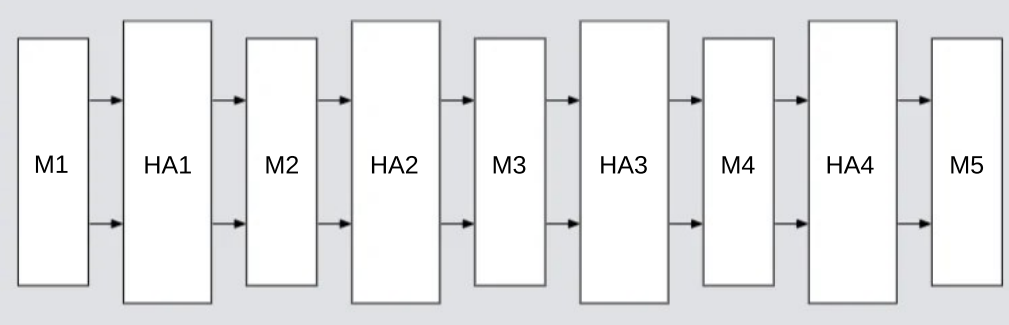


Adder Circuit Logic

There are pairs of registers to store the binary configuration of the states. The inputs which are stored in the first pair of registers (A1,i and B1,i) will pass bitwise through the XOR/AND gates of the half adders HA1,i. The two bits stored in a given level will both go through the HA of that level, and the output will be stored in the next pair of memory columns (registers A2,i and B2,i). **Every time the data goes through a column of HAs, we are changing to the next state. When the data has been transformed through the logical gates it is directed to two new columns of registers.** The output from the XOR gates is directed to the left column. The output from the AND gates is directed to the right column, with a bit shift.

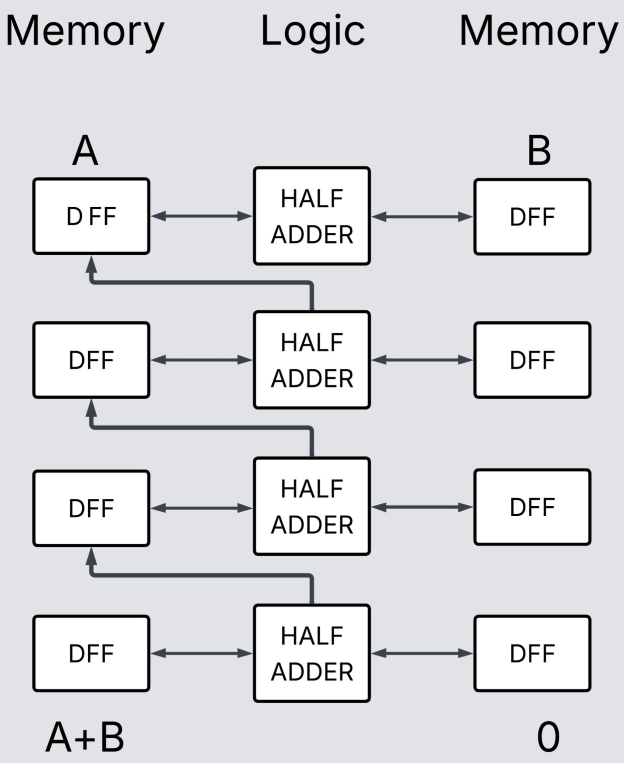


This diagram gives us a good idea of the finite state machine that is our adder. **We start with two columns of inputs** (left-most columns). **Those two numbers are fed bit wise to a sequence of Half Adders** (AND/XOR gates to calculate the bitwise intersection and symmetric difference). **The results of these operations are then sent to two new columns of memory** (next two columns to the right).



Although we have a simple way of passing from one state to the next using new memory registers, **it is an enormous waste of resources to use new memory registers for every iteration.** Especially considering that memory registers (internal memory of the processor) is the most expensive type of memory. So let us find a solution that does not require all of this extra memory. **It is easy to see a solution that involves only two pairs of columns.**

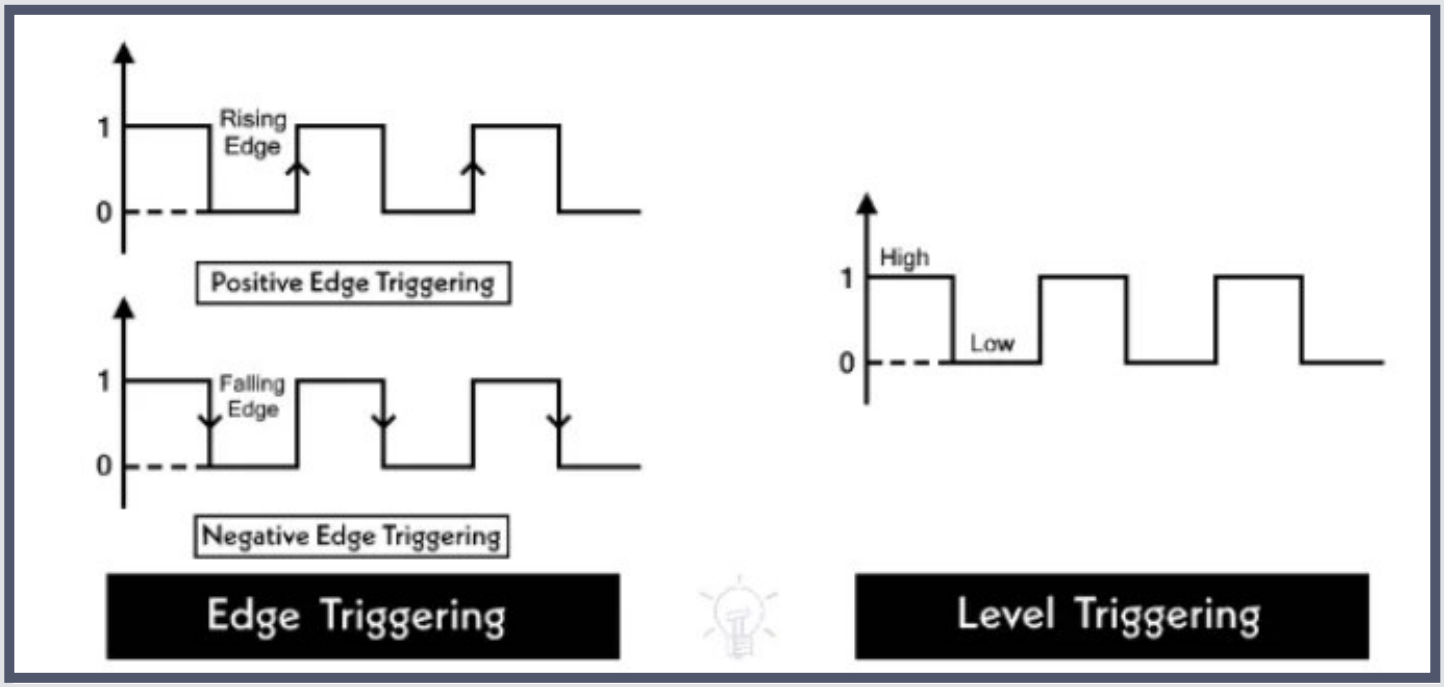
After passing the data through the column of HAs, to compute the new state, the new state is stored in the second pair of memory columns, and the first pair of memory becomes free to be used. Now, when the next state is computed, it can be stored in the first pair of memory registers. **In each iteration we will move back and forth between the two pairs of columns.**



This allows us to save area, silicon footprint and transistor count. The resulting circuit will only have two pairs of memory registers and a single column of HA in the middle. **To pass from one state to the next we move from one pair of registers, through the HA column and into the other pair of registers.**

There is an optimal solution, memory-wise, that only requires one pair of memory registers, not two. **The same memory where the initial inputs are stored, is all the memory we need,** if we carefully apply a feedback loop without a racing problem. **Saving the new state in the same memory registers, after going through the logical gates, requires double-edge triggered flip-flops.**

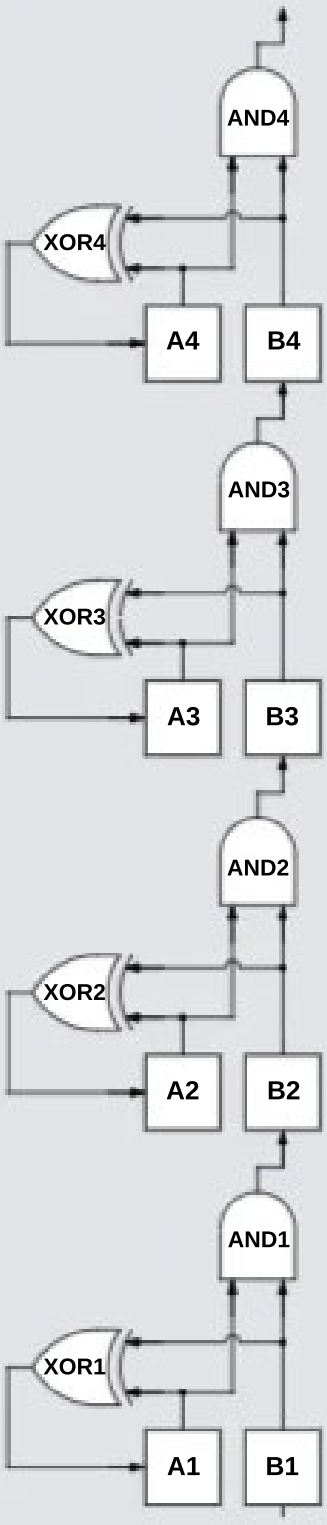
An edge-triggered flip-flop is a unit of memory that reads or writes when the voltage in the clock cycle is rising or falling. This is a different type of memory from level-triggered flip-flops that read or write when the voltage reaches a certain level. Below we see the difference between level-triggered and edge-triggered memory registers.



To execute the function we need, **the memory registers have to read on the rising edge** (at the beginning of the clock cycle), then **during the voltage plateau (mid-clock cycle) the data passes through the XOR/AND logical gates,** and then finally goes back to the memory registers **and is written when the voltage descends** (at the end of the clock cycle).

A version of the adder is possible that only requires two registers of memory (one column of memory for each input). **We can achieve a logarithmic (average) time adder using the theoretical minimum of memory resources.** The gate depth is constant, the topology is linearly scalable, and the connections are simple and regular. All these factors translate into an **optimally efficient (in time, energy, area and gate count) arithmetic core with full bit scalability, little R+D overhead, and standard manufacturing processes.**

One of the most important characteristics of this circuit is that **the memory and logic gates are co-located in a natural one-to-one correspondence, making it a CIM architecture** (Memory/ALU bandwidth is eliminated) because of this natural proximity between corresponding (communicating) memory and logical gates. **Our *n*-bit adder consists of *n* copies of identical subunits connected in series.** Each of these subunits consists of two edge-triggered registers, and two standard logical gates (AND/XOR), as pictured to the right.



Circuit Simulation

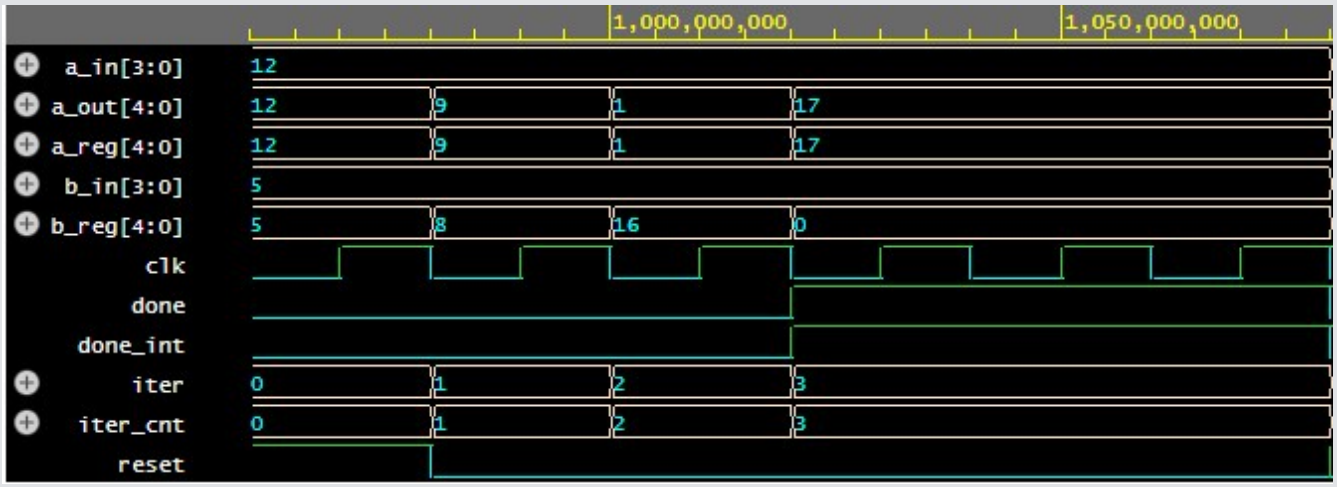
We have successfully verified the SLFA algorithm in VHDL simulation, proving its correctness, scalability, and synthesizability. You can see our simulation code, results and EPWave examples for signal analysis online at EDA Playground, and GitHub:

SLFA-4 bits (VHDL)

SLFA-8 bits (VHDL)

SLFA-4 bits (SystemVerilog)

SLFA-8 bits (SystemVerilog)



What This Simulation Represents

This confirms the mathematical model and the finite state machine operation, and represents functional verification of the SLFA algorithm's logic and correctness. It confirms that the finite state machine (FSM) operates as mathematically proven in the patent, accurately performing addition across multiple test cases, including edge cases and random inputs. Specifically, this simulation demonstrates:

- *Algorithmic Correctness:* The SLFA executes addition in logarithmic time, with proper state transitions, carry handling, and completion detection.
- *Bit-Level Precision:* The XOR and AND operations are correctly applied in parallel, with the carry evolving through the FSM iterations.
- *Scalability:* The same architecture scales cleanly from 2-bit, to 4-bit, to 8-bit (and will scale to 32-bit) without fundamental changes.

We have verified the behavior across thousands of test cycles, proving that the mathematical model translates faithfully to a synthesizable hardware description.

What This Simulation Does NOT Provide

This is not a physical silicon simulation. It does not reflect:

- *Power consumption:* There is no transistor-level modeling, switching activity data, or dynamic power estimation.
- *Real-time latency on silicon:* While the clock cycles are counted, the actual propagation delays, gate switching times, and physical wire delays are not modeled.
- *Compute-in-memory behavior:* The CIM nature of the SLFA, where memory and logic are integrated into a single regular grid, is not captured at this abstraction level. The simulation treats registers and logic as separate HDL constructs, not as physically co-located cells within a grid.

In short, this is a logic simulation, not a performance, power, or physical implementation benchmark.

Why This Is a Critical Milestone

Functional verification is the foundation upon which all subsequent design stages depend. Without this validation, we cannot proceed to synthesis, place-and-route, or physical design with confidence. This simulation proves that:

- The patent's **mathematical model is correct.**
- The algorithm is **implementable in hardware.**
- The design is **synthesizable and scalable.**

Next Steps: From Simulation to Silicon

To quantify the real-world advantages of the SLFA, the next phases will focus on:

- FPGA Prototyping: A hardware prototype on an FPGA will provide real-world timing, area, and power estimates—closer to silicon performance.
- ASIC Synthesis: Using industry-standard tools (Synopsys, Cadence) to synthesize the design for a target process node (e.g., 7nm), providing accurate power, area, and frequency metrics.
- Physical Design: The most critical step for proving the CIM advantage. We will place and route the regular grid of memory+logic cells to demonstrate the elimination of the on-chip memory wall.

These stages will deliver the hard data that matters: Tokens per dollar, Joules per Terahash, hashrate per dollar, etc.

Conclusion

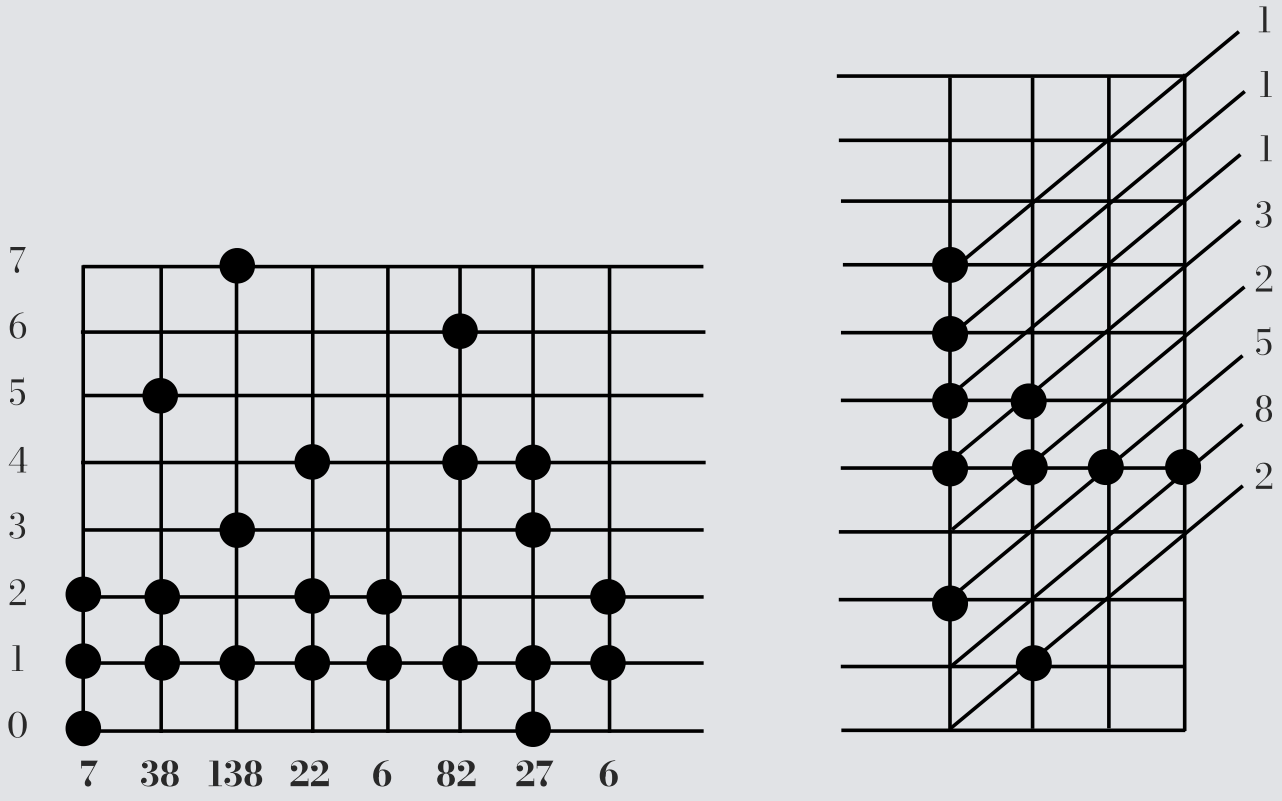
We have achieved the first essential milestone: proving the SLFA works. The next phase is to prove it outperforms. That requires silicon-level modeling, not just algorithmic simulation. The SLFA's CIM advantage cannot be fully captured in a VHDL testbench. It must be demonstrated in silicon.

Multiple-Input Addition Algorithm

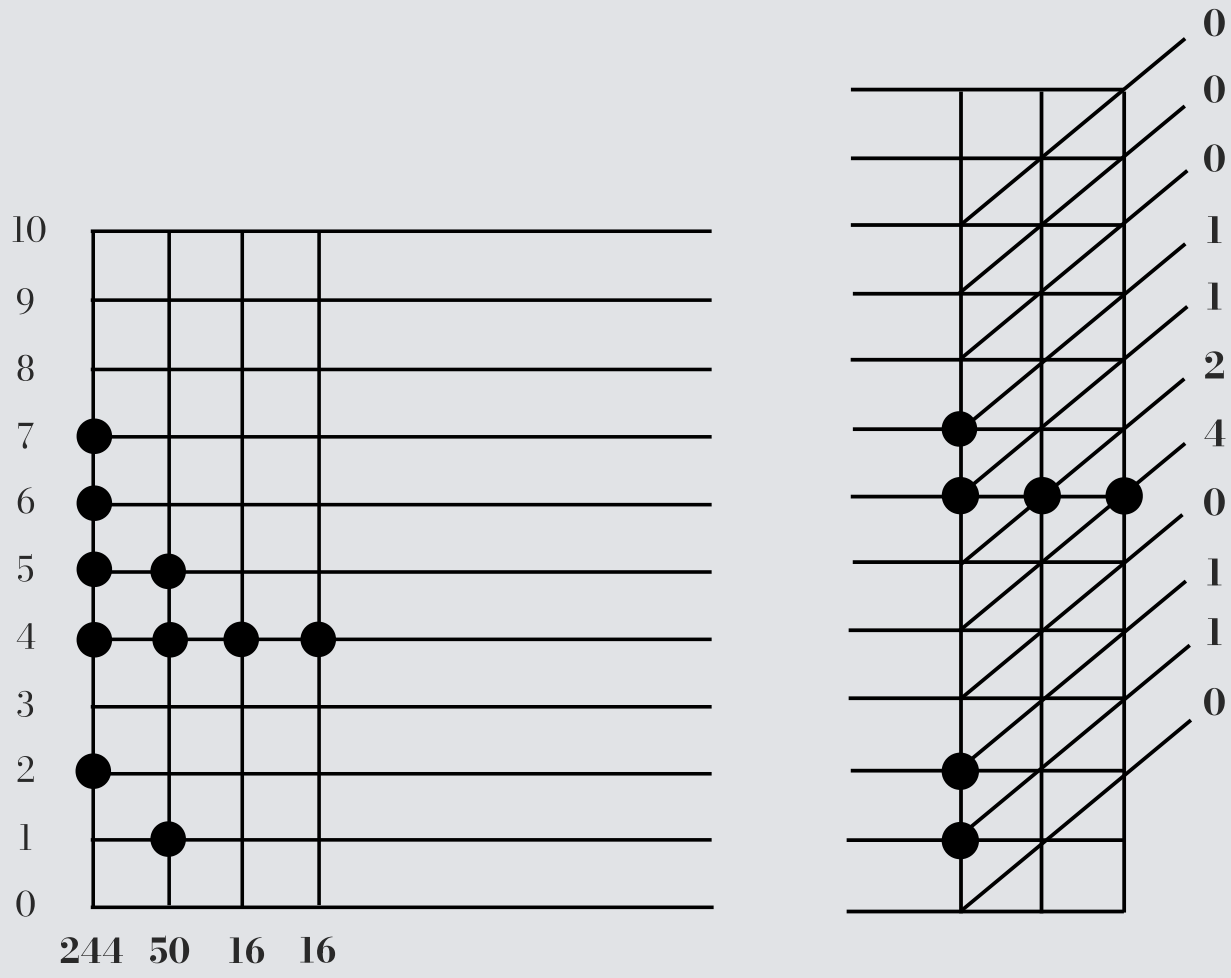
A multiple-input adder is possible in the form of a simple and rectangular array composed of several SLFAs connected in a Manhattan-routed grid. There are no diagonal or irregular connections. All connections are restricted to streets and avenues.

The multiple-input addition algorithm is a generalization of the two-input case we have detailed above. Now, we will detail the general case for multiple inputs. We will use addition of 8 inputs as an example, but other cases are similar.

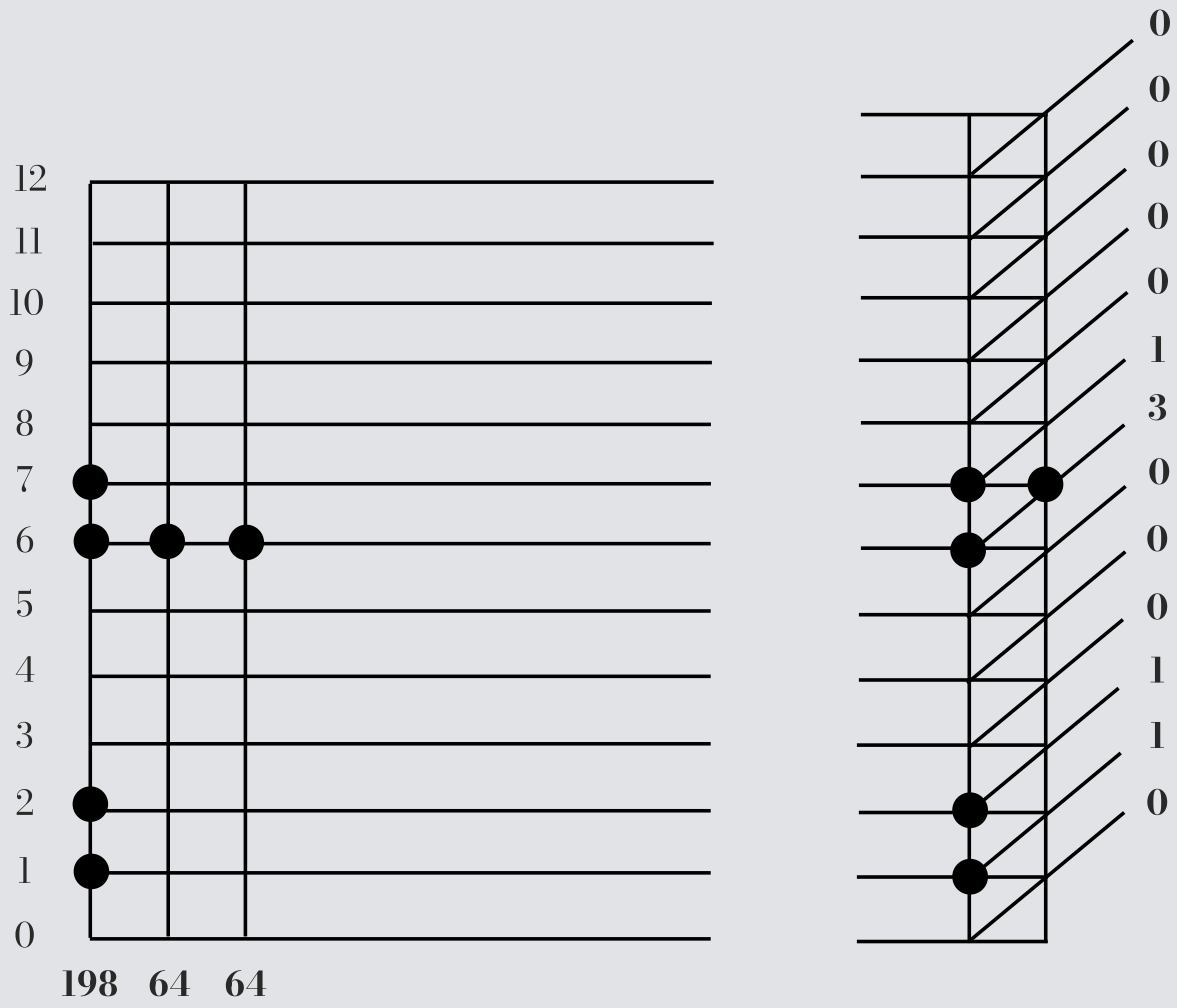
If we wish to add *n*-inputs, the first step will be to reduce the number of inputs to *max*(*n*)+1. For example, if we start with eight inputs, we will reduce this operation to an addition of *max*(8)+1=*max{3}+1=3+1=4* inputs, using the following method. Given the eight initial inputs below, we will count the number of particles on each row, and write those numbers in four new columns. We need four new columns because it takes four digits to represent the number eight in binary form 8=1000. Write the number of particles in each row, on the diagonal axis, starting from left to right, as shown below.



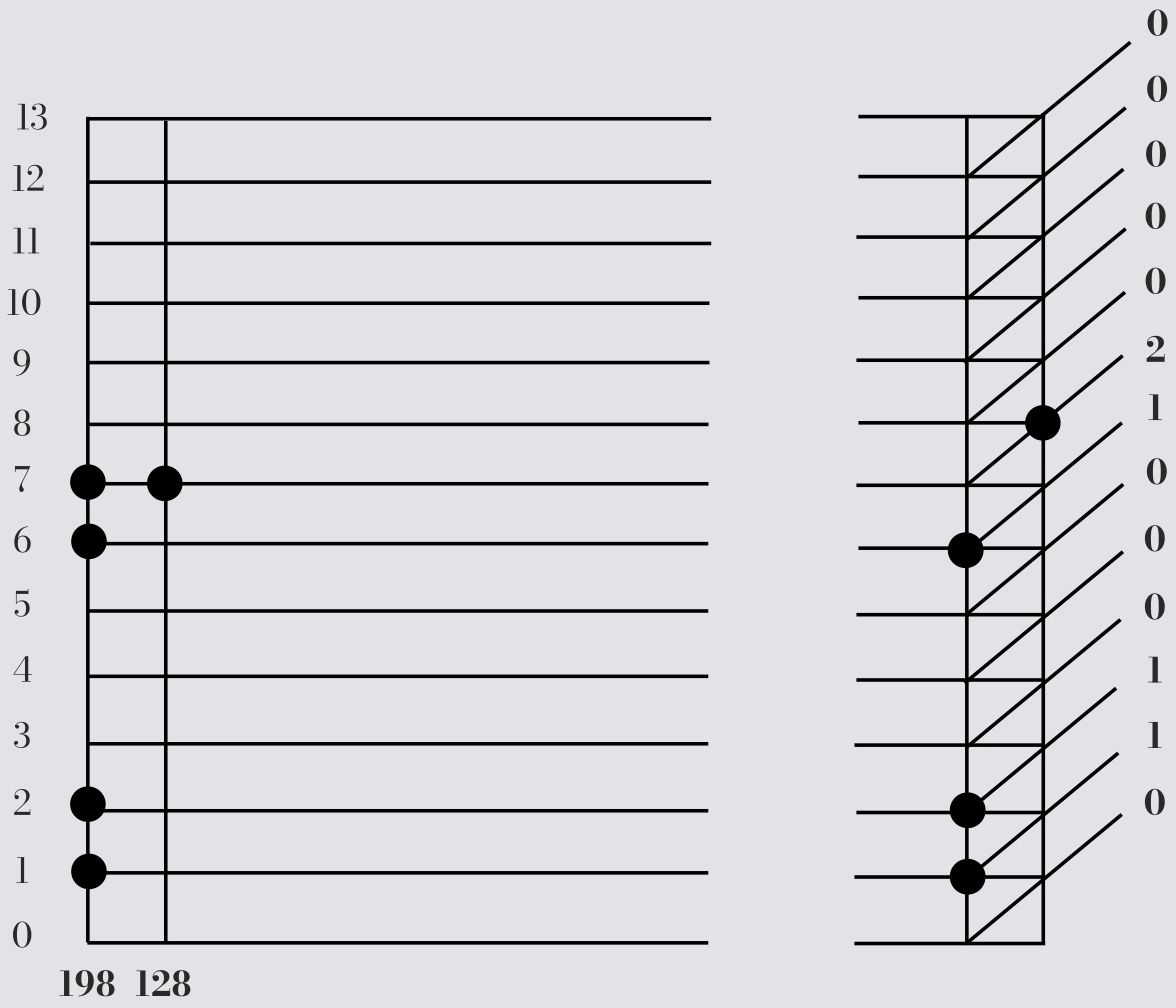
If we iterate this method, we can reduce the four columns from the last step, to a total of max(4)+1=max{2}+1=2+1=3 columns.



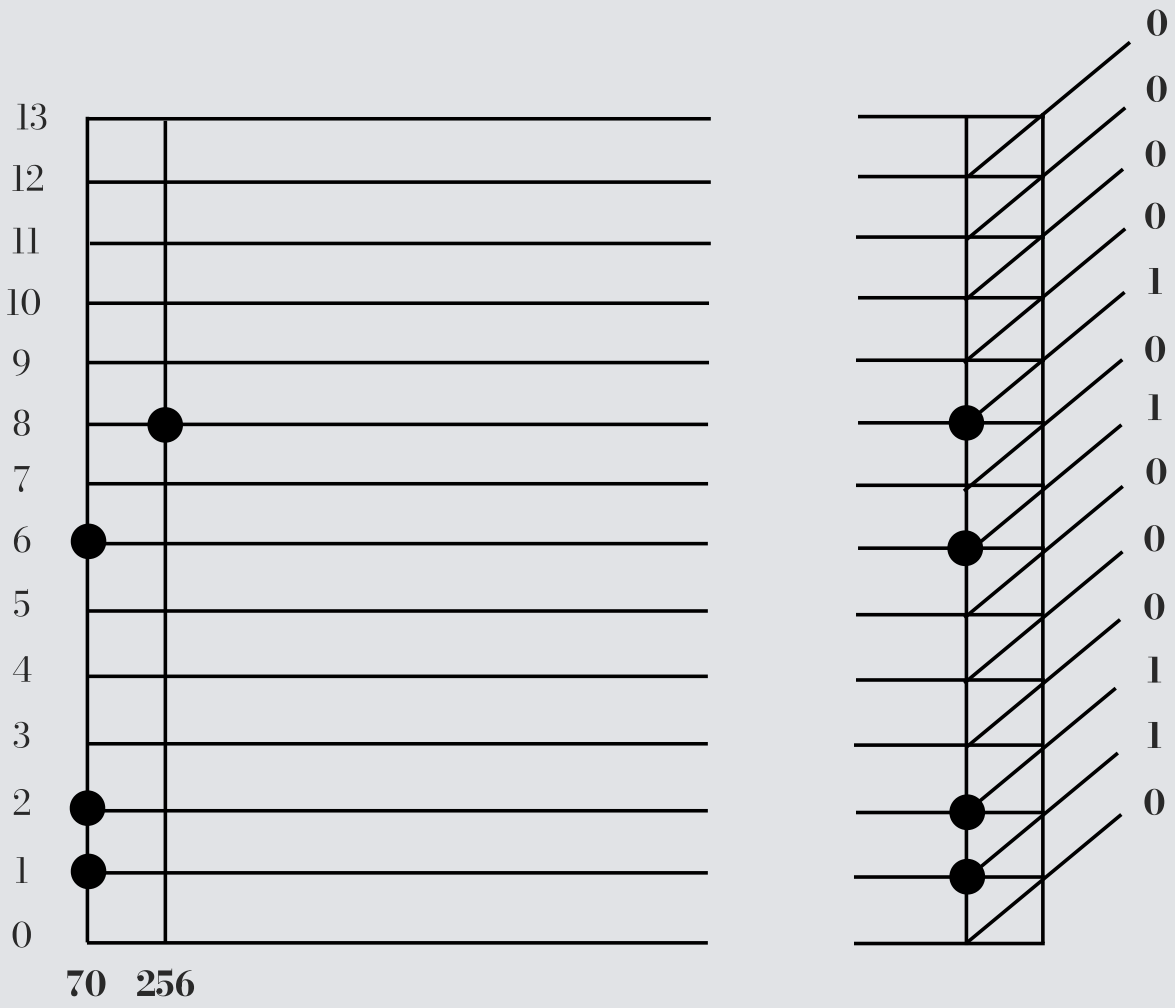
Then, we reduce those three columns to $max(3)+1=max\{0,1\}+1=1+1=2$ columns.



Something interesting, but expected happens here. If we apply the column reduction method to two columns we will get two new columns, according to the formula *max*(2)+1=max{1}+1=1+1=2. The two new columns are the same result we would get using the basic two-input Finite State Machine that we detailed above.



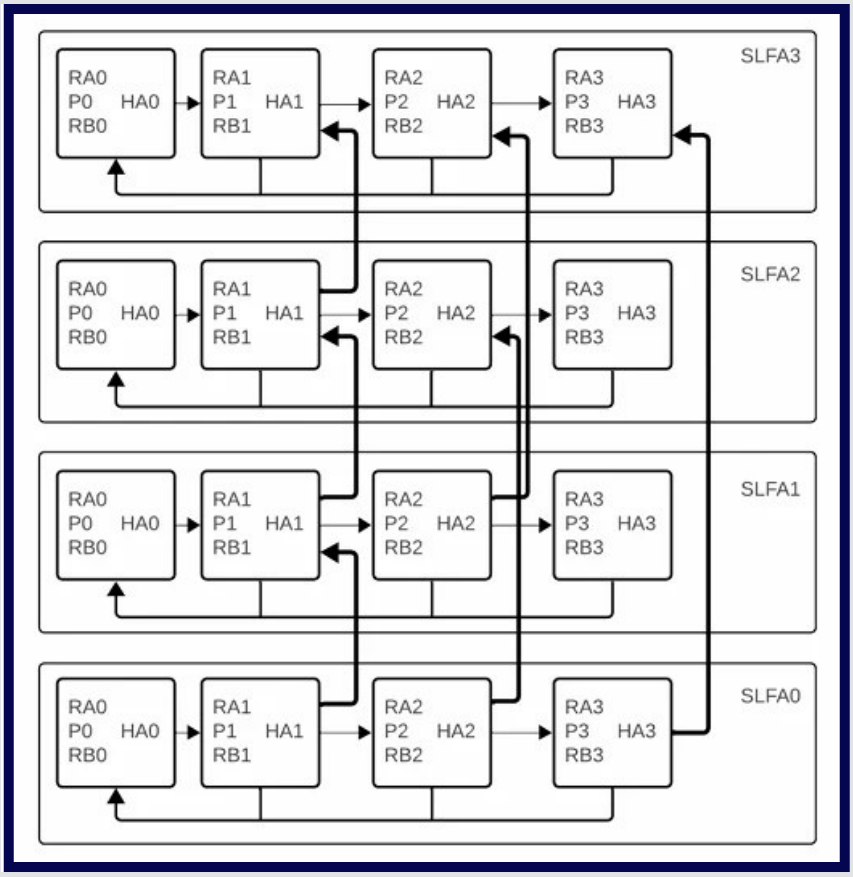
We have to iterate one more time to reach the final stable state that satisfies the stopping condition (empty right column). At this point, if we apply either of the algorithms (two-input or multiple input algorithm) we obtain the stable state configuration representing the number 326, shown below as the final result of our iterations.



Multiple-Input Adder Circuit

Recalling that it is possible to execute the iterations of the two-input algorithm in two columns, instead of multiple pairs of columns, **we are able to apply a similar technique to limit the execution of this multiple-input algorithm to the original *n* columns** that store the original *n* inputs.

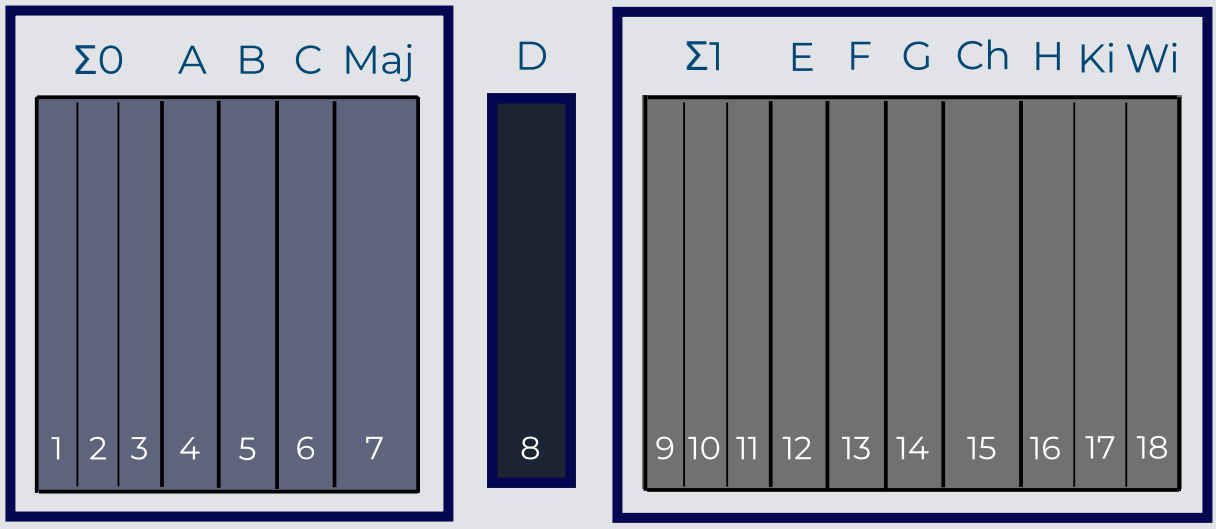
To achieve this, we need an SLFA on each row to count one-by-one all of the particles on that row. Then, once the rows have counted their particles (in parallel) we will displce the results to their corresponding diagonal. This is achieved by displacing the particles in the second column, one row upwards. The particles in the third row are discplaced two rows up, the fourth column is displaced three rows up, etc. This is shown below in a diagram that allows four inputs.



The details regarding the circuit that executes this multiple-input algorithm are provided in a peer-reviewed paper published as proceedings of an IEEE international conference: J. P. Ramírez, "Simple and Linear Fast Adder of Multiple Inputs and Its Implementation in a Compute-In-Memory Architecture," 2024 International Conference on Artificial Intelligence, Computer, Data Sciences and Applications (ACDSA), Victoria, Seychelles, 2024, pp. 1-11, doi: 10.1109/ACDSA59508.2024.10467957.

Part IV: In-Situ SHA Compression Function

We can exploit the multiple-input adder architecture for the In-Memory execution of the 64-round SHA-256 compression function. The circuit that achieves this requires 32-bit registers and a total of 18 registers (18x32 bit memory array). It is a Manhattan-routed grid of streets and avenues, with a regular distribution of memory nodes and logic gates. We divide the grid into three sections. The first 7 columns (on the left) are the registers that hold the values to calculate **T2=Σ0(*a*)+Maj(*a,b,c*)**, while the last 10 columns (on the right) are used to calculate **T1=Σ1(*e*)+Ch(*e,f,g*)+Σ2(h,ki,wi).** Registers **1, 2, 3** of **Σ0** are connected to form a three-input adder. The three registers **9,10,11** of **Σ1** are connected to form a three-input adder also. A third adder **Σ2**, is formed with registers **16, *17, 18.*** In between the two sections, we place one column for input ***d.***



We will divide a single round into three stages. In the first stage we send **a** into columns **Σ0,** and **Maj.** Send **b,c** into column **Maj** as well. Simultaneously, in the right side, send **e** into the three registers of **Σ1** and **Ch**, and send **f,g** into **Ch.** Compute the three additions **Σ0, Σ1, Σ2.** The results will be stored in columns **1, 9, 16,** respectively when their stopping conditions are met.

The second stage will be for computing **T1** and **T2.** To do this, we send the contents of register **7 (Maj)** to register **2.** On the right side, we send the data from register **15 (Ch)** to **10** and register **16 (h)** to **11.** Now we can carry out a second round of addition in **Σ0** and **Σ1** and the results, **T1** and **T2** are stored in registers **9** and **1,** respectively.

We continue to the third stage. Copy **T1** from register **9** to register **2,** and move **d** from register **8** to register **10.** Now, if we execute addition at **Σ0** and **Σ1** we will have **T1+T2** saved in register **1,** and **d+T1** saved in register **9.** To reset for the next iteration, we make the update **a=T1+T2, b=a, c=b, d=c, e=d+T1, f=e, g=f, h=g.**

THANK **YOU!**

www.arrayarchitecture.com

Juan Pablo Ramirez

Architect, Founder & CEO

juan.ramirez@arrayarchitecture.com

Pablo César Vázquez Estrella

Co-Founder & Chief Financial Officer

pablo.vazquez@arrayarchitecture.com