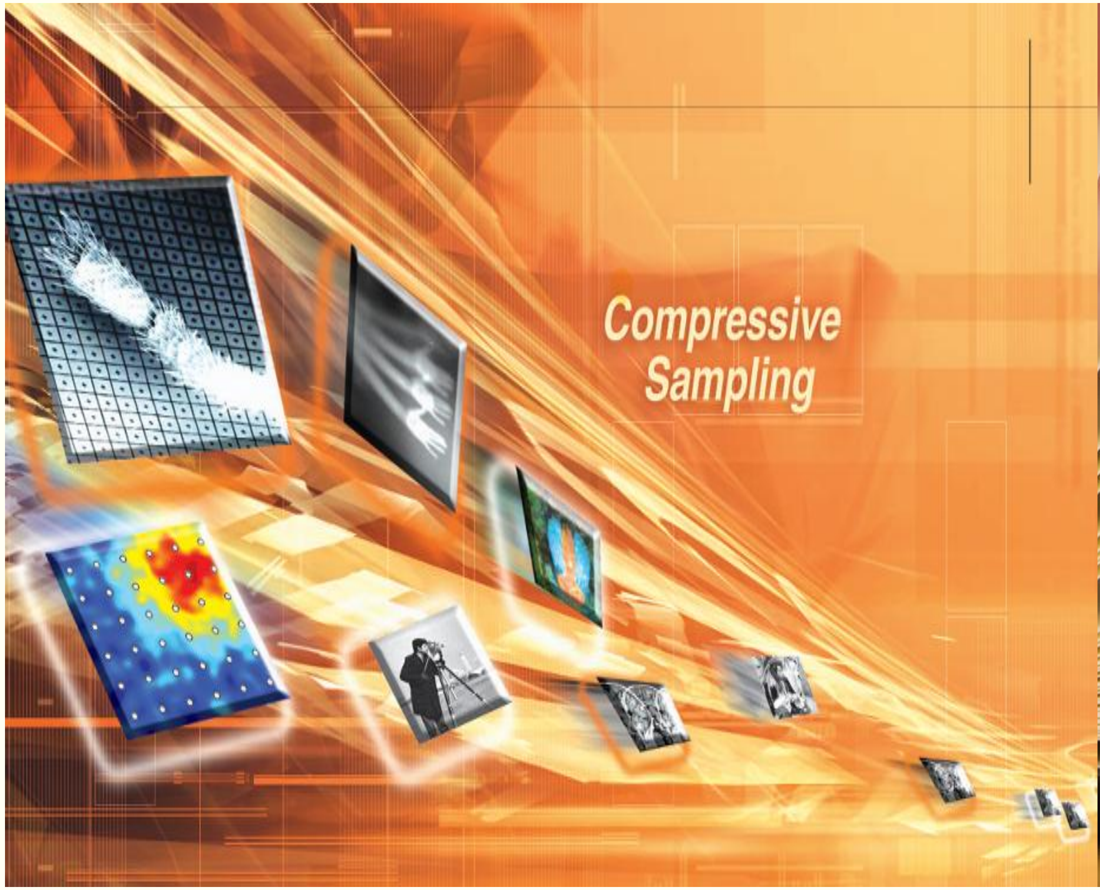
EFFICIENT SPARSE RECOVERY IN COMPRESSIVE SENSING: OMP & AMP WITH ADVANCED SORTING TECHNIQUES



-BY
Himanshu Kumar (122201041)
Vaibhav (122201016)

Introduction & Motivation

What is Compressive Sensing?

Compressive Sensing (CS) is a signal acquisition paradigm that enables the recovery of sparse or compressible signals from far fewer measurements than traditional methods like Nyquist sampling require. CS relies on two fundamental principles:

- **Sparsity**: Many natural signals can be represented using only a few significant components in an appropriate basis.
- **Incoherence**: The sensing mechanism must be designed so that sparse signals appear spread out when sampled.

Why It Matters:

CS allows for efficient data acquisition and signal recovery in scenarios where taking a large number of measurements is impractical or costly. This is particularly useful in applications like medical imaging (MRI), sensor networks, and image compression

Applications of CS:

- **Wireless Communications** – Reducing data overhead in sensor networks.
- **Astronomical Imaging** – Efficient data collection from telescopes.
- **Medical Imaging** (MRI, CT scans) – Faster scans with fewer measurements.

Big-Picture Importance of *OMP & AMP* in Compressive Sensing

Role in Compressive Sensing (CS):

- **OMP (Orthogonal Matching Pursuit)** and **AMP (Approximate Message Passing)** are critical for **sparse recovery**, allowing efficient signal reconstruction from fewer samples.
- These algorithms enable **fast, accurate, and scalable** recovery, making them vital in applications like medical imaging (MRI), wireless communication, and remote sensing.

Architectural Considerations:

- **OMP:** Requires intensive **correlation calculations** and iterative updates, making it computationally expensive. It benefits from **hardware parallelism** to speed up execution.
- **AMP:** Leverages **Bayesian-inspired message passing**, reducing computational complexity and improving efficiency over large datasets. Its iterative nature makes it well-suited for **hardware acceleration** on FPGAs.

Our Focus:

Our work focuses on the **Orthogonal Matching Pursuit (OMP)** and **Approximate Message Passing (AMP)** algorithms for sparse recovery. These methods help reconstruct signals efficiently from compressed measurements.

OMP ALGORITHM

The OMP algorithm iteratively estimates the original signal x using the measurement matrix Φ and the measured vector y. At each iteration, the column of Φ most correlated with the residual r is selected, and its contribution is removed.

Step-1 : Correlation Computation

The correlation vector w is computed as:

$$w = \Phi^T r_{i-1}$$

The index λ_i of the maximum absolute value in w determines the selected column.

Step-2 : Signal Estimation

The estimated signal $\tilde{x}_i$ is obtained by solving:

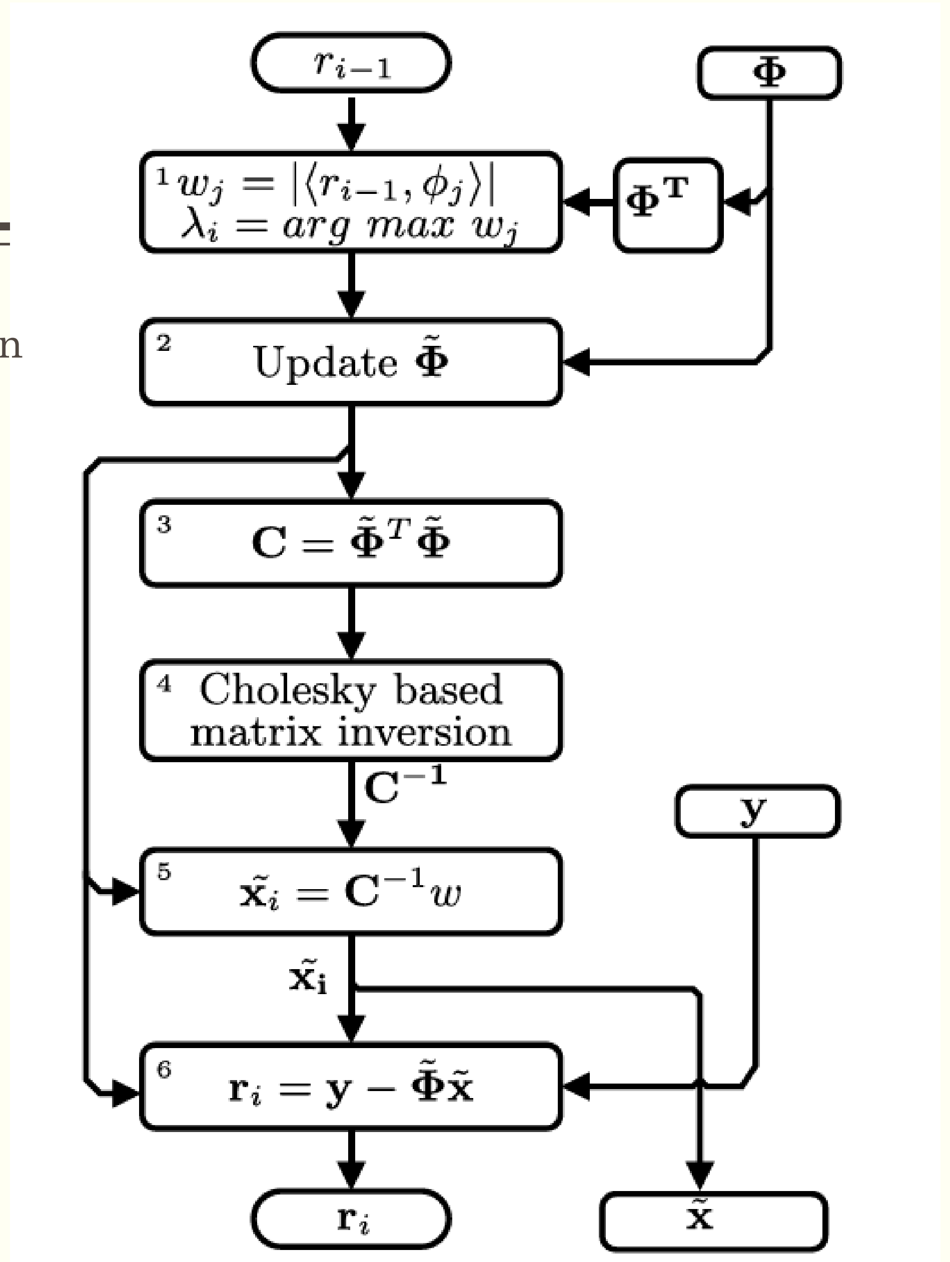
$$y = \tilde{\Phi}\tilde{x}$$

where $\tilde{\Phi}$ consists of the selected columns of Φ. The Moore–Penrose pseudoinverse is used:

$$\tilde{\Phi}^{\dagger} = (\tilde{\Phi}^T\tilde{\Phi})^{-1}\tilde{\Phi}^T$$

which yields:

$$w = C\tilde{x}, \quad C = \tilde{\Phi}^T\tilde{\Phi}$$



(Fig-1:- Flow graph of one iteration of OMP algorithm)

Step-3 : Matrix Inversion via Modified Cholesky Factorization

The matrix C is decomposed as:

$$C = LDL^T$$

where L is a lower triangular matrix and D is a diagonal matrix. The elements of L and D are computed as:

$$L_{i,j} = \frac{1}{D_{j,j}} \left(C_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k} D_{k,k} \right), \quad i > j$$

$$D_{i,i} = C_{i,i} - \sum_{k=1}^{i-1} L_{i,k}^2 D_{k,k}$$

Inverse of C

The inverse of C is found as:

$$C^{-1} = (L^{-1})^T D^{-1} L^{-1}$$

where D^{-1} is the inverse of its diagonal components, and L^{-1} is computed iteratively as:

$$L_{i,j}^{-1} = -\sum_{k=j}^{i-1} L_{i,k} L_{k,j}^{-1}, \quad i > j$$

Step-4 : Residual Update

The residual is updated as:

$$r_i = y - \tilde{\Phi} \tilde{x}$$

Functions		Multiplication	Addition/subtraction	Comparison	Negate	Division
Function-1		NKm	$N(K-1)m$	0	0	0
Function-2		0	0	$(N-1)m$	0	0
Function-3		$K(m+1)m/2$	$(K-1)(m+1)m/2$	0	0	0
Function-4	L	$(m-1)m/2$	0	0	0	0
	D	$(m^3-m)/6$	$(m^3-m)/6$	0	0	0
	D^{-1}	0	0	0	0	m
	L^{-1}	$(m^3-3m^2+2m)/6$	$(m^3-3m^2+2m)/6$	0	$m(m-1)/2$	0
	C^{-1}	$(2m^3+3m^2-5m)/6$	$(m^3-m)/6$	0	0	0
	Subtotal	$(4m^3+3m^2-7m)/6$	$(m^3-m^2)/2$		$m(m-1)/2$	m
Function-5		$(2m^3+3m^2-5m)/6$	$(m^3-m)/6$	0	0	0
Function-6		$Km(m+1)/2$	$K(m+1)m/2$	0	0	0

[Table :- Computation Complexity Of OMP Algorithm (m degree of Sparsity, K size of Measurement vector & N no. of samples)]

References (Literature Review) :-
For OMP, we followed from [1 to 3]

https://docs.google.com/spreadsheets/d/1iAuvE4Xed1vjkgfS1Oa_osTjAYgSEFPLsNnapttr0lM/edit?usp=sharing

MATLAB Implementation of OMP

```matlab
function x_hat = OMP3(y, A, N)
    % Orthogonal Matching Pursuit (OMP) for sparse signal recovery

    [m, n] = size(A);
    x_hat = zeros(n, 1);
    residual = y;
    support_set = [];

    for iter = 1:N
        tic;
        % Compute correlations
        correlations = zeros(n, 1);
        for j = 1:n
            sum_val = 0;
            for i = 1:m
                sum_val = sum_val + A(i, j) * residual(i);
            end
            correlations(j) = sum_val;
        end
        toc;

        tic;
        % Select index with highest correlation manually
        max_val = abs(correlations(1));
        idx = 1;
        for j = 2:n
            if abs(correlations(j)) > max_val
                max_val = abs(correlations(j));
                idx = j;
            end
        end
        toc;

        tic;
        % Update support set
        support_set = [support_set, idx];
        A_selected = A(:, support_set);
        toc;

        tic;
        % Solve least-squares problem
        AtA_inv = my_inv(A_selected' * A_selected);
        x_temp = AtA_inv * (A_selected' * y);
        toc;

        tic;
        % Update residual
        residual = y - A_selected * x_temp;
        toc;

        if norm(residual) <= 1e-6
            break;
        end
    end

```

```matlab
    for i = 1:length(support_set)
        x_hat(support_set(i)) = x_temp(i);
    end
end

function invM = my_inv(M)
    % Computes matrix inverse using Cholesky decomposition
    L = chol(M, 'lower');
    invL = L \ eye(size(M, 1));
    invM = invL' * invL;
end
```

For

y = [0; 2; 3; 5];
A = [1 0 1 0 0 1;
0 1 1 1 0 0;
1 0 0 1 1 0;
0 1 0 0 1 1];
N=6;

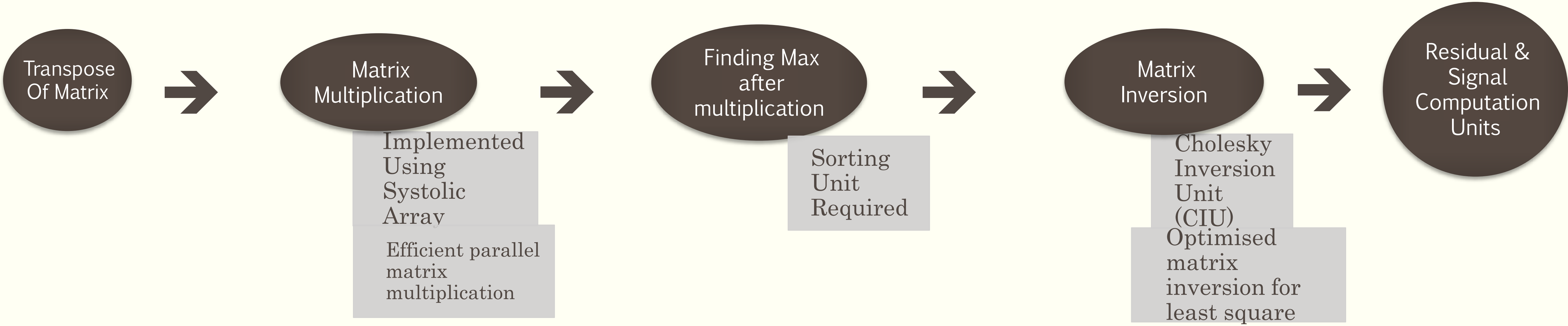
Output:

```
Elapsed time is 0.001022 seconds.
Elapsed time is 0.000196 seconds.
Elapsed time is 0.001199 seconds.
Elapsed time is 0.000206 seconds.
Elapsed time is 0.000254 seconds.
Elapsed time is 0.000166 seconds.
Elapsed time is 0.000098 seconds.
Elapsed time is 0.000031 seconds.
Elapsed time is 0.000062 seconds.
Elapsed time is 0.000170 seconds.
Elapsed time is 0.000208 seconds.
Elapsed time is 0.000123 seconds.
```

x_hat =

```
         0
    2.0000
         0
         0
    3.0000
         0
```

Hardware Blocks for **OMP** Algorithm & **Comparison b/w OMP & AMP**



Aspect	**OMP Algorithm**	**AMP Algorithm**
Computation Approach	Iterative selection and least squares update	Approximate message passing with iterative refinement
Matrix Operations	Requires matrix inversion in each iteration	Avoids matrix inversion, reducing complexity
Sorting - Common Step in Both OMP and AMP		
Sorting Requirement	Full sorting (Finding MAX at each iteration)	Partial sorting (Selecting a subset of largest elements)
Computational Complexity	High (Sorting entire dataset, matrix inversion)	Lower (Sorting only relevant subset, no inversion)
Hardware Demand	Requires full sorting network, costly matrix inversion	Needs only partial sorting unit, reducing resource usage
Performance	Slower due to full sorting and matrix inversion	Faster due to reduced sorting and simpler computations
Overall Efficiency	More expensive in hardware	More efficient and hardware-friendly

SORTING ALGORITHM

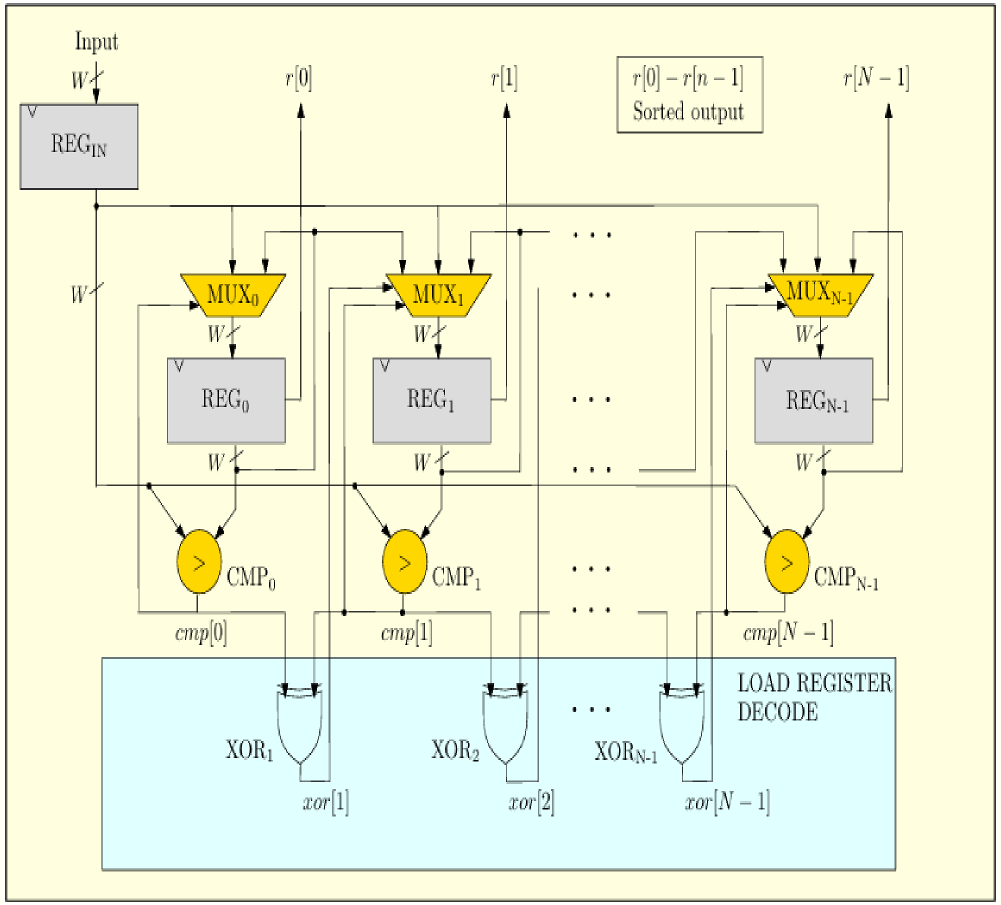
Serial Sorting (From SORT-N Algorithm)

- We **started researching** sorting algorithms and came across the **SORT-N** algorithm.
- We then explored **XSORT-N**, an improved version of SORT-N that sorts N samples in **N clock cycles** with **lower delay and better performance**.

How it works:

- SORT-N uses **comparators, a leading-one detector (LOD) and a counter** to determine the insertion position.
- XSORT-N improves SORT-N by replacing LOD with a simpler **XOR-based logic**, reducing delay.

- **Limitation:** Despite XSORT-N's improvements, **both methods are still serial sorting**, requiring **N cycles**, which is inefficient for large datasets.
- To improve **performance,** we will explore **parallel sorting** methods.



(Fig-2 :- XSORT – N Architecture)

Parallel Sorting

Serial sorting methods like **SORT-N** and **XSORT-N** take **N clock cycles**, making them slow for large datasets.

We explored **parallel sorting algorithms**, such as **bitonic** and **odd-even merge sorting networks**, which sort in **logarithmic time** $\mathcal{O}(\log^2 N)$.

How it works:

Uses **compare-and-exchange (CAE) blocks** to process multiple elements in parallel.

Sorting happens in **stages**, significantly reducing sorting time.

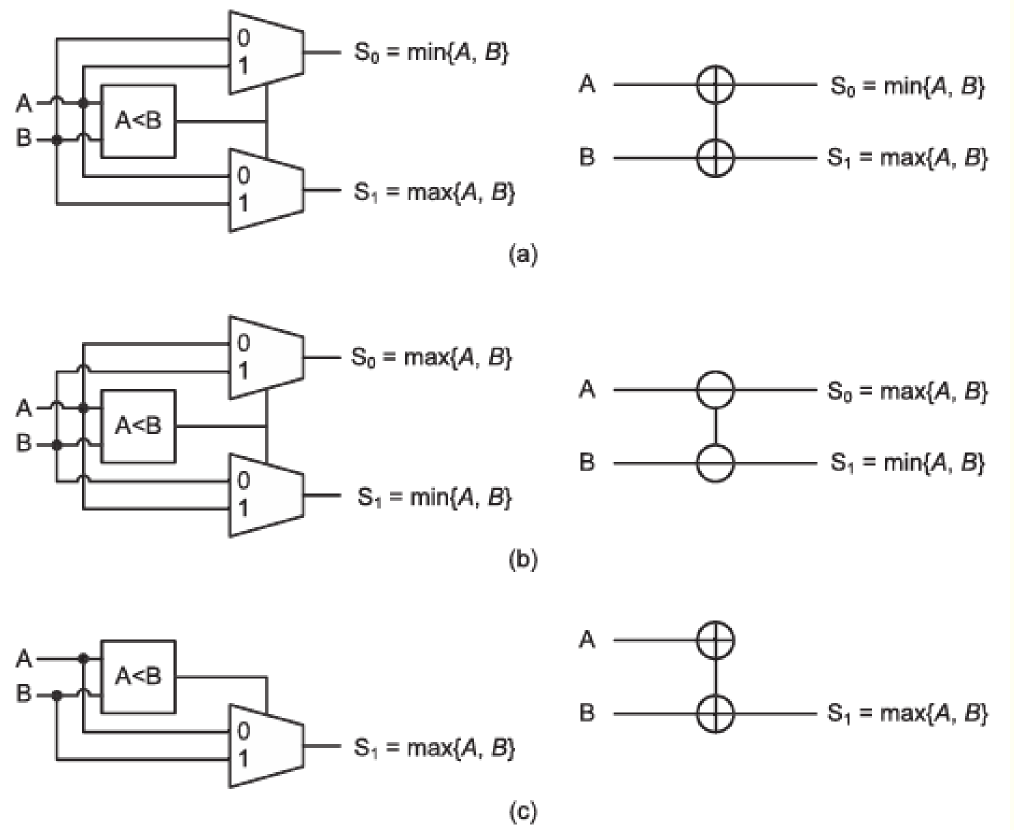
Advantages of Parallel Sorting:

High throughput: Multiple elements are processed simultaneously.

Lower latency: Sorting time scales efficiently with input size.

Optimized for hardware: Suitable for **FPGAs and VLSI** applications.

We will now focus on implementing a parallel sorting approach for better performance.



[Fig-3 :- Compare-And-Exchange (CAE) blocks & their representation]

References (Literature Review) :-
For Sorting, we followed from [4 to 8]

https://docs.google.com/spreadsheets/d/1iAuvE4Xed1vjkgfS1Oa_osTjAYgSEFPLsNnapttr0lM/edit?usp=sharing

Bitonic-Merge Sort

. **Concept:**

Bitonic sorting merges an **ascending** and **descending** sequence into a sorted sequence.

Structure:

A **K-input bitonic merging unit (BM-K)** has $\log_2(K)$ stages.

Each stage contains $\frac{K}{2}$ Compare-and-Exchange (CAE) blocks.

Formulas:

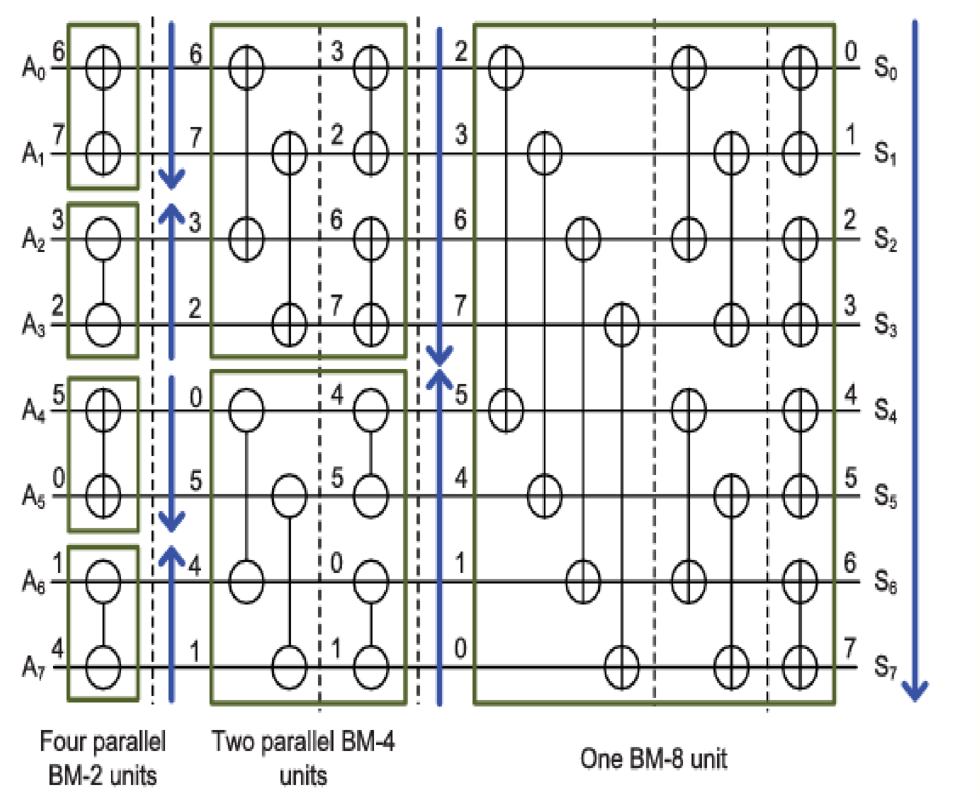
Number of CAE Stages: $\log_2(N) \times \frac{(\log_2(N)+1)}{2}$

Total CAE Blocks: $\frac{N \times \log_2(N) \times (\log_2(N)+1)}{4}$

Example:

A 16-input bitonic sorter requires **10 CAE stages** and **80 CAE blocks**.

A 256-input bitonic sorter requires **36 CAE stages** and **4,608 CAE blocks**.



[Fig-4 :- Bitonic sorting of 8 unsorted elements using BM-2 (4) → BM-4 (2) → BM-8(1)]

Odd-Even-Merge Sort

Concept:

Odd-even merge sorting recursively merges **two ascending sequences** into a sorted sequence.

Structure:

A **K-input odd-even merging unit (OEM-K)** has $\log_2(K)$ stages.

Each stage contains between $\frac{K}{4}$ and $\frac{K}{2}$ CAE blocks.

Formulas:

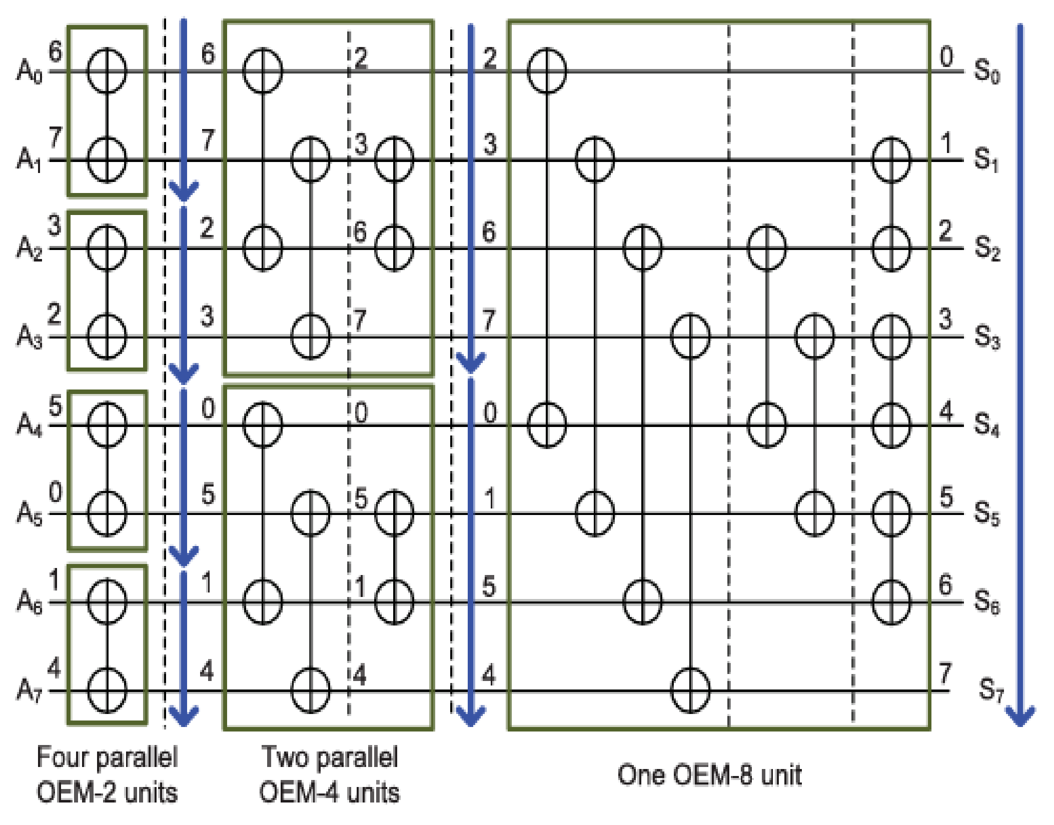
Number of CAE Stages: $\log_2(N) \times (\log_2(N) + 1)$

Total CAE Blocks: $\frac{N}{4} \times \log_2(N) \times (\log_2(N) - 1) + N - 1$

Example:

An 8-input odd-even merge sorter has **6 CAE stages** and **19 CAE blocks**.

A 256-input sorter requires **36 CAE stages** and **3,839 CAE blocks**.



[Fig-5 :- OEM sort of 8 unsorted elements using OEM-2 (4) → OEM-4 (2) → OEM-8(1)]

Comparison between BM & OEM Sort :-

Feature	Bitonic Sorting (BM)	Odd-Even Merge Sorting (OEM)
Latency (CAE Stages)	$\log_2(N) \times \frac{(\log_2(N)+1)}{2}$	$\log_2(N) \times (\log_2(N)+1)$
CAE Blocks	$\frac{N \times \log_2(N) \times (\log_2(N)+1)}{4}$	$\frac{N}{4} \times \log_2(N) \times (\log_2(N)-1) + N - 1$
Difference in CAE Blocks	$2^{n-1} \times (n-2) + 1$	

PARTIAL SORTING :-

. **Concept:**

Partial sorting extracts only the **M largest values** from **N inputs** instead of fully sorting the dataset.

Reduces **latency** and **hardware complexity** by discarding smaller values early.

Sorting Methods:

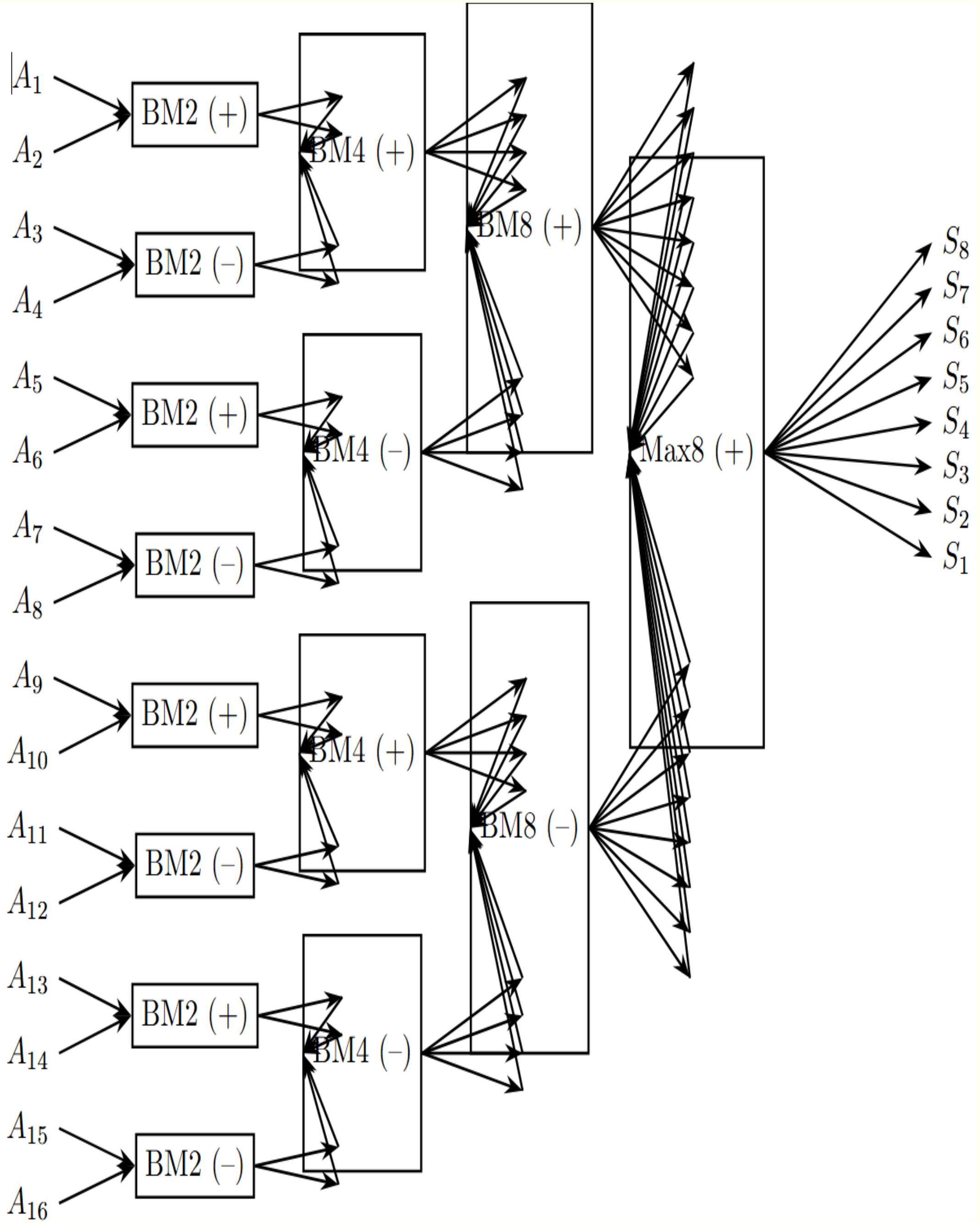
Bitonic Merge (BM) Sorting Network

Odd-Even Merge (OEM) Sorting Network

Key Concept:

Uses specialized **BM-2^{k+1}-to-2^k** and **OEM-2^{k+1}-to-2^k** merging units.

Advantage: Reduces the total number of **CAE stages** and **CAE blocks**.



[Fig-6 :- Partial sorting in BM for 16-to-8]

Partial Sorting In BM & OEM

. Partial Sorting in Bitonic Merge (BM)

Selects **M largest values** from **N inputs**.

Uses **BM-2^{k+1}-to-2^k** merging units to extract top M values.

Partial Sorting in Odd-Even Merge (OEM)

Uses **OEM-2^{k+1}-to-2^k** merging units.

Designed to handle **two ascending sequences**.

Formulas:

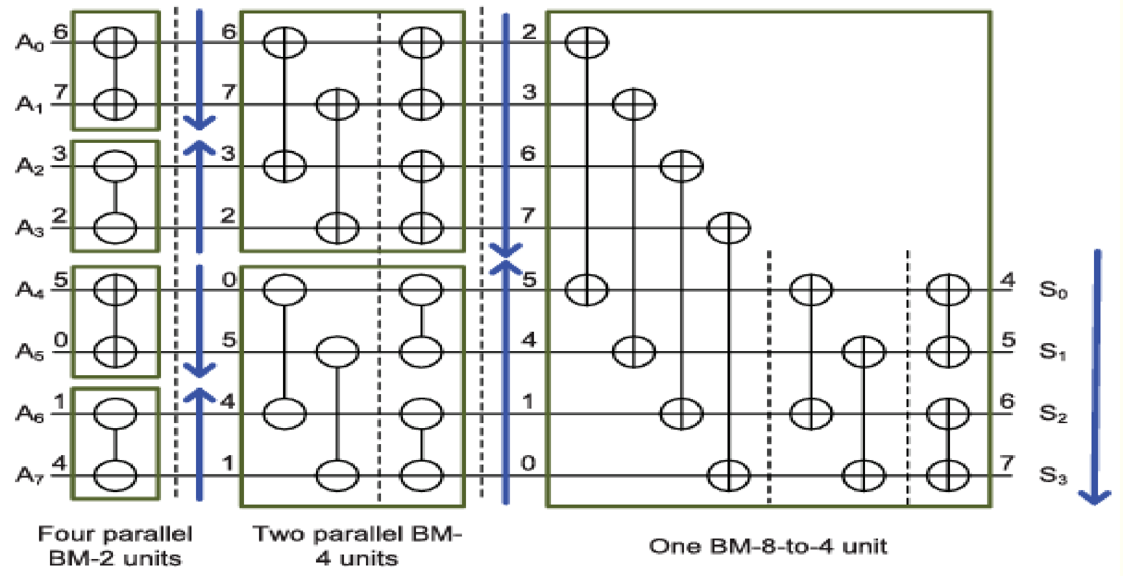
Number of CAE Stages:

$$\text{CAE Stages} = \frac{2n - m \times (m+1)}{2}$$

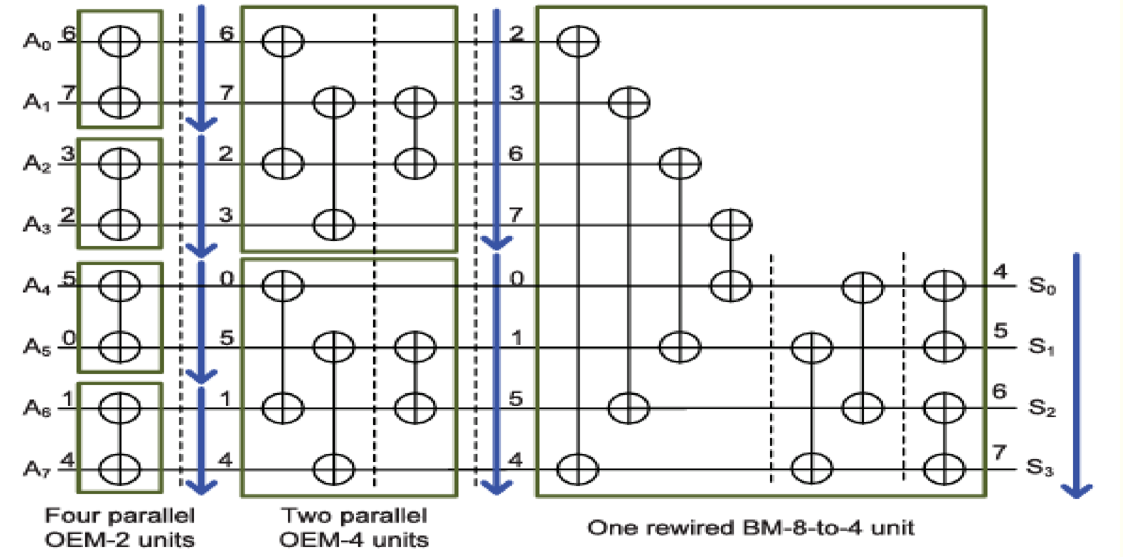
Number of CAE Blocks:

$$\text{CAE Blocks (BM)} = (m(m+3)+4) \times 2^{(n-2)} - 2^{m-1}(m+2)$$

$$\text{CAE Blocks (OEM)} = (m(m+3)+4) \times 2^{(n-2)} - m \times 2^{(n-1)} + (1-2^{-m}) \times 2^{n} - 2^{m-1}(m+2)$$



[Fig-7 :- The CAE network for an 8-to-4 bitonic partial sorting unit with six CAE stages and 20 CAE blocks.]



[Fig-8 :- The CAE network for an 8-to-4 odd-even merge partial sorting unit with six CAE stages and 18 CAE blocks.]

Comparison of Partial Sorting in BM & OEM

. **Example Calculation (N = 1024, M = 512)**

Sorting Method	CAE Stages Formula	CAE Stages (Result)
BM Partial Sorting	$\frac{(2\times10-9)\times(9+1)}{2}$	10
OEM Partial Sorting	$\frac{(2\times10-9)\times(9+1)}{2}$	10

Sorting Method	CAE Blocks Formula	CAE Blocks (Result)
BM Partial Sorting	$(9(9+3)+4)\times 2^{(10-2)} - 2^{9-1}(9+2)$	2304
OEM Partial Sorting	$(9(9+3)+4)\times 2^{(10-2)} - 9\times 2^{(10-1)} + (1-2^{-9})\times 2^{10} - 2^{9-1}(9+2)$	2176

Observations:

Both **BM** and **OEM** have the **same number of CAE stages**.

OEM requires fewer CAE blocks than BM, making it more hardware-efficient.

BM is simpler to implement due to its regular structure, making it preferable in FPGA/VLSI designs.

OEM has more complex wiring, which may increase design difficulty.

Why Bitonic Merge (BM) over Odd-Even Merge (OEM) ?

OEM requires fewer CAE blocks, but BM is **easier to implement in VLSI**.

Due to VLSI efficiency, we proceed with BM for partial sorting.

Investigating Patterns for Partial Sorting:

Partial sorting of large datasets.

Scaling patterns for partial sorting from:

- $N \to N/2$
- $N \to N/4$
- $N \to N/8$

Pattern for $N \to N/2$:

$$\text{BM-2(N/2)} \to \text{BM-4 (N/4)} \to \cdots \to \text{BM-N/2 (2)} \to \text{MAX(N/2)}$$

Since MAX-SET-SELECTION (MAX) does not provide sorted elements in order, we apply complete BM-N/2 sorting.

Example for 64 to 32:

$$\text{BM-2(32)} \to \text{BM-4(16)} \to \text{BM-8(8)} \to \text{BM-16(4)} \to \text{BM-32(2)} \to \text{MAX(32)}$$

Example for 1024 to 512:

$$\text{BM-2(512)} \to \text{BM-4(256)} \to \cdots \to \text{BM-512(2)} \to \text{MAX-512(1)}$$

Pattern for $N \to N/4$ (for $N = 1024$):

$$\text{BM-2(512)} \to \cdots \to \text{BM-512-to-256(2)} \to \text{MAX-256(1)}$$

After reaching BM-N/4, we use 2 blocks of BM-N/2-to-N/4, then apply complete BM-N/4 sorting.

Pattern for $N \to N/8$ (for $N = 1024$):

$$\text{BM-2(512)} \to \cdots \to \text{BM-256-to-128(4)} \to \text{BM-256-to-128(2)} \to \text{MAX-128(1)}$$

After reaching BM-N/8, we use 4 blocks of BM-N/4-to-N/8, then 2 blocks of BM-N/4-to-N/8, followed by complete BM-N/4 sorting.

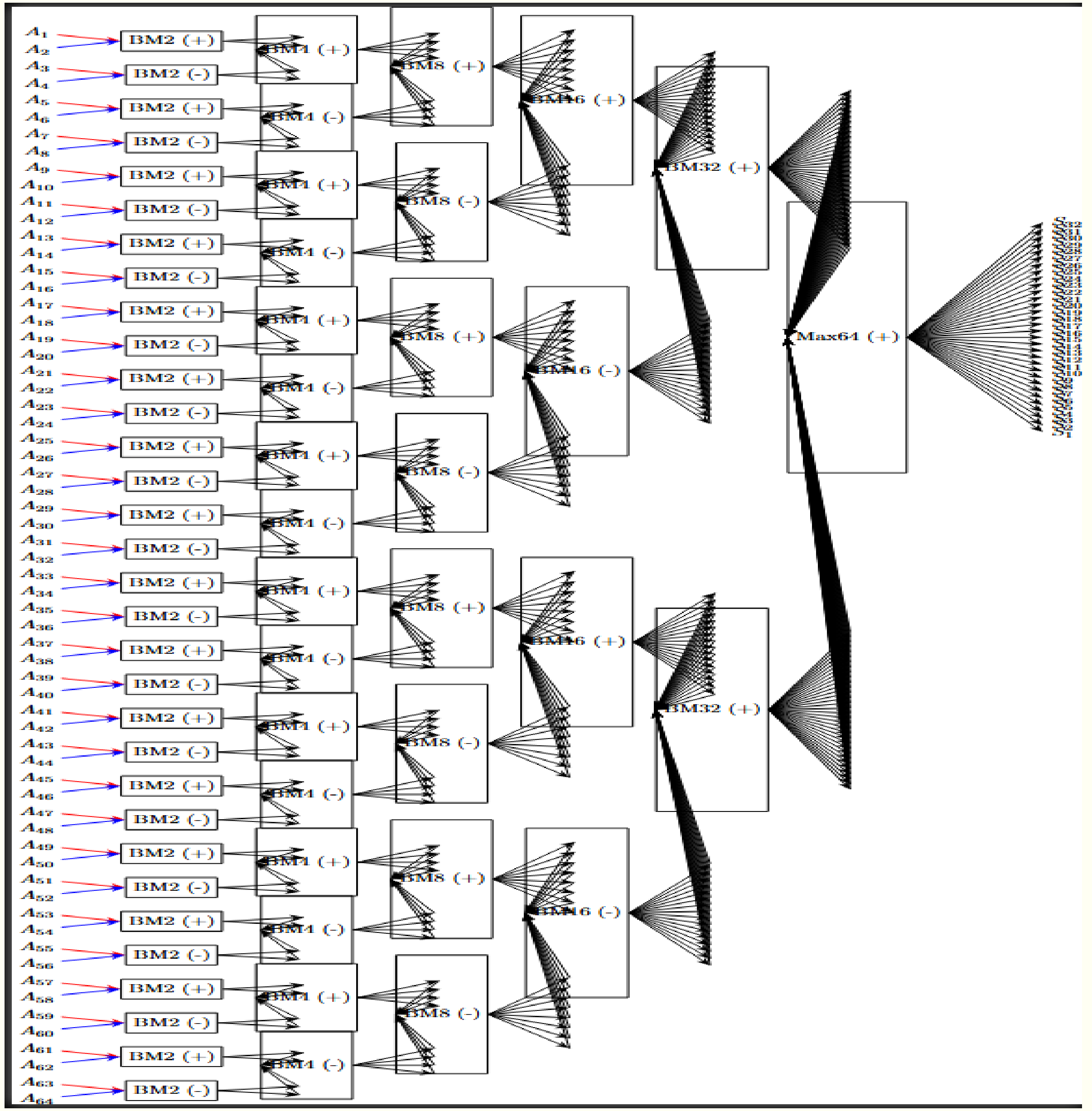


Fig-9 :- 64-to-32 bitonic max-set-selection unit

MATLAB Implementation for Partial Sorting In Bitonic-Merge

```matlab
function psmhe  %partial sort max half bitonic
n = 64;
a = randperm(n);
disp('original array');
disp(a);
a = psmh(a);
disp('max sorted half ');
disp(a(1:n/2));
end

function a = psmh(a) % partial sort part
n = length(a);
h = n / 2;
% first half in ascending order
a = bsort(a, 1, h, true);
% second half in ascending order
a = bsort(a, h+1, h, true);
for i = 1:h
if a(i) < a(n + 1 - i)
t = a(i);
a(i) = a(n + 1 - i);
a(n + 1 - i) = t;
end
end
a = bsort(a, 1, h, true); % sort the top half
end

function a = bsort(a, s, k, asc) %bitonic sort
if k > 1
h = k / 2;
% sort first half in ascending order
a = bsort(a, s, h, true);
% sort second half in descending order
a = bsort(a, s + h, h, false);
a = bmerge(a, s, k, asc);
end
end

function a = bmerge(a, s, k, asc) %bitonic merge
if k > 1
h = k / 2;
for i = s:(s + h - 1)
a = cswap(a, i, i + h, asc);
end
% recursively merging both halves
a = bmerge(a, s, h, asc);
a = bmerge(a, s + h, h, asc);
end
end

function a = cswap(a, i, j, asc) % compare and swap
if asc
if a(i) > a(j)
t = a(i);
a(i) = a(j);
a(j) = t;
end
else
if a(i) < a(j)
t = a(i);
a(i) = a(j);
a(j) = t;
end
end
end
```

Output of MATLAB Code:-

```
>> psmhe
original array
  Columns 1 through 17

    13    41    23     5    16    56    38    49    15    40     1    37    45    53    18    58    61

  Columns 18 through 34

    64    25    62    11    31    20    46    17    52    59     2     6     8    29    26    54    12

  Columns 35 through 51

    47    55    19     7    34    27    42    60    57    63    10    51     9    33    36    44    35

  Columns 52 through 64

    32    24     4     3    50    43    22    28    21    48    30    14    39

max sorted half
  Columns 1 through 17

    33    34    35    36    37    38    39    40    41    42    43    44    45    46    47    48    49

  Columns 18 through 32

    50    51    52    53    54    55    56    57    58    59    60    61    62    63    64
```

FUTURE SCOPE

. **Implementation in Verilog:**

We will implement the **partial sorting algorithm in Bitonic Merge (BM)** using Verilog.

A suitable **hardware architecture** will be designed for efficient implementation.

Optimizations for Area Efficiency:

Instead of using separate stages for $N \to N/2$, $N \to N/4$, and $N \to N/8$, we will explore a **folding architecture**.

This approach allows for **hardware reuse**, reducing area requirements while maintaining performance.

Verification Synthesis:

The Verilog implementation will be verified by **matching the output with MATLAB simulations**.

The code will be **synthesizable**, ensuring it can be implemented on hardware (FPGA/ASIC).