

SIMDRAM: A Framework for Bit-Serial SIMD Processing using DRAM

Computer Architecture, Lecture 24c
Fall 2021

Nastaran Hajinazar*

Sven Gregorio

Joao Ferreira

Geraldo F. Oliveira*

Nika Mansouri Ghiasi

Minesh Patel

Mohammed Alser

Saugata Ghose

Juan Gómez-Luna

Onur Mutlu

Executive Summary

- **Motivation:** Processing-using-Memory (PuM) architectures can effectively perform bulk bitwise computation
- **Problem:** Existing PuM architectures are not widely applicable
 - Support only a limited and specific set of operations
 - Lack the flexibility to support new operations
 - Require significant changes to the DRAM subarray
- **Goals:** Design a processing-using-DRAM framework that:
 - Efficiently implements complex operations
 - Provides the flexibility to support new desired operations
 - Minimally changes the DRAM architecture
- **SIMDRAM:** An end-to-end processing-using-DRAM framework that provides the programming interface, the ISA, and the hardware support for:
 1. Efficiently computing complex operations
 2. Providing the ability to implement arbitrary operations as required
 3. Using a massively-parallel in-DRAM SIMD substrate
- **Key Results:** SIMDRAM provides:
 - 88x and 5.8x the throughput and 257x and 31x the energy efficiency of a baseline CPU and a high-end GPU, respectively, for 16 in-DRAM operations
 - 21x and 2.1x the performance of the CPU and GPU over seven real-world applications

Outline

1. Processing-using-DRAM

2. Background

3. SIMD RAM

Processing-using-DRAM Substrate
Framework

4. System Integration

5. Evaluation

6. Conclusion

Outline

1. Processing-using-DRAM

2. Background

3. SIMD RAM

Processing-using-DRAM Substrate
Framework

4. System Integration

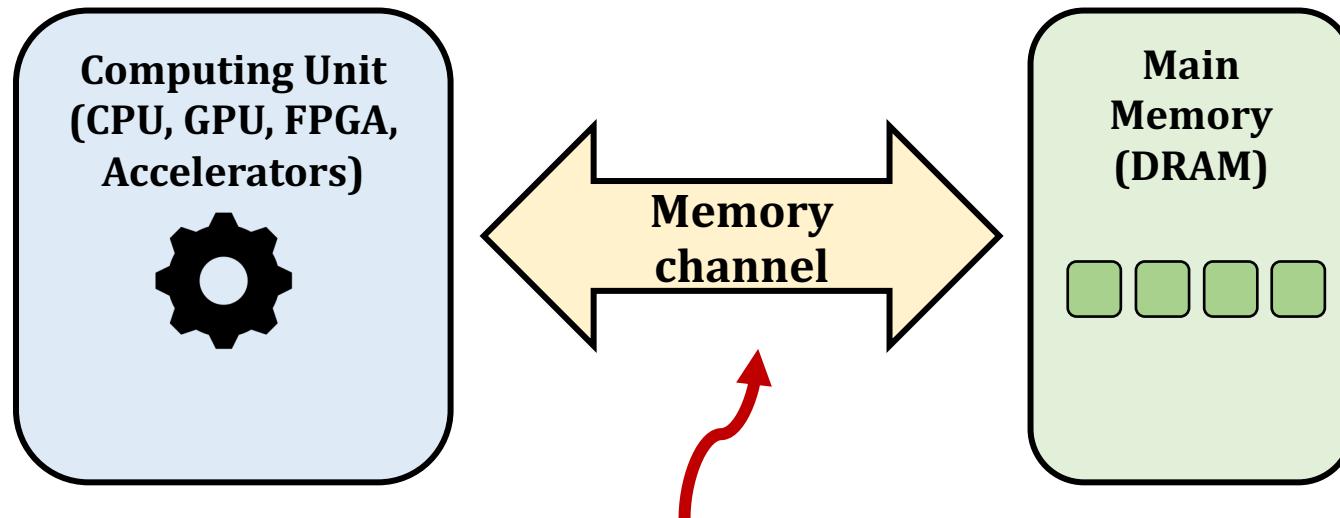
5. Evaluation

6. Conclusion

Data Movement Bottleneck

- Data movement is a major bottleneck

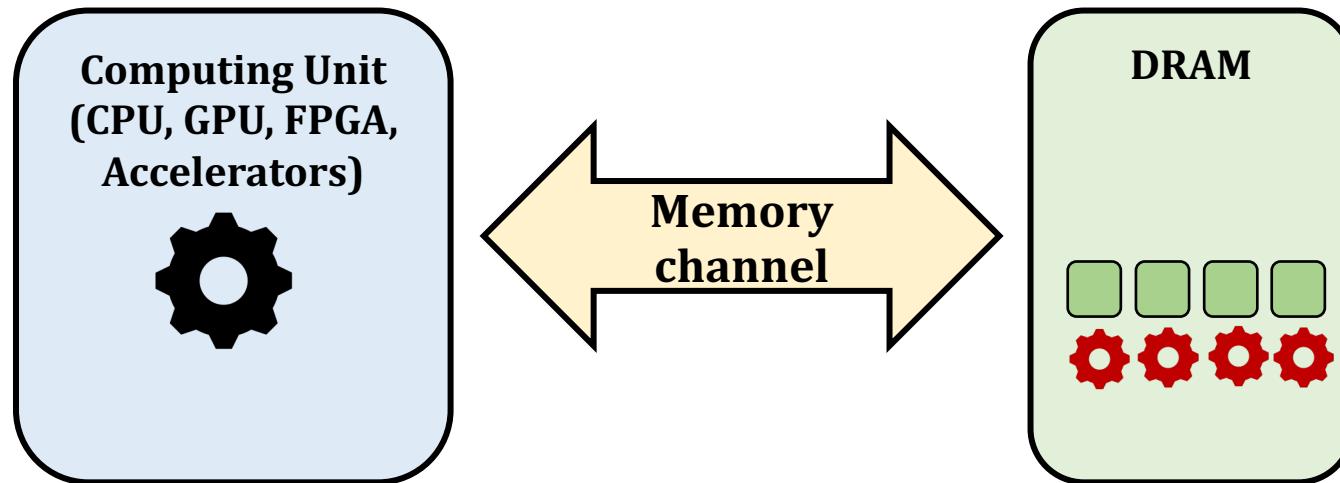
More than **60%** of the total system energy
is spent on **data movement**¹



Bandwidth-limited and power-hungry memory channel

Processing-in-Memory (PIM)

- **Processing-in-Memory:** moves computation closer to where the data resides
 - Reduces/eliminates the need to move data between processor and DRAM



Processing-using-Memory (PuM)

- **PuM:** Exploits analog operation principles of the memory circuitry to perform computation
 - Leverages the **large internal bandwidth** and **parallelism** available inside the memory arrays
- A common approach for **PuM** architectures is to perform **bulk bitwise operations**
 - Simple logical operations (e.g., AND, OR, XOR)
 - More complex operations (e.g., addition, multiplication)

Outline

1. Processing-using-DRAM

2. Background

3. SIMD RAM

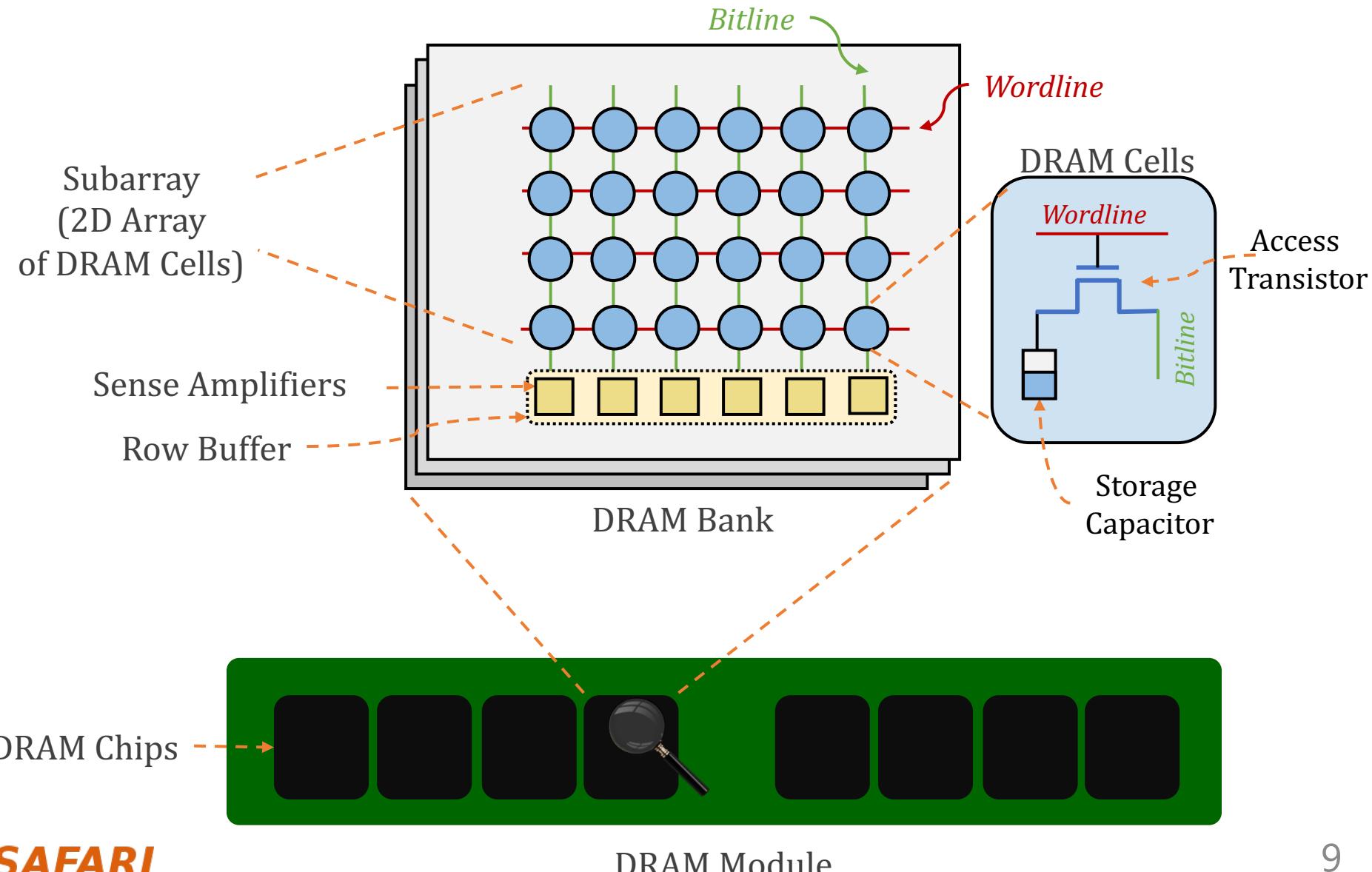
Processing-using-DRAM Substrate
Framework

4. System Integration

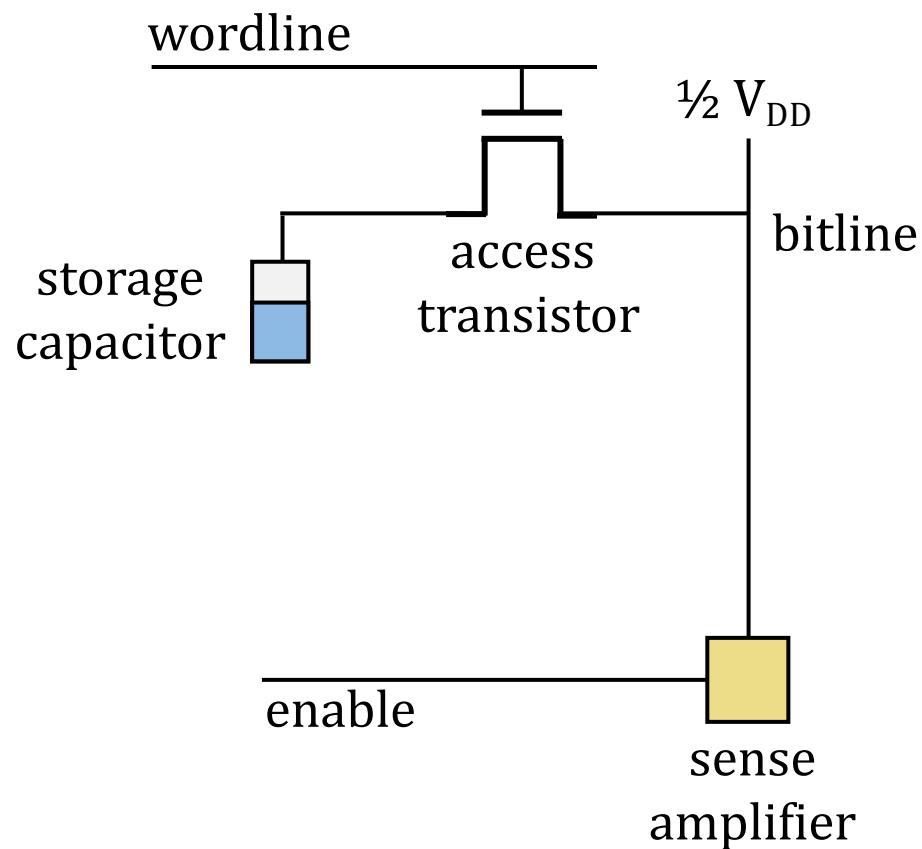
5. Evaluation

6. Conclusion

Inside a DRAM Chip



DRAM Cell Operation

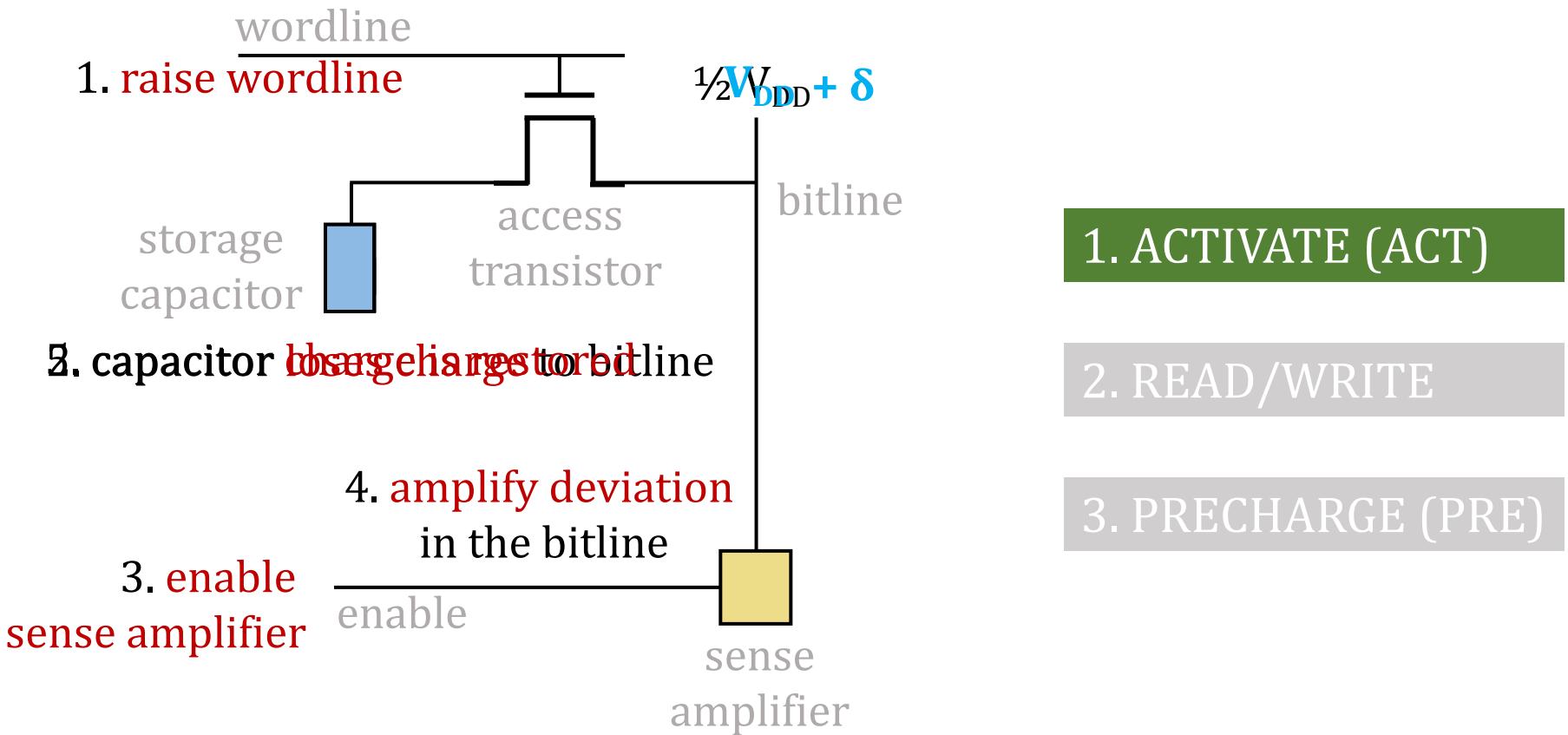


1. ACTIVATE (ACT)

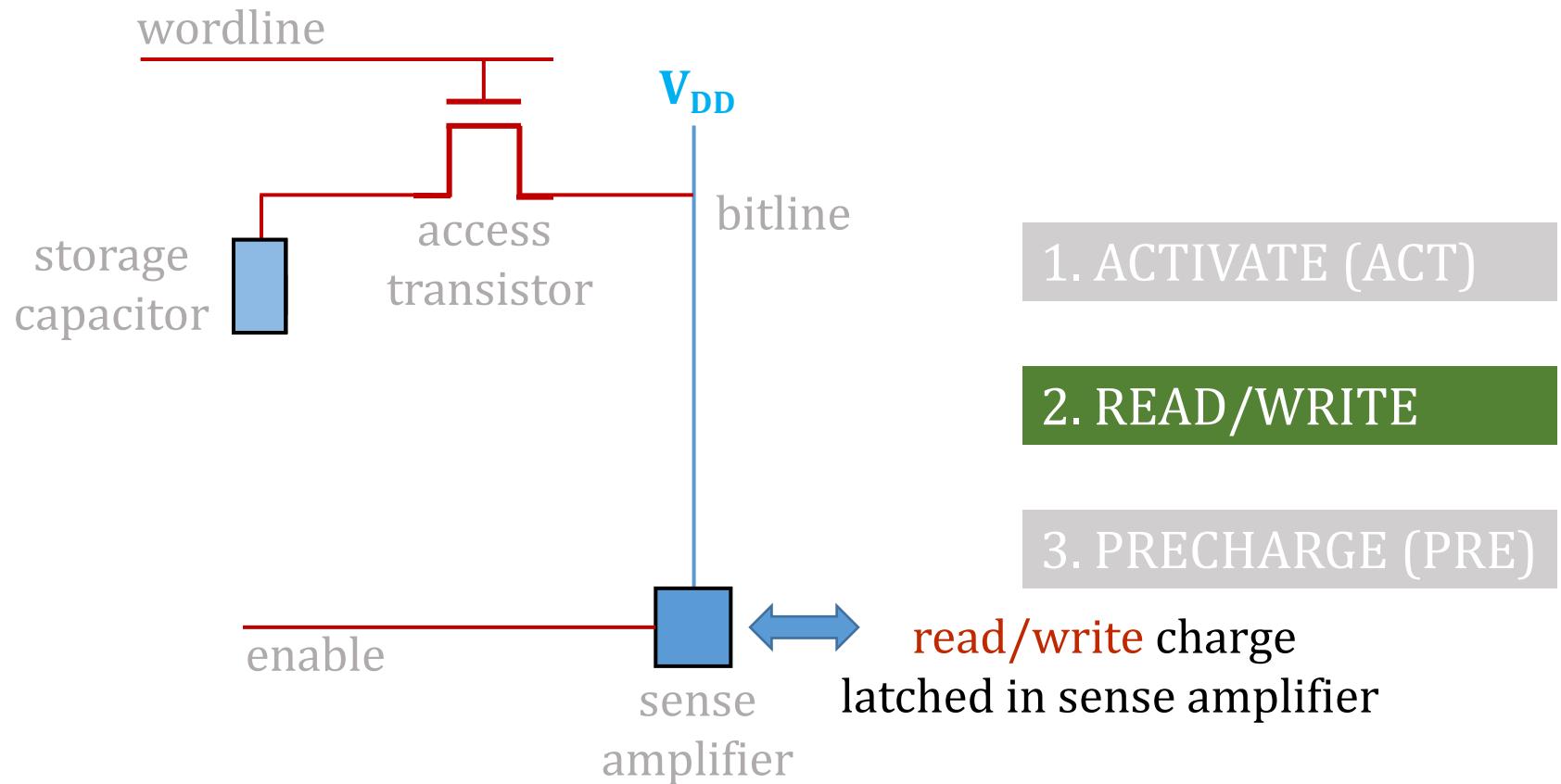
2. READ/WRITE

3. PRECHARGE (PRE)

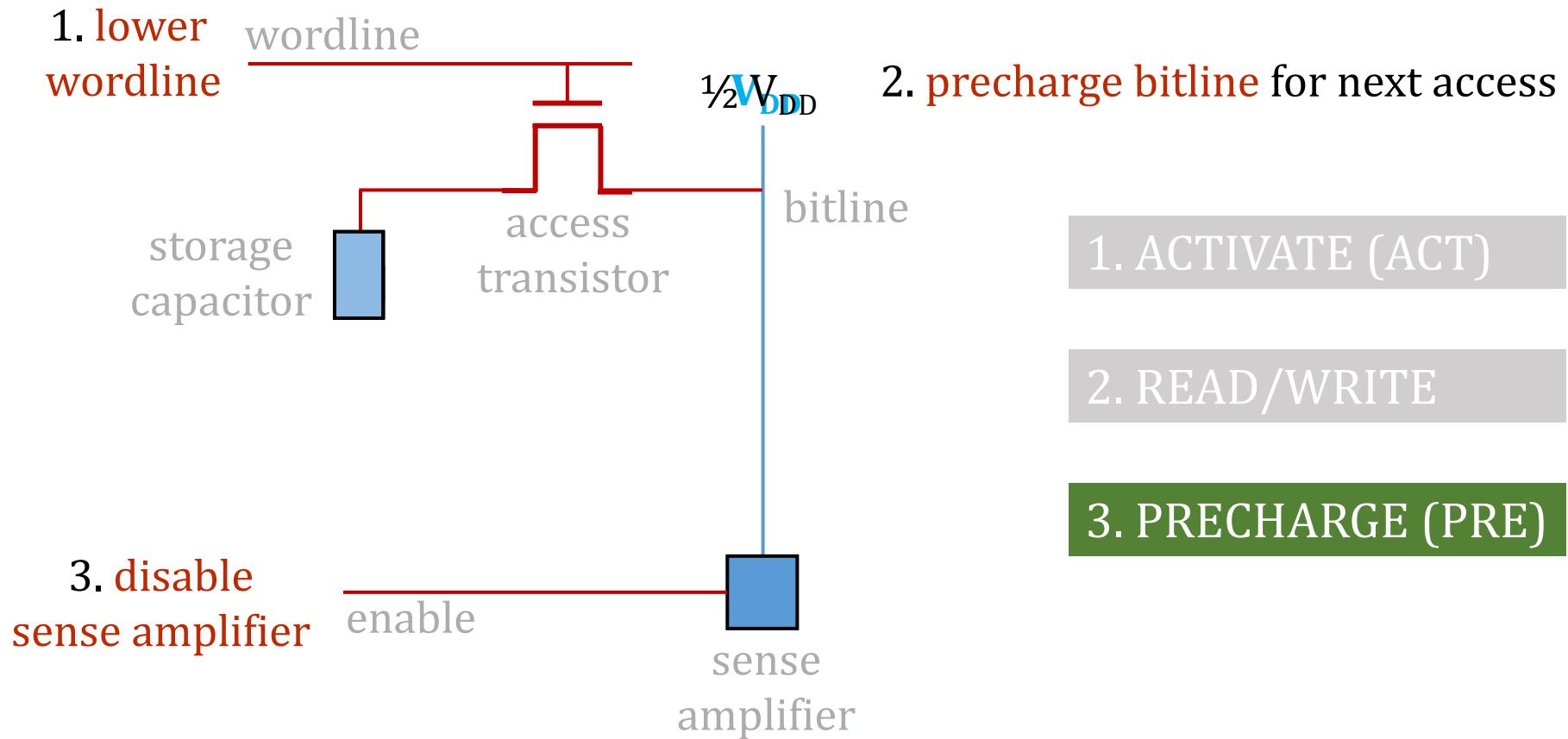
DRAM Cell Operation (1/3)



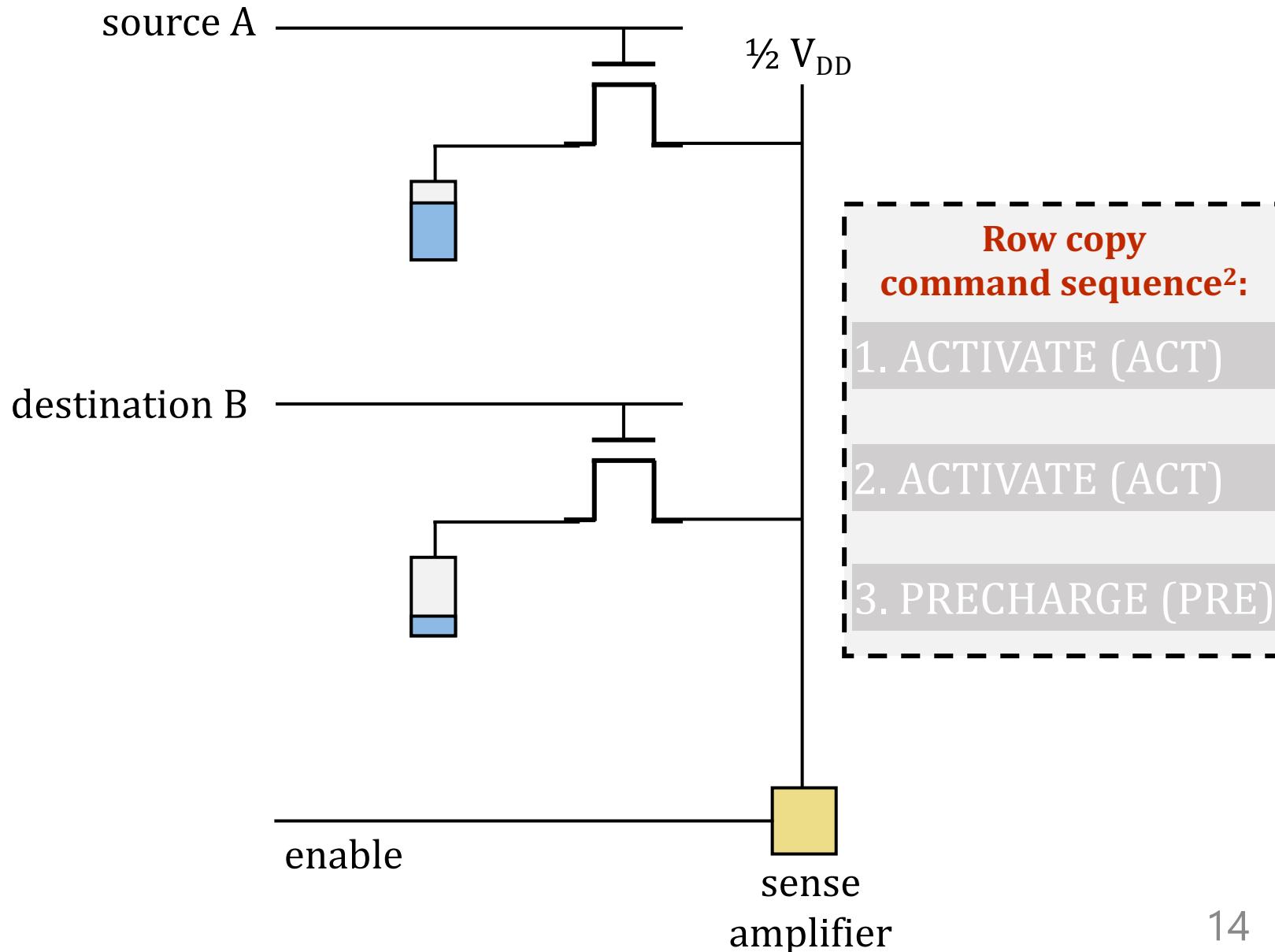
DRAM Cell Operation (2/3)



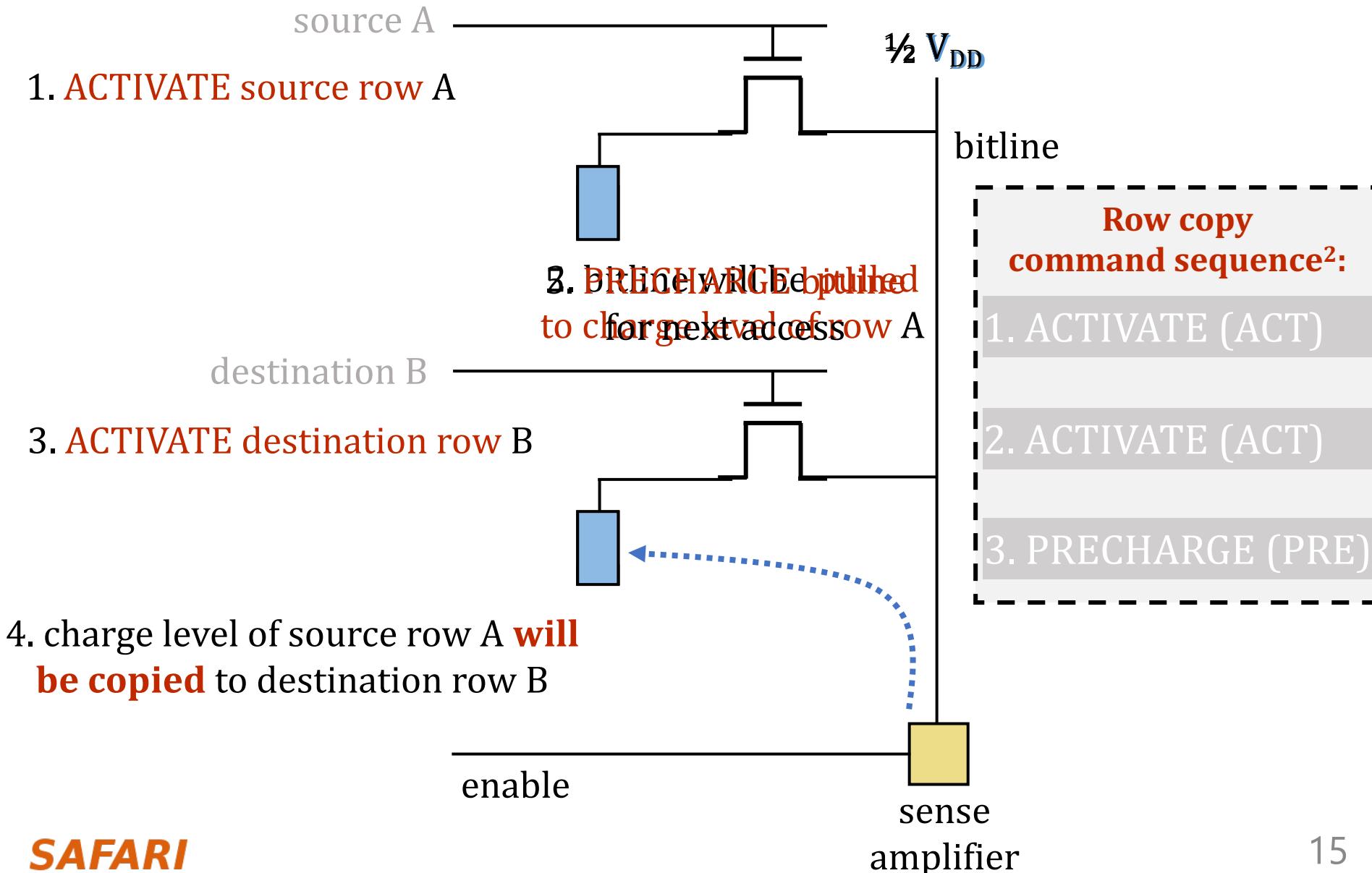
DRAM Cell Operation (3/3)



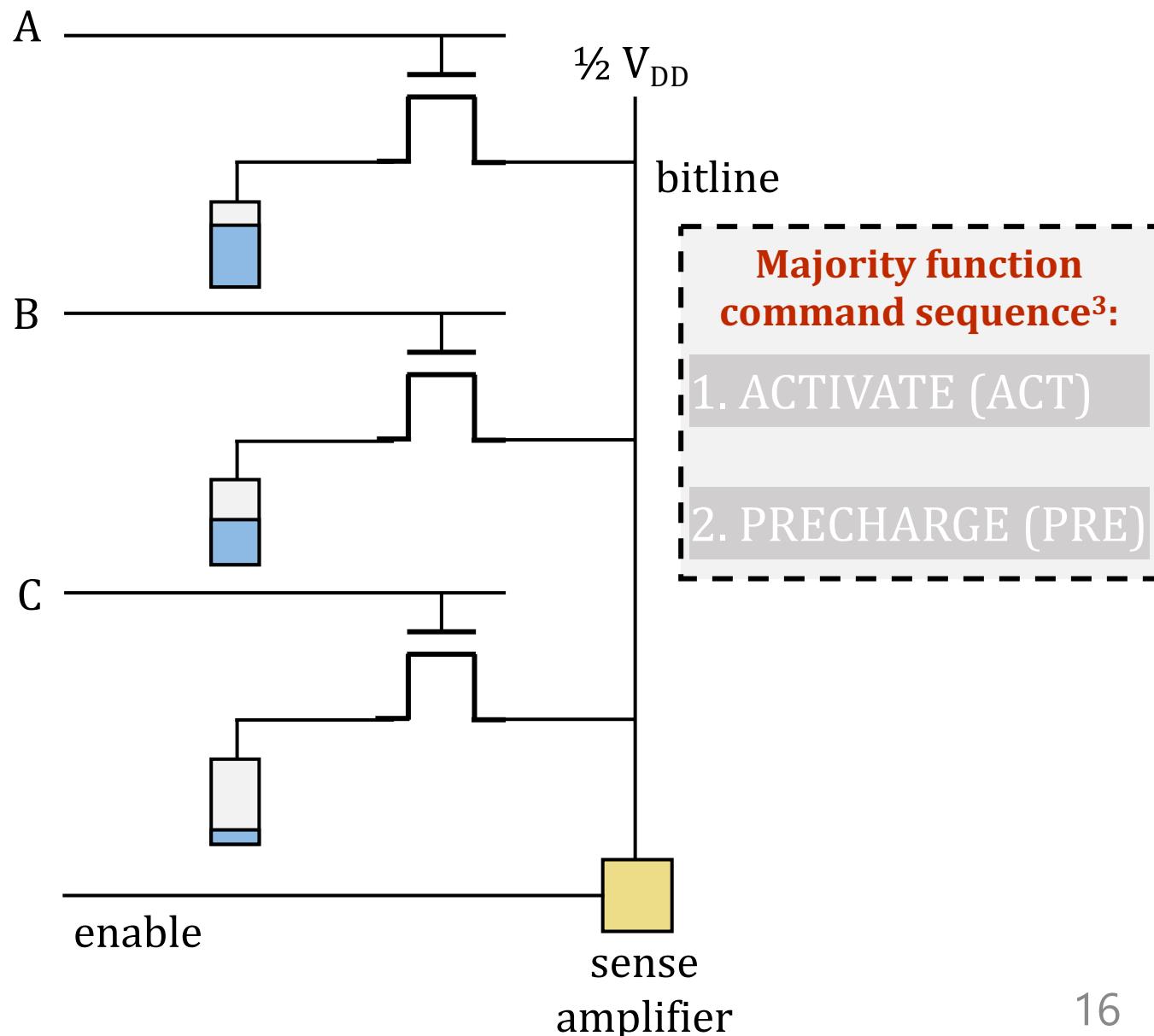
RowClone: In-DRAM Row Copy (1/2)



RowClone: In-DRAM Row Copy (2/2)



Triple-Row Activation: Majority Function



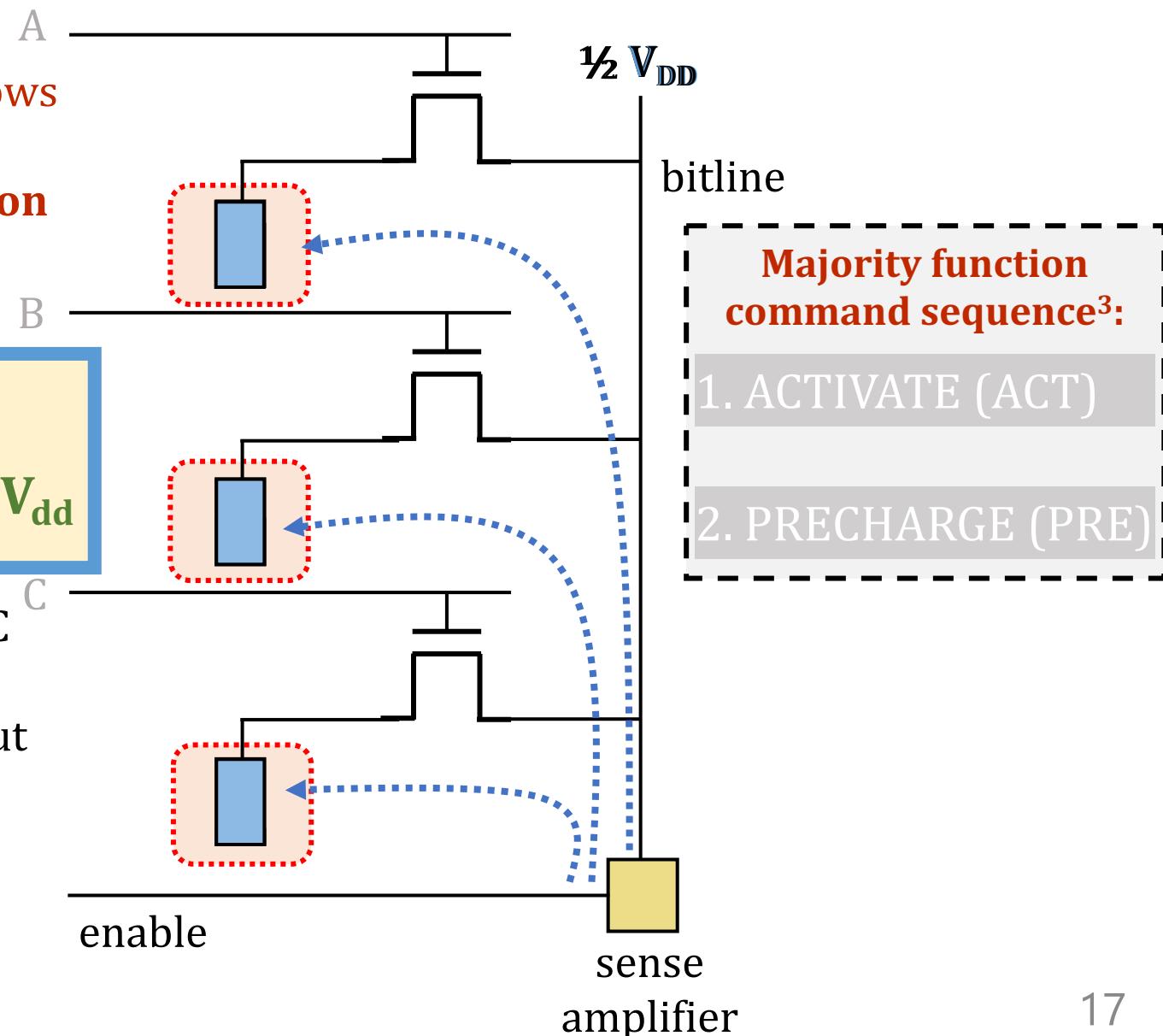
Triple-Row Activation: Majority Function

1. ACTIVATE three rows simultaneously
→ triple-row activation

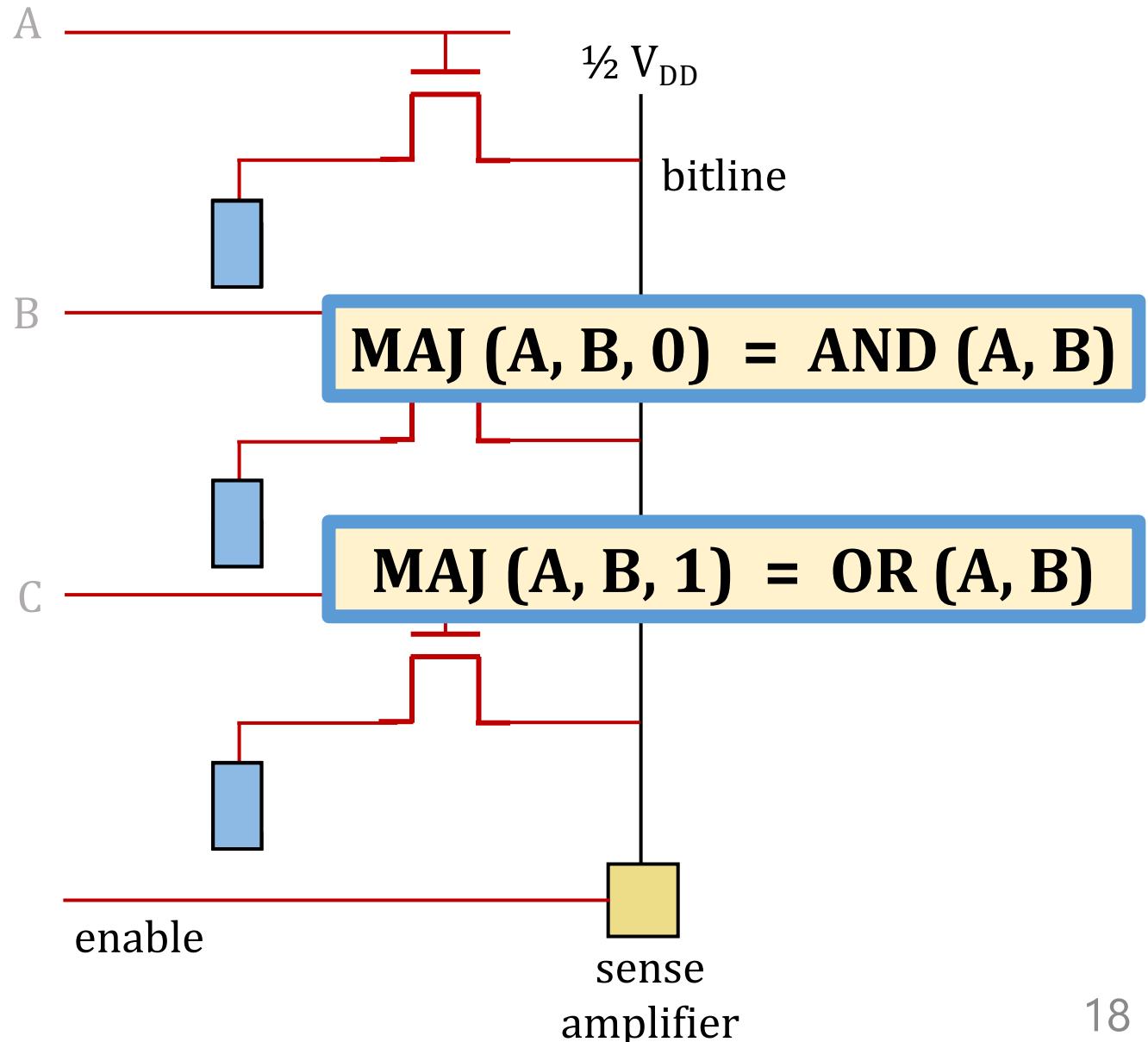
$$\text{MAJ}(A, B, C) =$$
$$\text{MAJ}(V_{dd}, V_{dd}, 0) = V_{dd}$$

3. values in cells A, B, C will be overwritten with the majority output

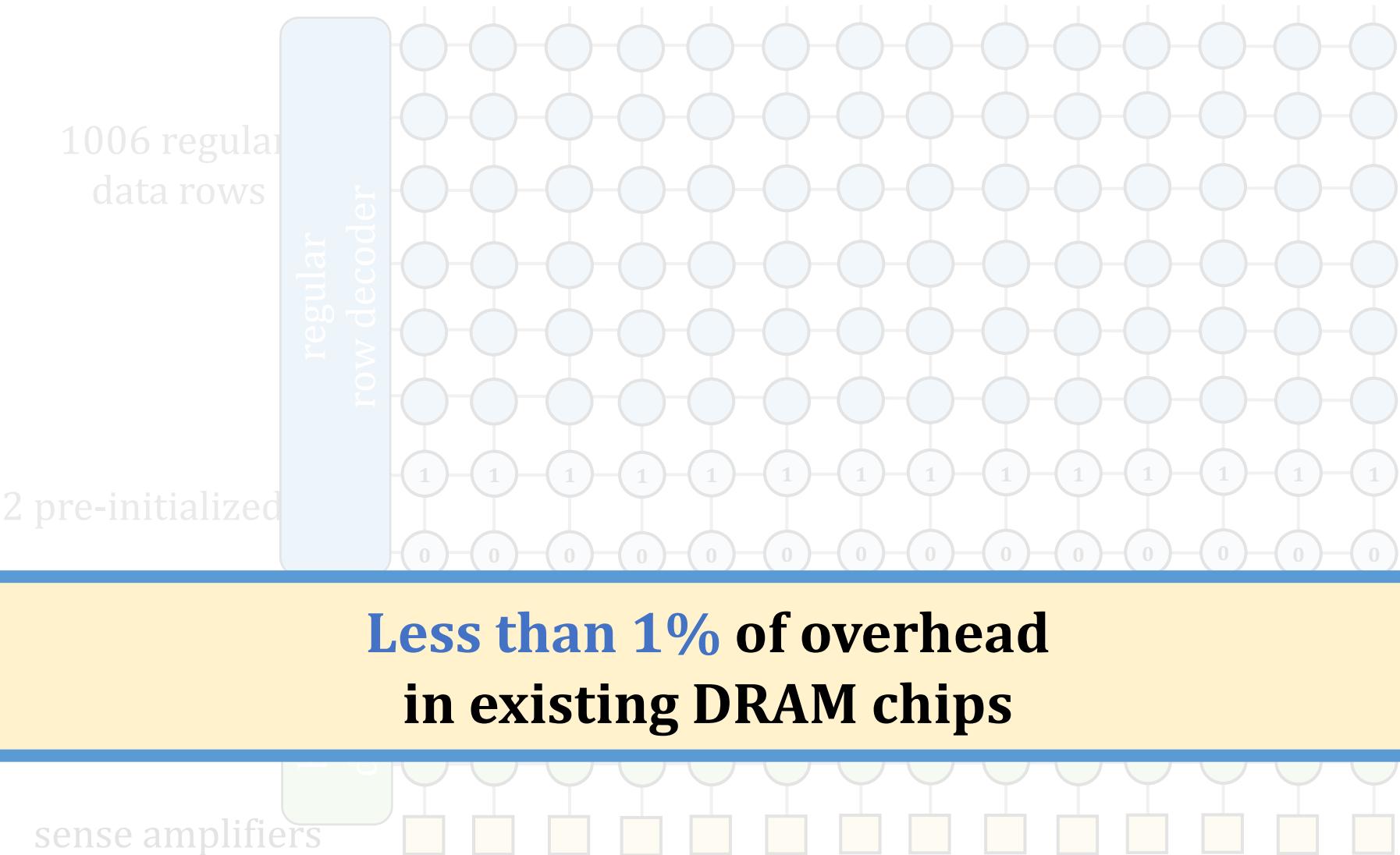
4. PRECHARGE bitline for next access



Ambit: In-DRAM Bulk Bitwise AND/OR



Ambit: Subarray Organization



PuM: Prior Works

- DRAM and other memory technologies that are capable of performing **computation using memory**

Shortcomings:

- Support **only basic** operations (e.g., Boolean operations, addition)
 - Not widely applicable
- Support a **limited** set of operations
 - Lack the flexibility to support new operations
- Require **significant changes** to the DRAM
 - Costly (e.g., area, power)

PuM: Prior Works

- DRAM and other memory technologies that are capable of performing **computation using memory**

Shortcomings:

- Support **only basic** operations (e.g., Boolean operations, addition)

Need a framework that aids **general adoption of PuM**, by:

- Efficiently implementing **complex operations**
- Providing **flexibility to support new operations**

- Costly (e.g., area, power)

Our Goal

Goal: Design a PuM framework that

- Efficiently implements complex operations
- Provides the flexibility to support new desired operations
- Minimally changes the DRAM architecture

Outline

1. Processing-using-DRAM

2. Background

3. SIMD RAM

Processing-using-DRAM Substrate
Framework

4. System Integration

5. Evaluation

6. Conclusion

Key Idea

- **SIMDRAM:** An end-to-end processing-using-DRAM framework that provides the programming interface, the ISA, and the hardware support for:
 - Efficiently computing complex operations in DRAM
 - Providing the ability to implement arbitrary operations as required
 - Using an in-DRAM massively-parallel SIMD substrate that requires minimal changes to DRAM architecture

Outline

1. Processing-using-DRAM

2. Background

3. SIMD RAM

Processing-using-DRAM Substrate

Framework

4. System Integration

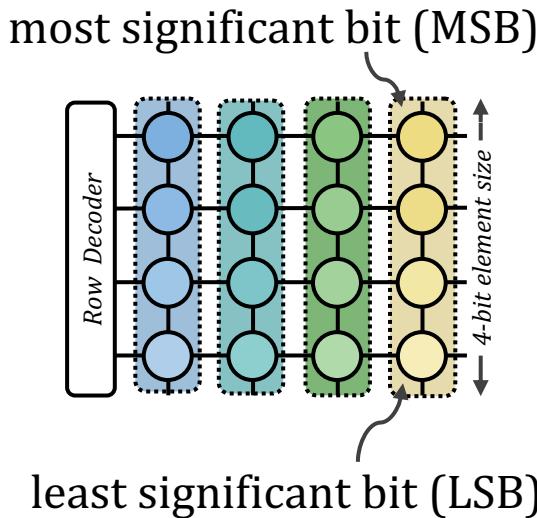
5. Evaluation

6. Conclusion

SIMDRAM: PuM Substrate

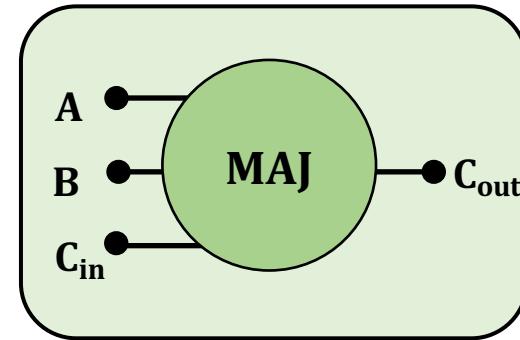
- SIMDRAM framework is built around a DRAM substrate that enables two techniques:

(1) Vertical data layout



(2) Majority-based computation

$$C_{out} = AB + AC_{in} + BC_{in}$$



Pros compared to the conventional **horizontal layout:**

- Implicit shift operation
- Massive parallelism

Pros compared to **AND/OR/NOT-based computation:**

- Higher performance
- Higher throughput
- Lower energy consumption

Outline

1. Processing-using-DRAM

2. Background

3. SIMD RAM

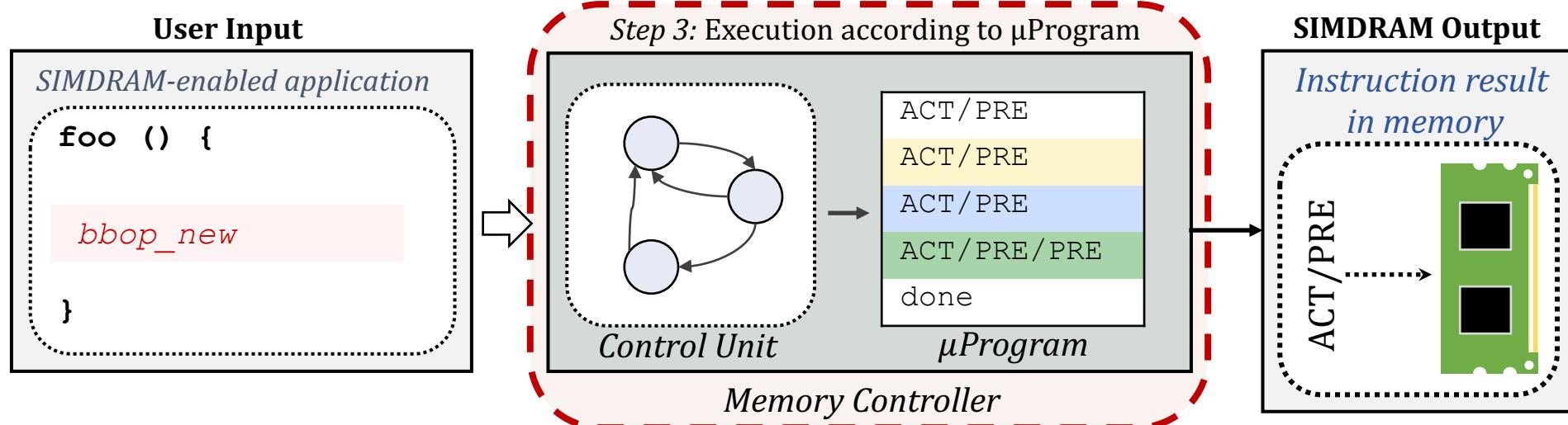
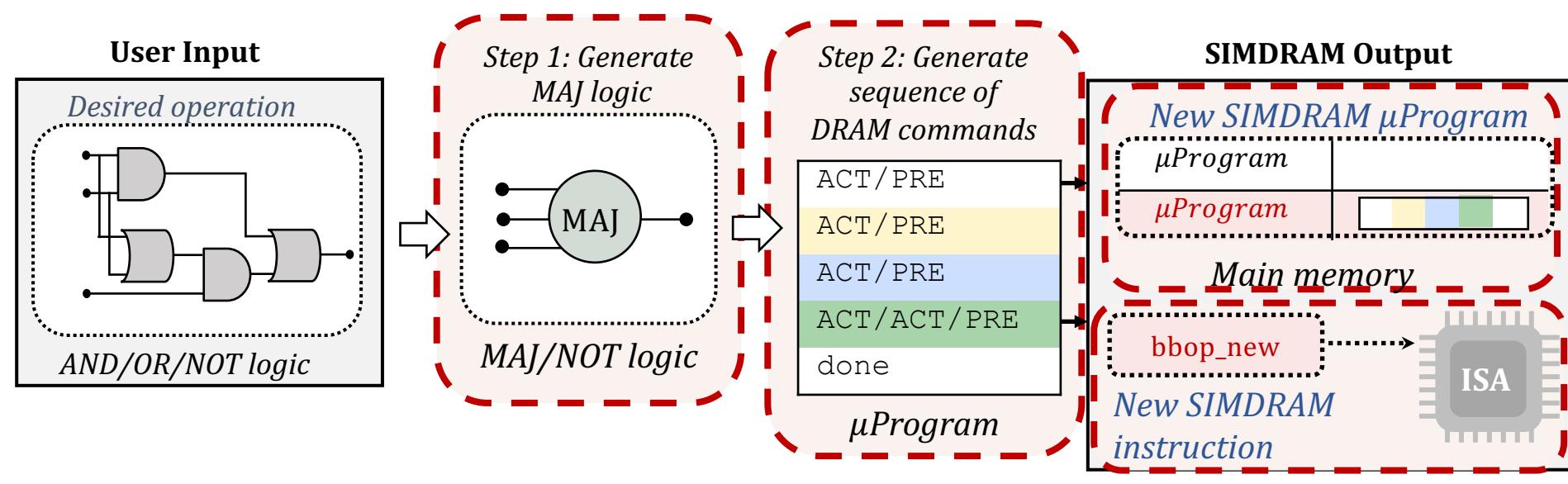
Processing-using-DRAM Substrate
Framework

4. System Integration

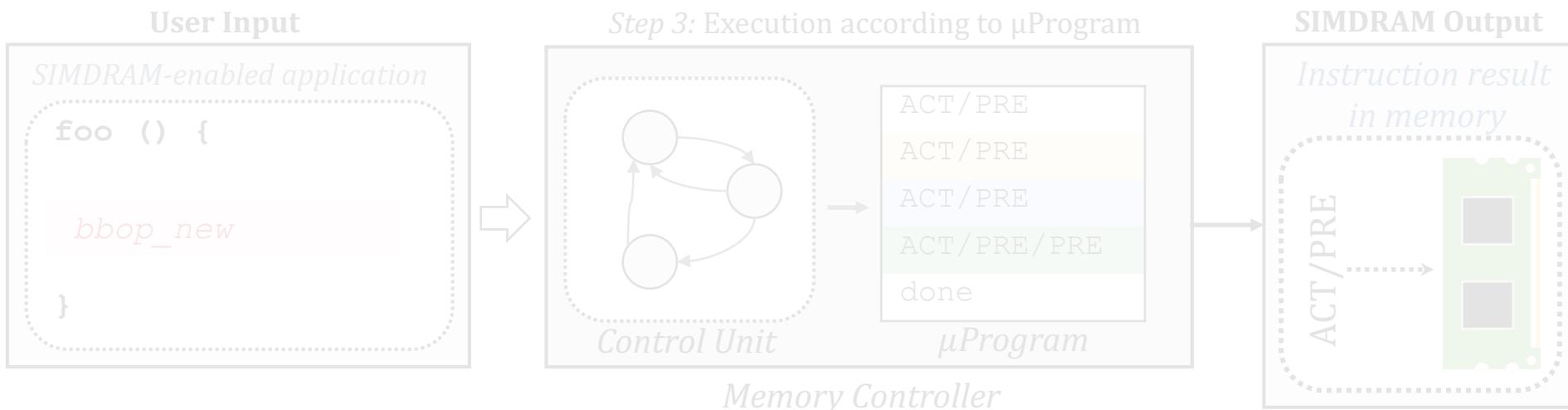
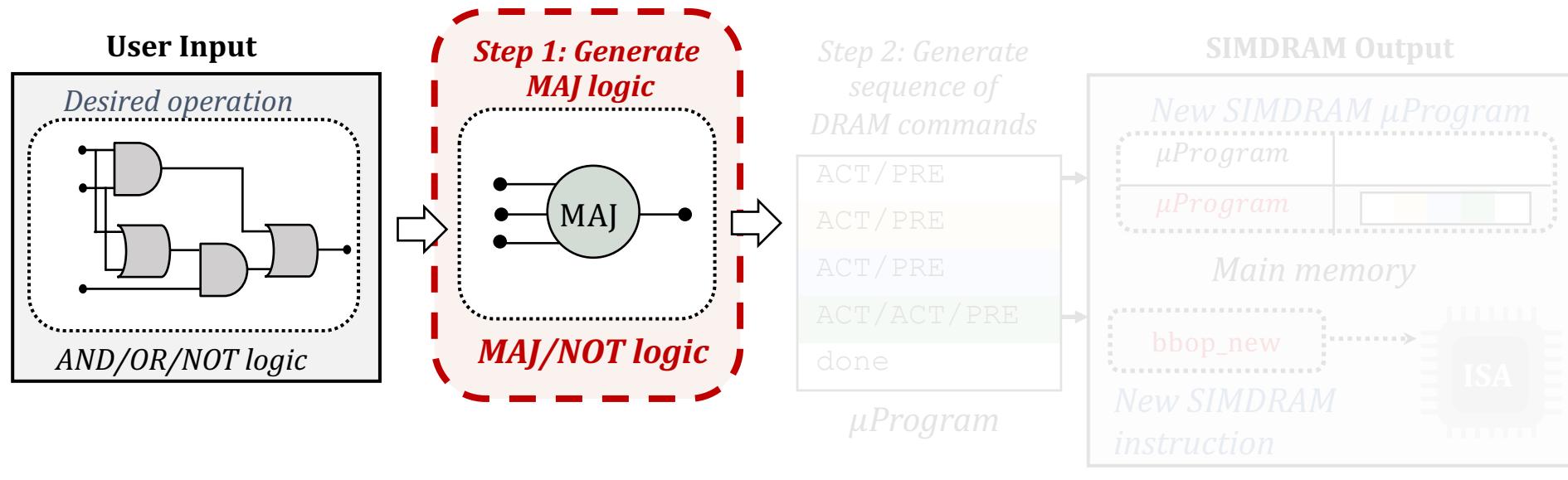
5. Evaluation

6. Conclusion

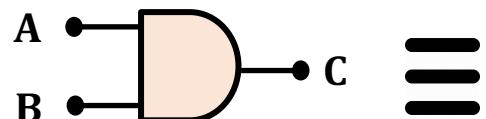
SIMDRAM Framework



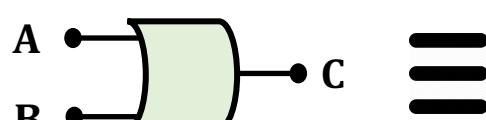
SIMDRAM Framework: Step 1



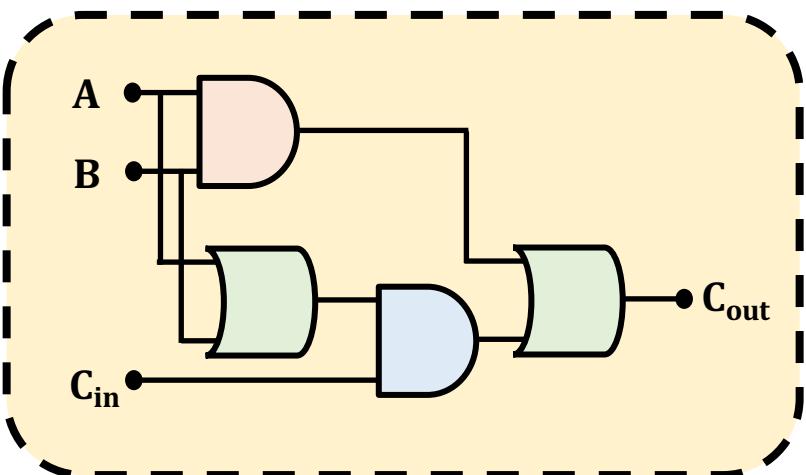
Step 1: Naïve MAJ/NOT Implementation



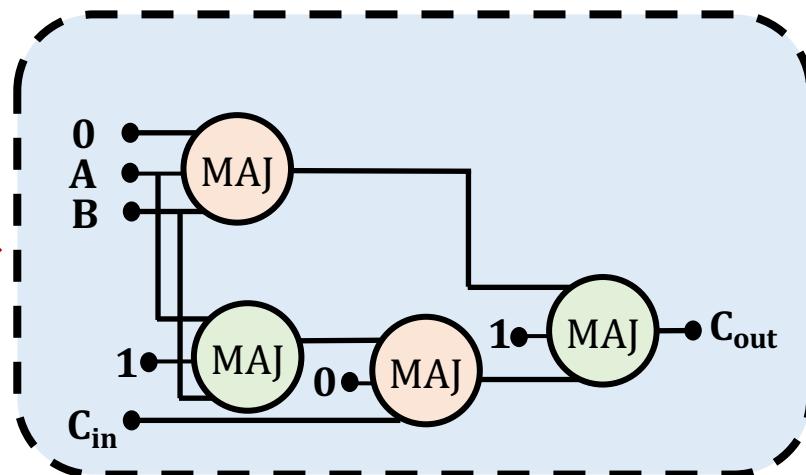
output is “1” only when $A = B = “1”$



output is “0” only when $A = B = “0”$

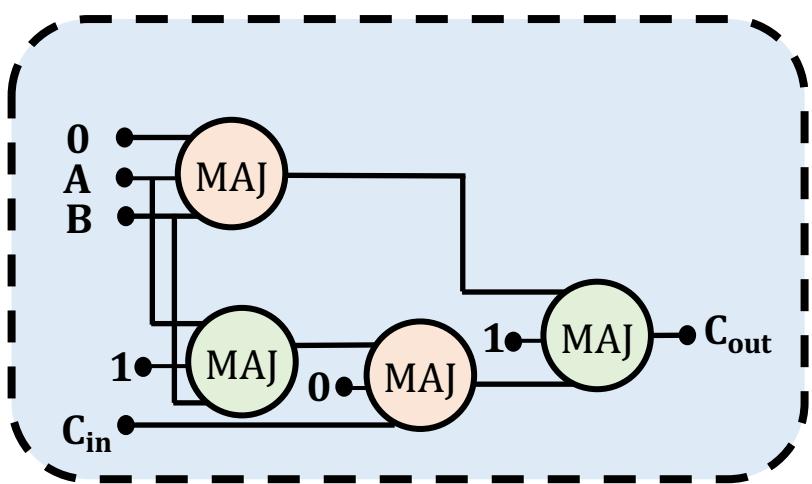


Part 1



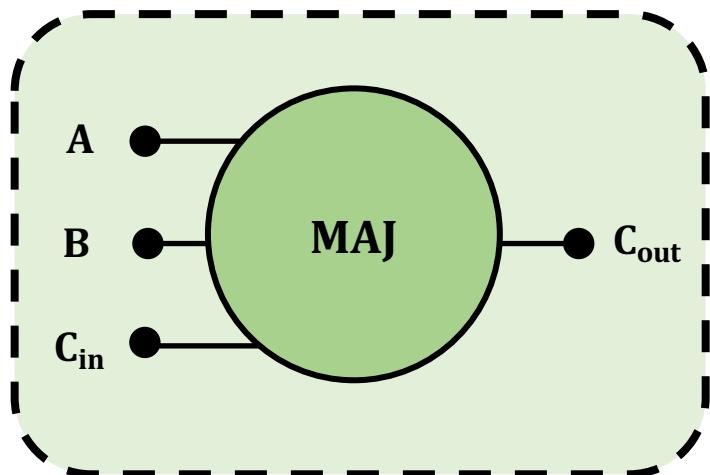
Naïvely converting AND/OR/NOT-implementation to MAJ/NOT-implementation leads to an unoptimized circuit

Step 1: Efficient MAJ/NOT Implementation



Greedy
optimization
algorithm⁴

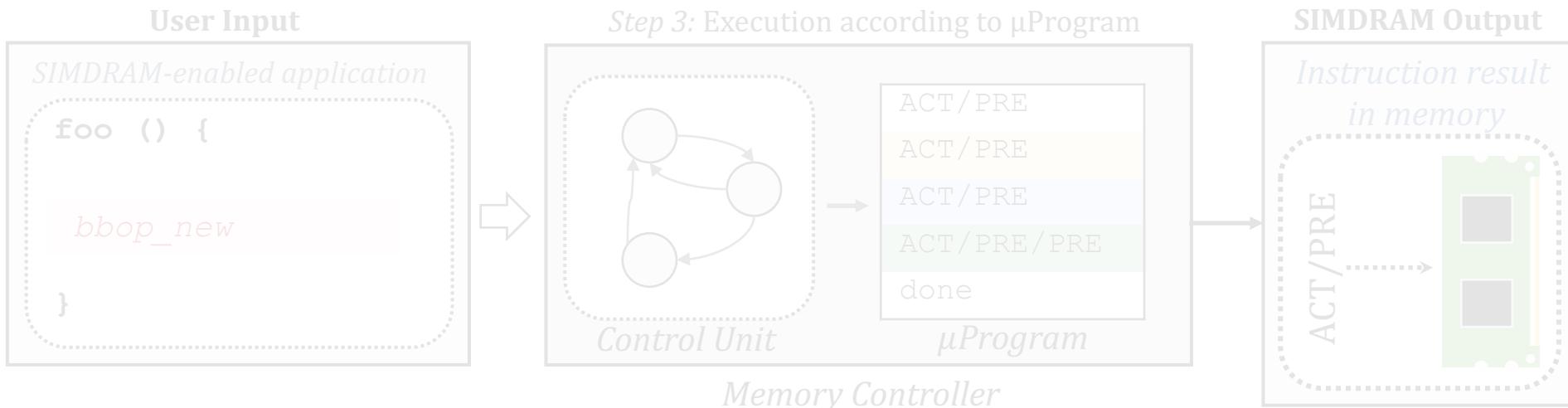
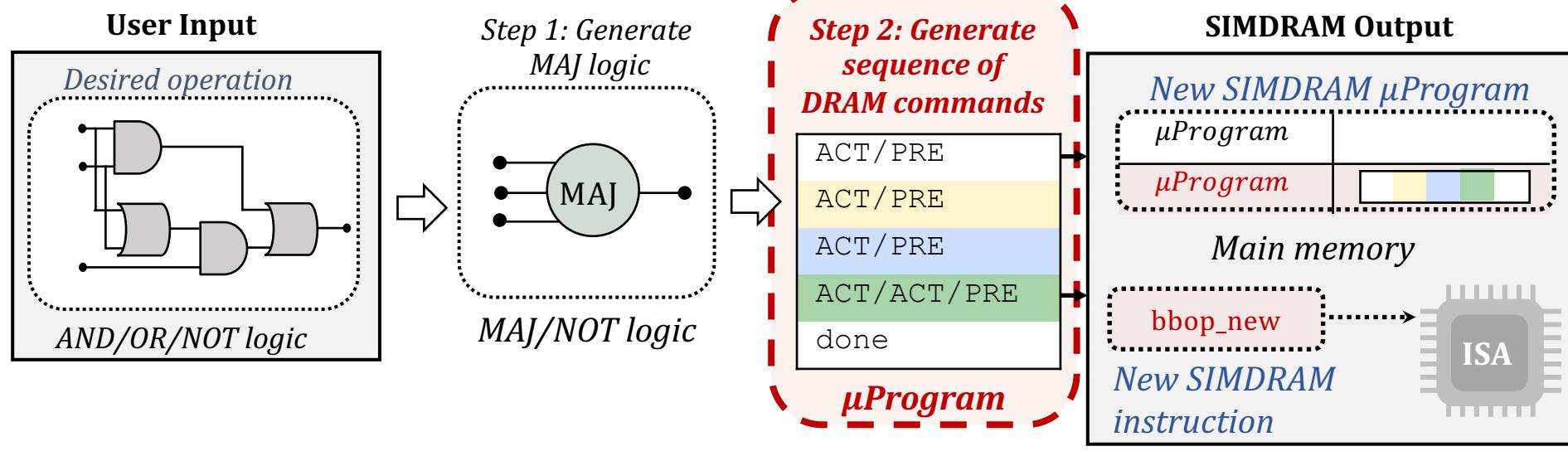
Part 2



Step 1 generates an optimized
MAJ/NOT-implementation of the desired operation

⁴ L. Amarù et al, "Majority-Inverter Graph: A Novel Data-Structure and Algorithms for Efficient Logic Optimization", DAC, 2014.

SIMDRAM Framework: Step 2



Step 2: μProgram Generation

- **μProgram:** A series of microarchitectural operations (e.g., ACT/PRE) that SIMDRAM uses to execute SIMDRAM operation in DRAM
- **Goal of Step 2:** To generate the μProgram that executes the desired SIMDRAM operation in DRAM

Task 1: Allocate DRAM rows to the operands

Task 2: Generate μProgram

Step 2: μProgram Generation

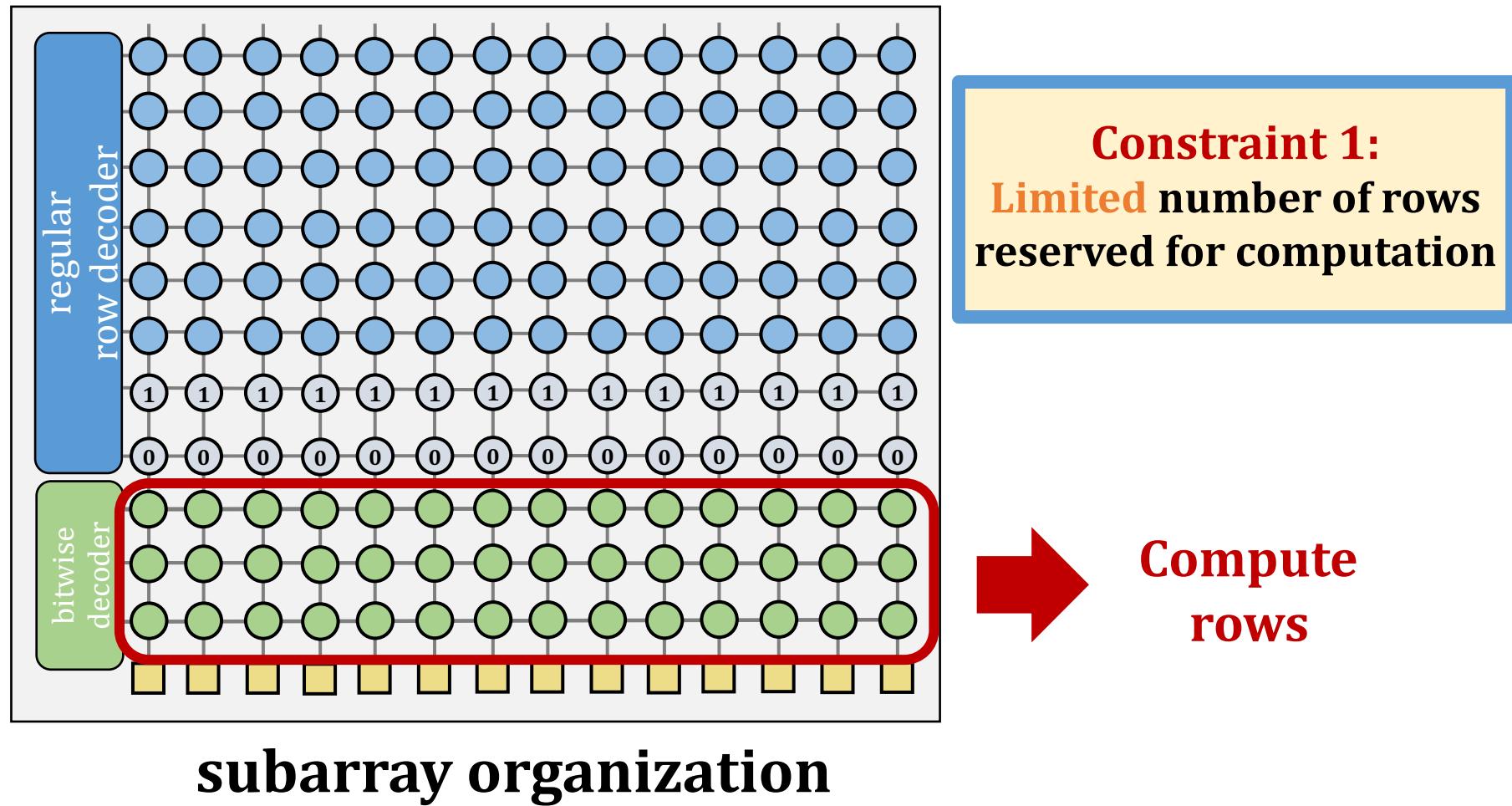
- **μProgram:** A series of microarchitectural operations (e.g., ACT/PRE) that SIMDRAM uses to execute SIMDRAM operation in DRAM
- **Goal of Step 2:** To generate the μProgram that executes the desired SIMDRAM operation in DRAM

Task 1: Allocate DRAM rows to the operands

Task 2: Generate μProgram

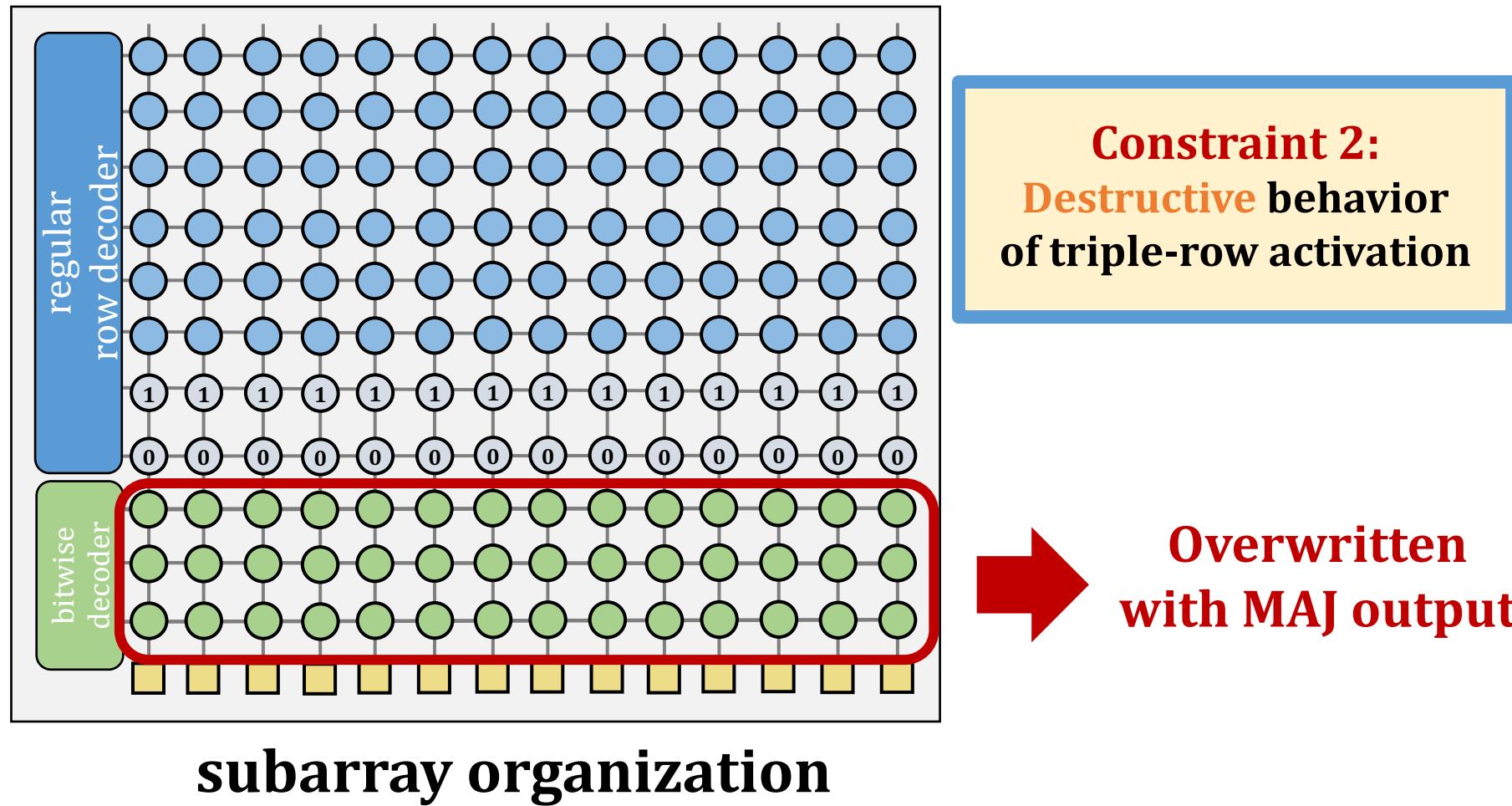
Task 1: Allocating DRAM Rows to Operands

- Allocation algorithm considers two constraints specific to processing-using-DRAM



Task 1: Allocating DRAM Rows to Operands

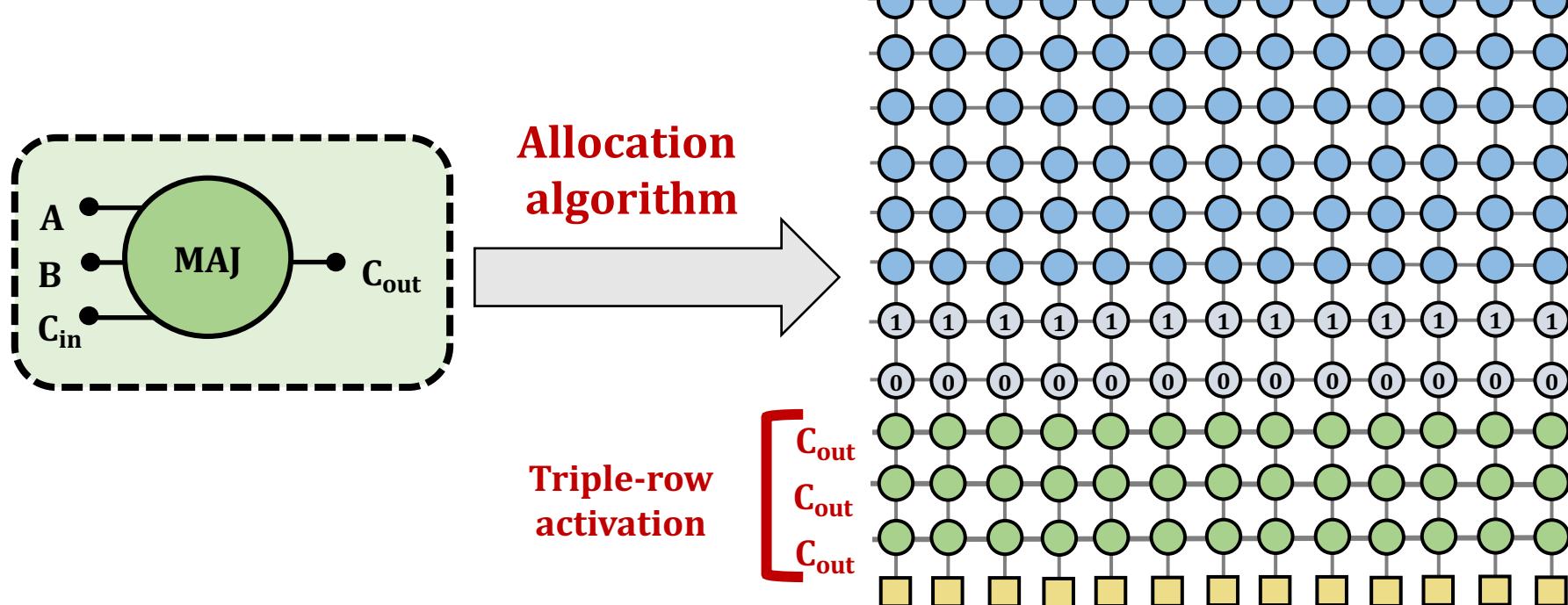
- Allocation algorithm considers two constraints specific to processing-using-DRAM



Task 1: Allocating DRAM Rows to Operands

- Allocation algorithm:

- Assigns as many inputs as the number of free compute rows
- All three input rows contain the MAJ output and can be reused



Step 2: μProgram Generation

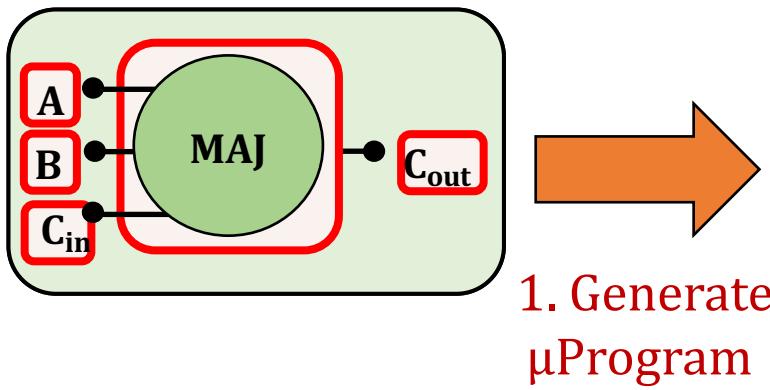
- **μProgram:** A series of microarchitectural operations (e.g., ACT/PRE) that SIMDRAM uses to execute SIMDRAM operation in DRAM
- **Goal of Step 2:** To generate the μProgram that executes the desired SIMDRAM operation in DRAM

Task 1: Allocate DRAM rows to the operands

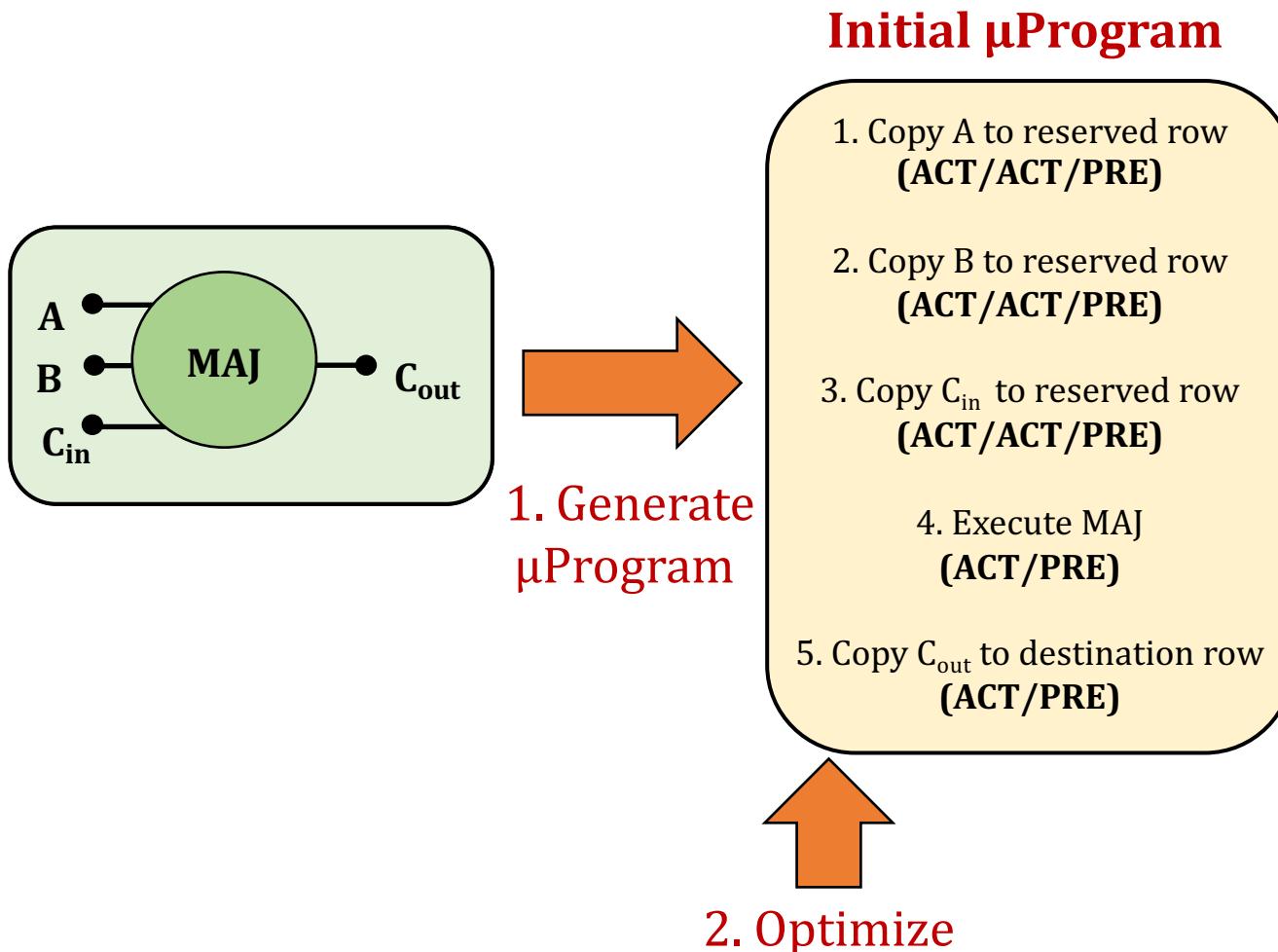
Task 2: Generate μProgram

Task 2: Generate an initial μ Program

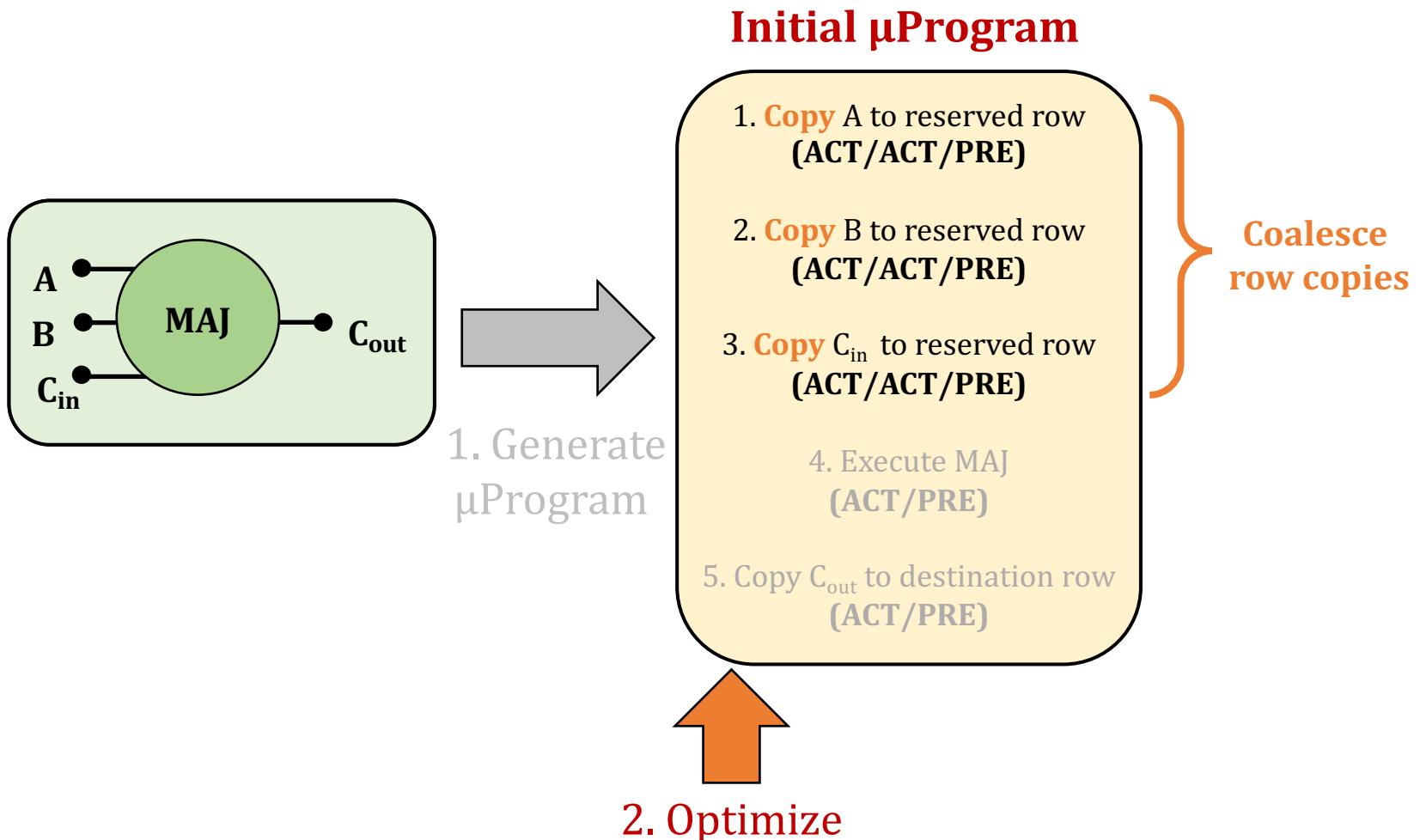
—



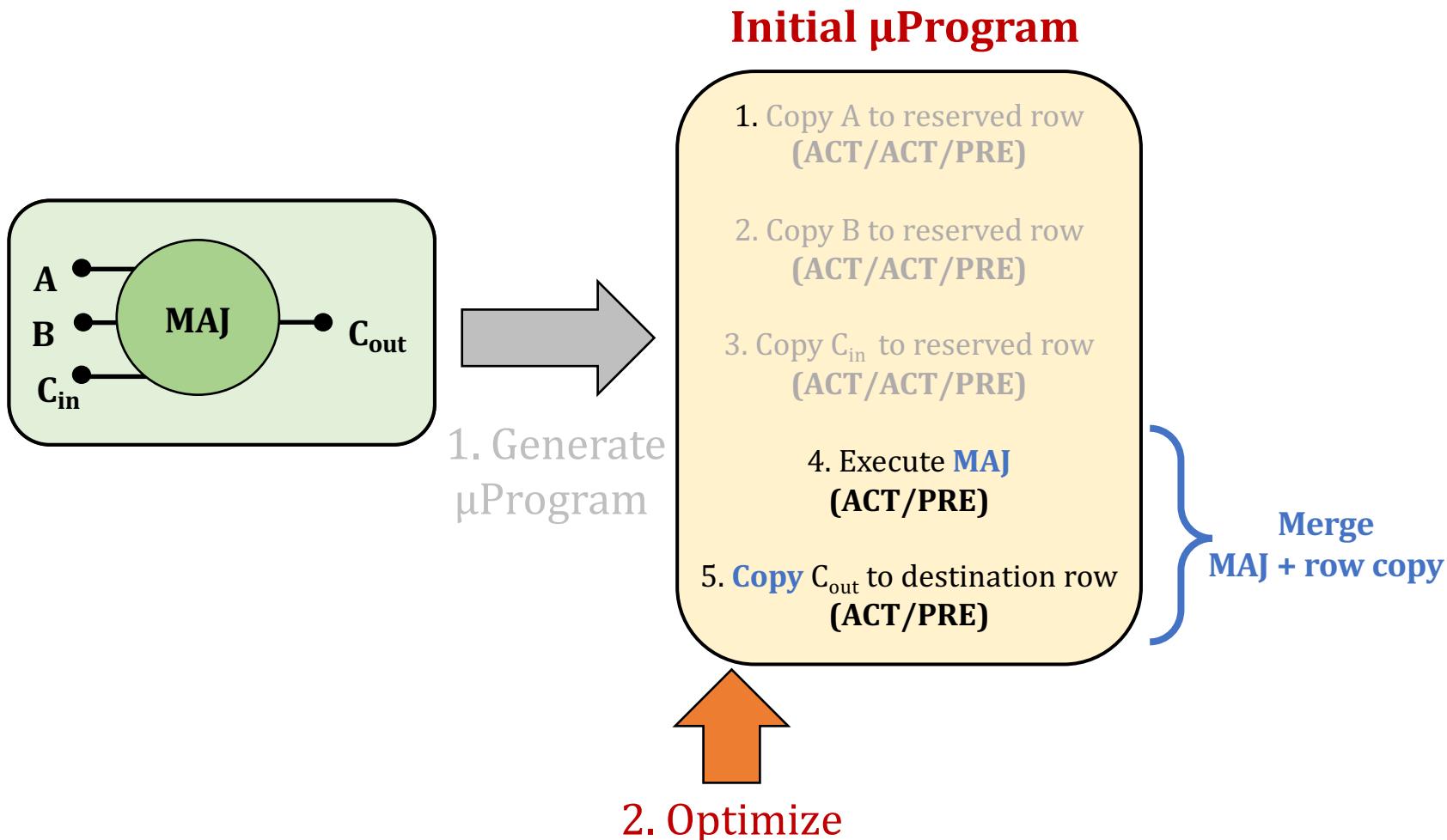
Task 2: Optimize the μ Program



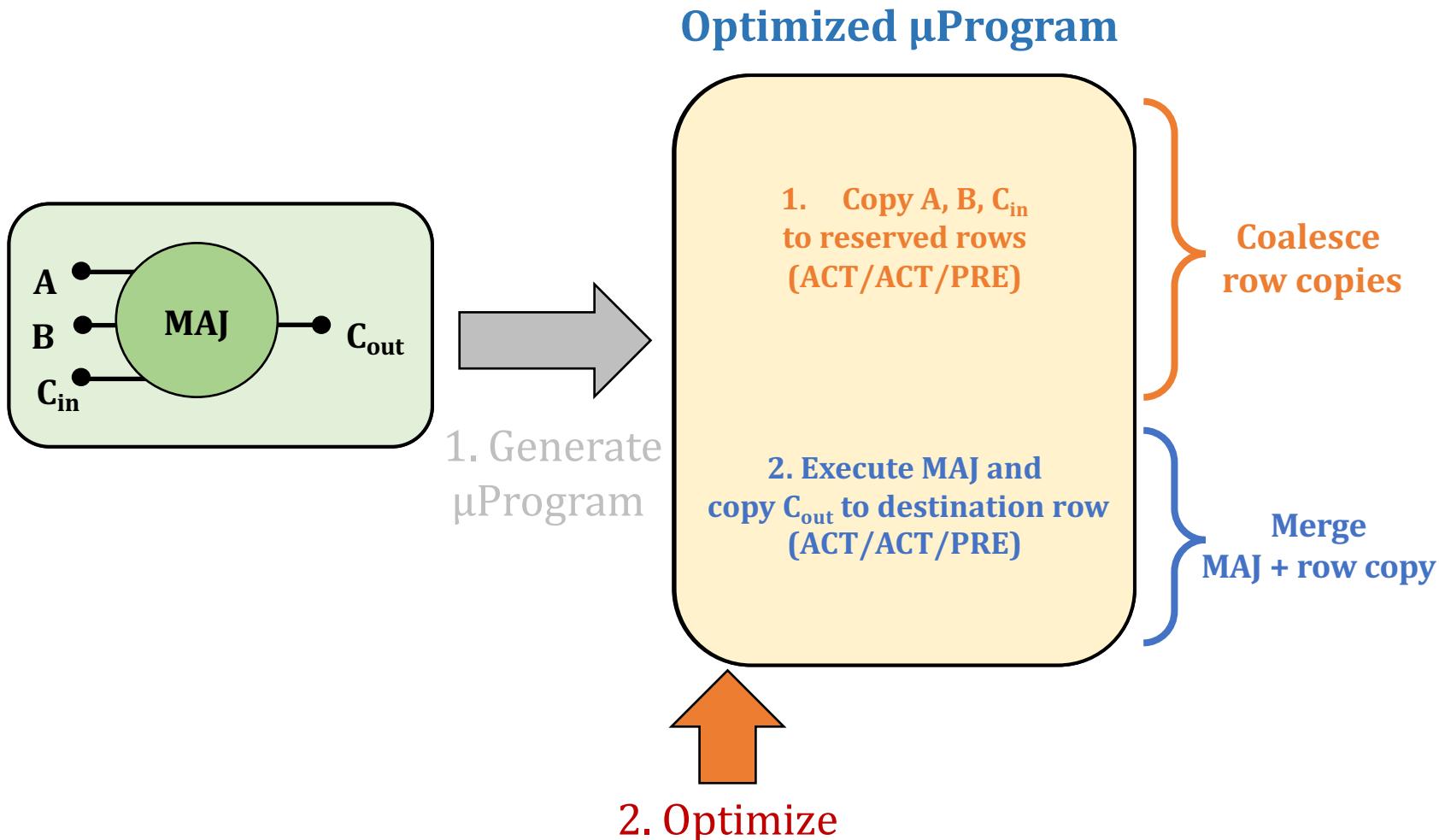
Task 2: Optimize the μ Program



Task 2: Optimize the μ Program

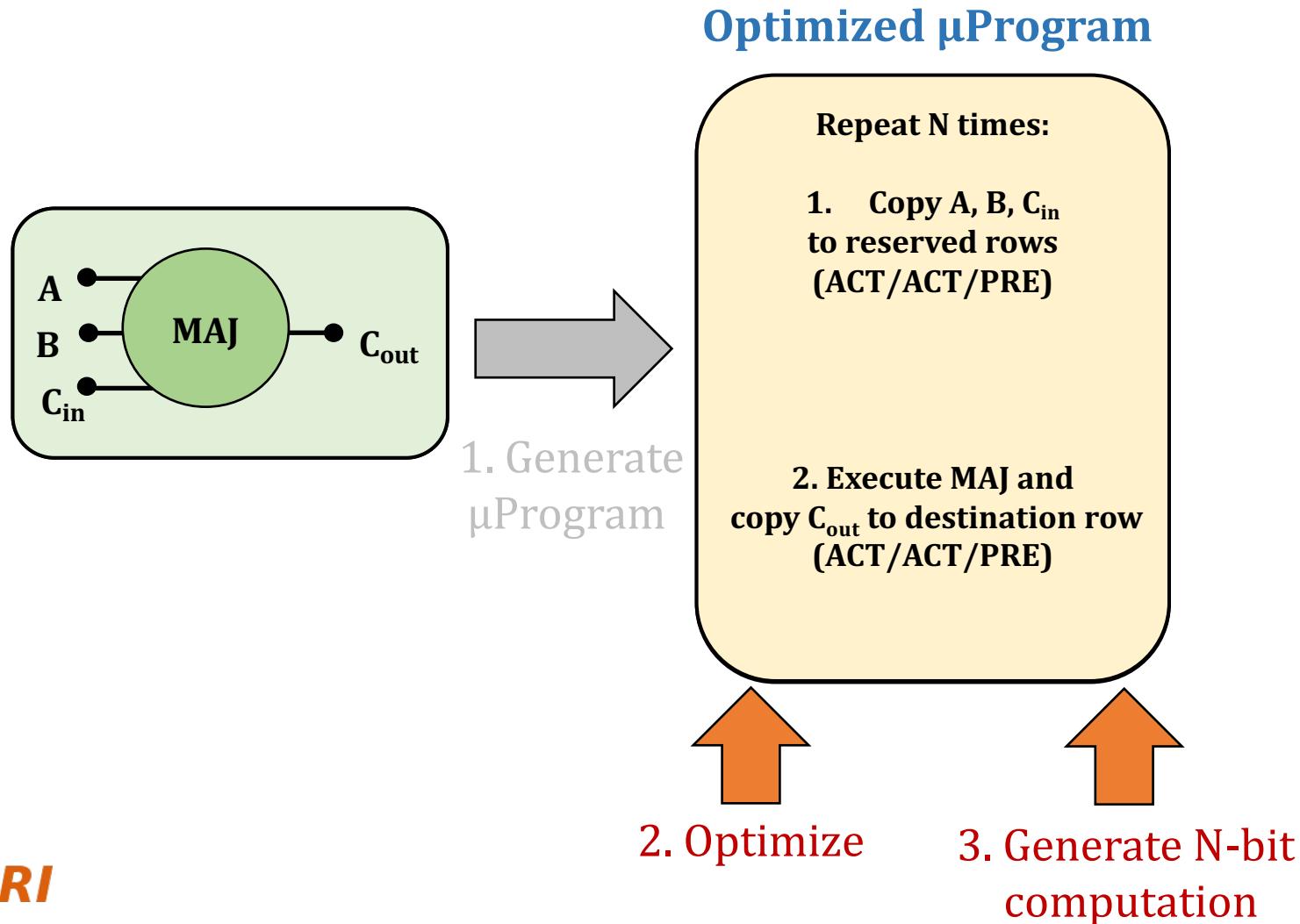


Task 2: Optimize the μ Program



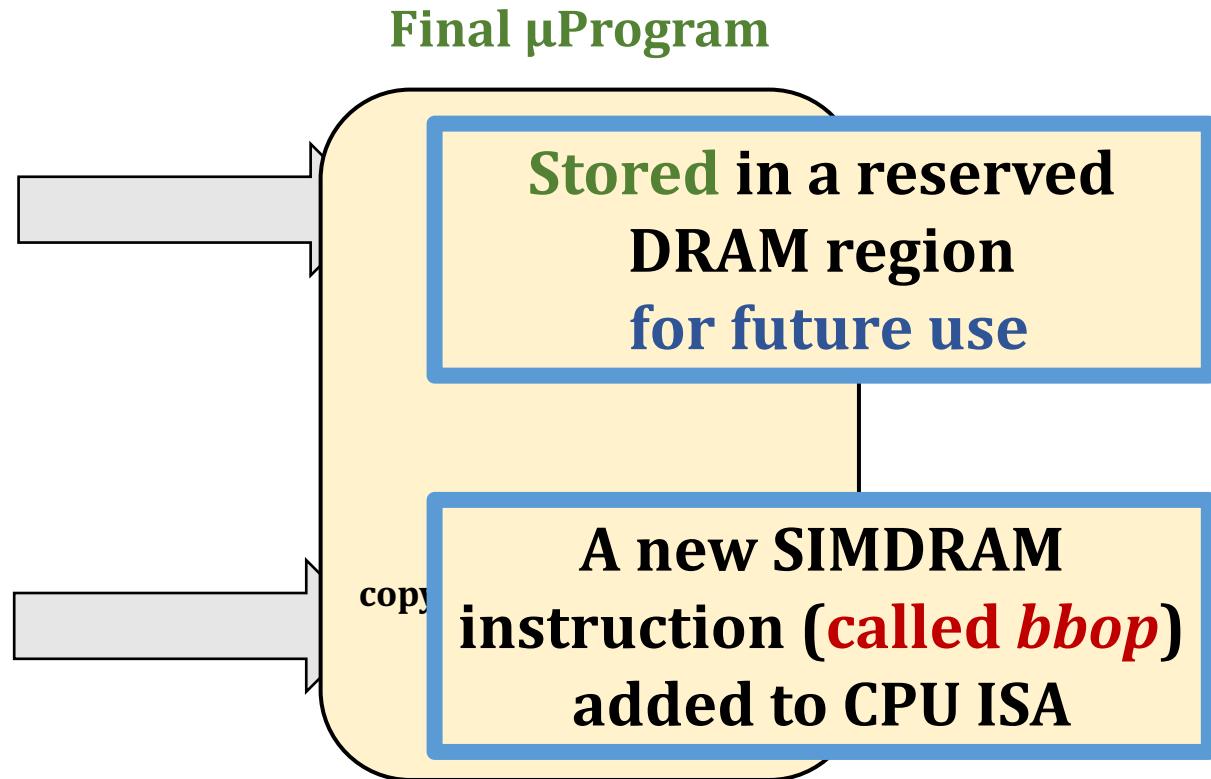
Task 2: Generate N-bit Computation

- Final **μProgram** is optimized and computes the desired operation for operands of N-bit size in a bit-serial fashion

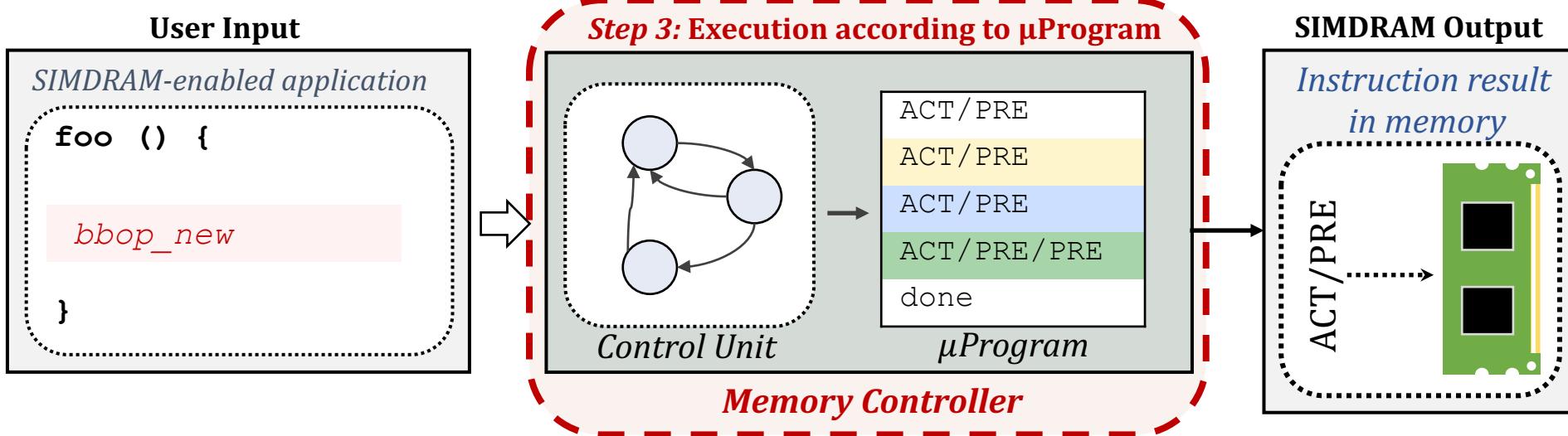
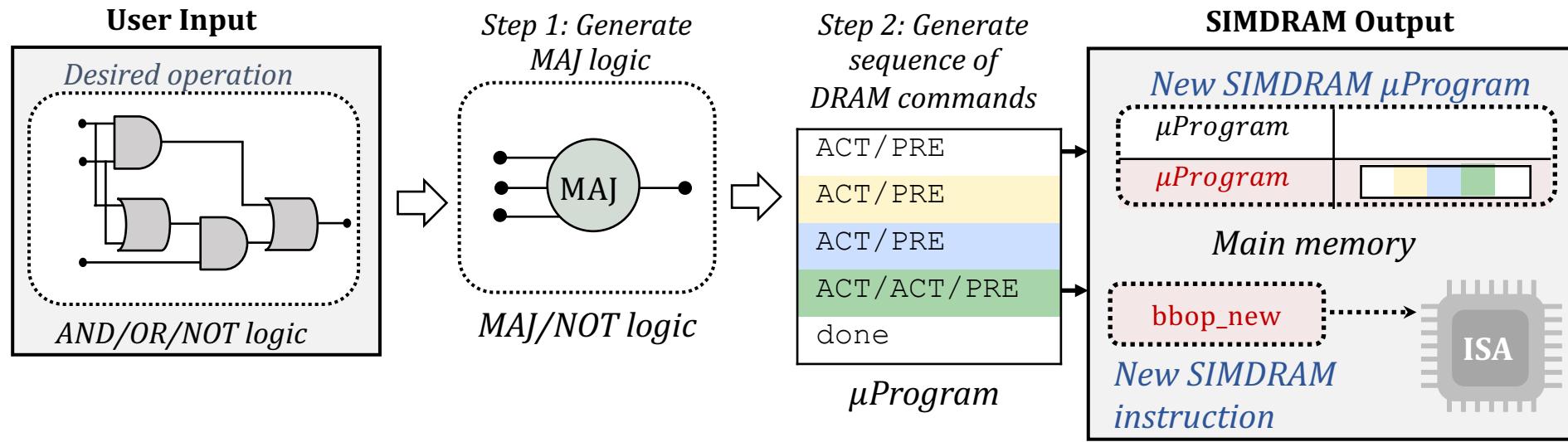


Task 2: Generate μProgram

- **Final μProgram** is optimized and computes the desired operation for operands of N-bit size in a bit-serial fashion

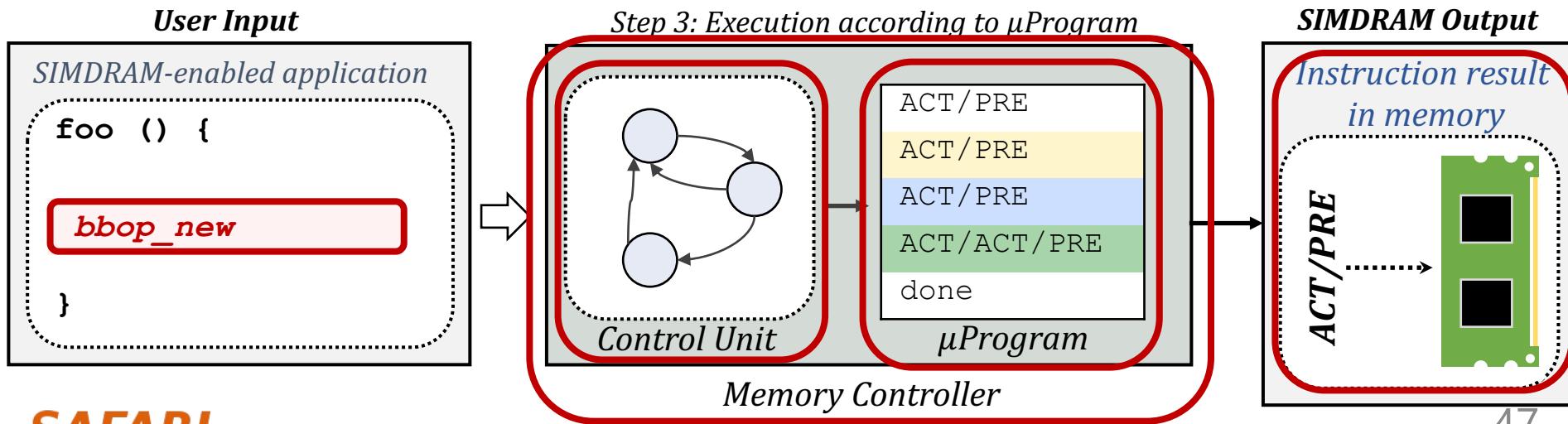


SIMDRAM Framework: Step 3



Step 3: μProgram Execution

- **SIMDRAM control unit:** handles the execution of the μProgram at runtime
- Upon receiving a **bbop instruction**, the control unit:
 1. Loads the μProgram corresponding to SIMDRAM operation
 2. Issues the sequence of DRAM commands (ACT/PRE) stored in the μProgram to SIMDRAM subarrays to perform the in-DRAM operation



Outline

1. Processing-using-DRAM

2. Background

3. SIMD RAM

Processing-using-DRAM Substrate
Framework

4. System Integration

5. Evaluation

6. Conclusion

System Integration

Efficiently transposing data

Programming interface

Handling page faults, address translation,
coherence, and interrupts

Handling limited subarray size

Security implications

Limitations of our framework

System Integration

Efficiently transposing data

Programming interface

Handling page faults, address translation,
coherence, and interrupts

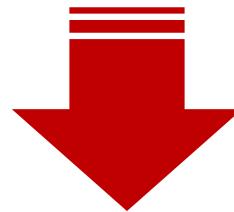
Handling limited subarray size

Security implications

Limitations of our framework

Transposing Data

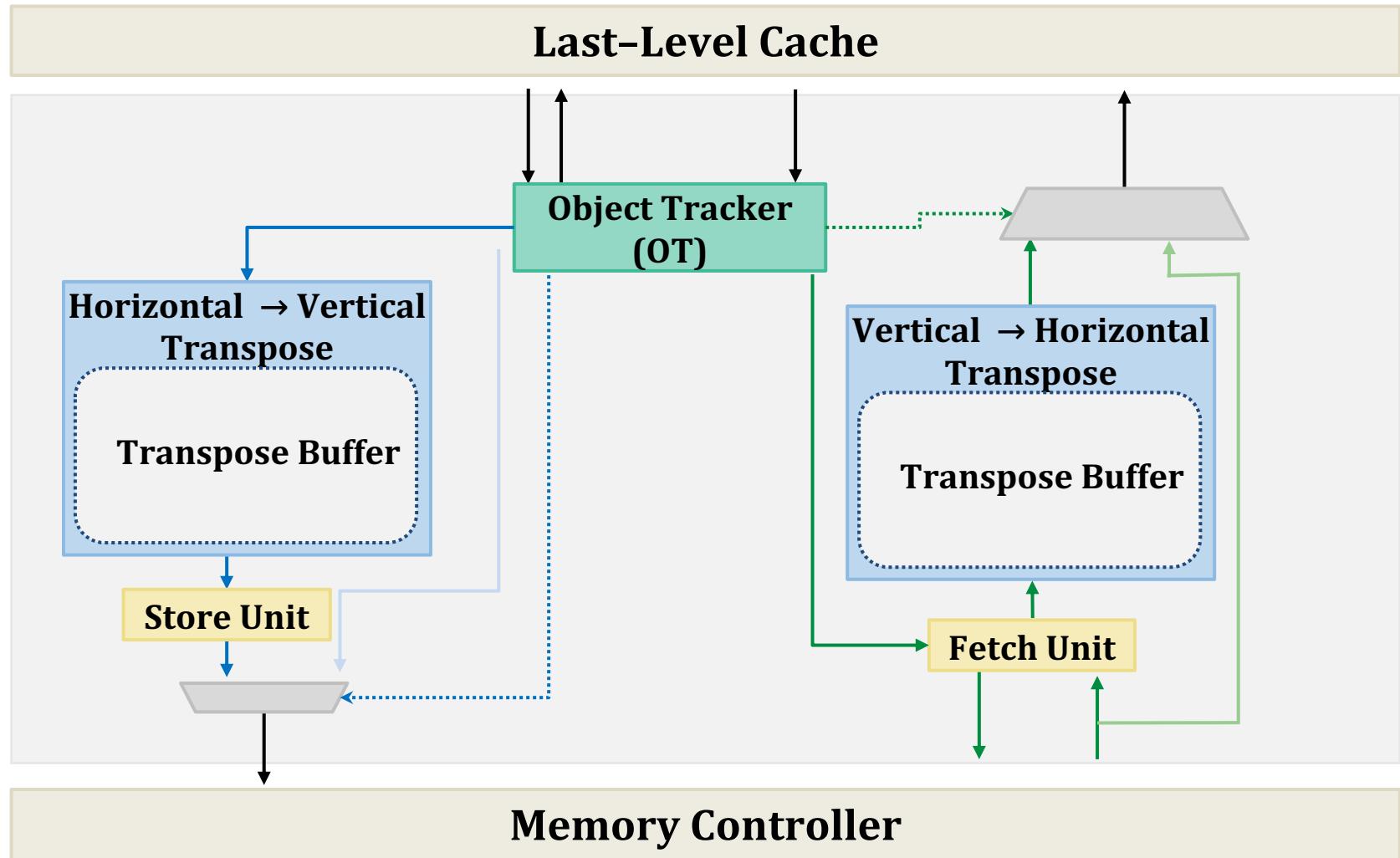
- SIMD RAM operates on vertically-laid-out data
- Other system components expect data to be laid out horizontally



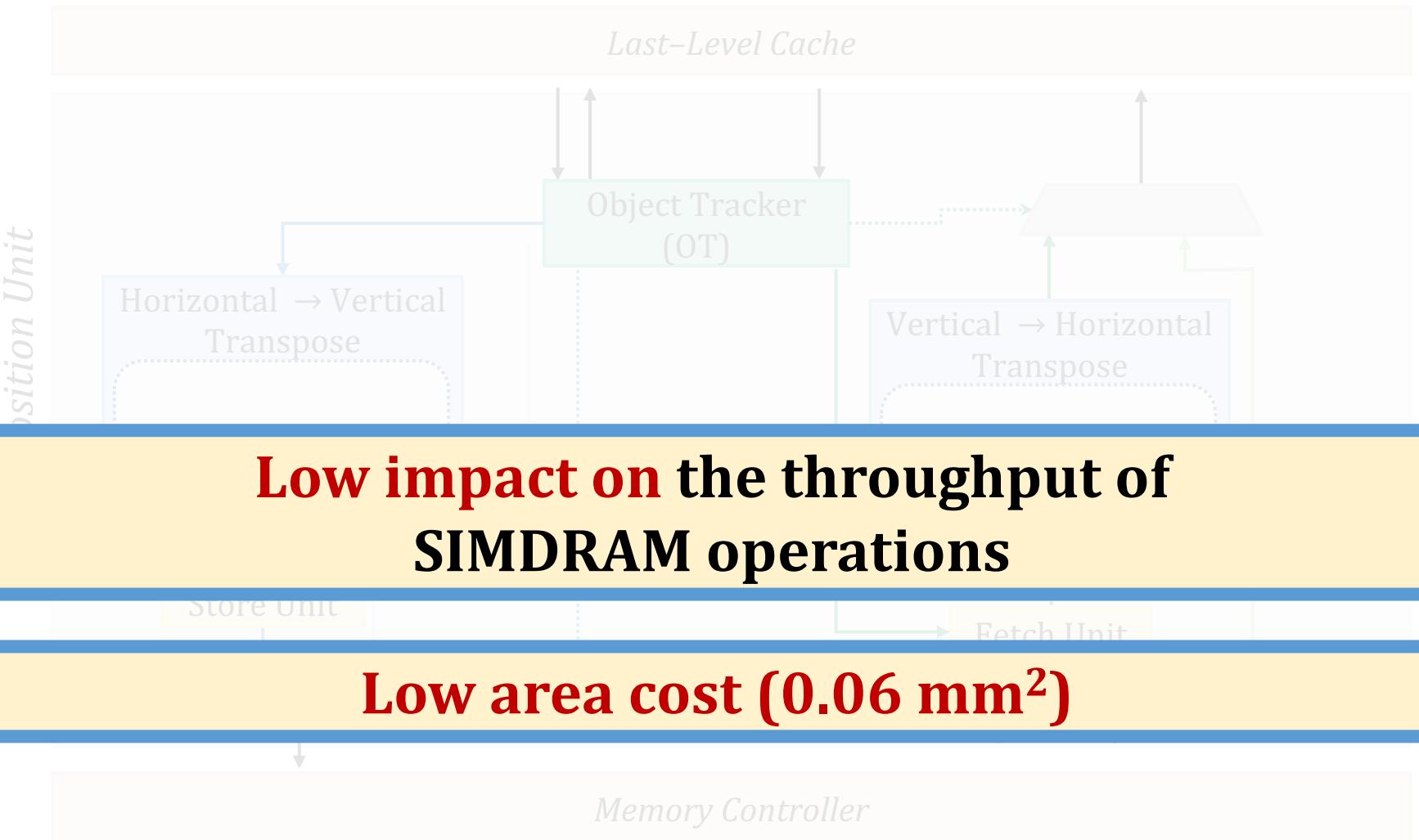
Challenging to share data between SIMD RAM and CPU

Transposition Unit

Transposition Unit



Efficiently Transposing Data



System Integration

Efficiently transposing data

Programming interface

Handling page faults, address translation,
coherence, and interrupts

Handling limited subarray size

Security implications

Limitations of our framework

Programming Interface

- Four new SIMD RAM ISA extensions

Type	ISA Format
------	------------

Programming Interface

- Four new SIMD RAM ISA extensions

Type	ISA Format
Initialization	bbop_trsp_init address, size, n

Programming Interface

- Four new SIMD RAM ISA extensions

Type	ISA Format
Initialization	bbop_trsp_init address, size, n
1-Input Operation	bbop_op dst, src, size, n

Programming Interface

- Four new SIMD RAM ISA extensions

Type	ISA Format
Initialization	bbop_trsp_init address, size, n
1-Input Operation	bbop_op dst, src, size, n
2-Input Operation	bbop_op dst, src_1, src_2, size, n

Programming Interface

- Four new SIMD RAM ISA extensions

Type	ISA Format
Initialization	bbop_trsp_init address, size, n
1-Input Operation	bbop_op dst, src, size, n
2-Input Operation	bbop_op dst, src_1, src_2, size, n
Predication	bbop_if_else dst, src_1, src_2, select, size, n

Code Using SIMD RAM Instructions

```
1 int size = 65536;
2 int elm_size = sizeof (uint8_t);
3 uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5 ...
6 for (int i = 0; i < size ; ++ i){
7     bool cond = A[i] > pred[i];
8     if (cond)
9         C [i] = A[i] + B[i];
10    else
11        C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub
with predicated execution

```
1 int size = 65536;
2 int elm_size = sizeof(uint8_t);
3 uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5 bbop_trsp_init(A , size , elm_size);
6 bbop_trsp_init(B , size , elm_size);
7 bbop_trsp_init(C , size , elm_size);
8 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9 // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 ...
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

Equivalent code using
SIMDRAM operations →

Code Using SIMD RAM Instructions

```
1 int size = 65536;
2 int elm_size = sizeof (uint8_t);
3 uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5 ...
6 for (int i = 0; i < size ; ++ i){
7     bool cond = A[i] > pred[i];
8     if (cond)
9         C [i] = A[i] + B[i];
10    else
11        C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub
with predicated execution

```
1 int size = 65536;
2 int elm_size = sizeof(uint8_t);
3 uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5 bbop_trsp_init(A , size , elm_size);
6 bbop_trsp_init(B , size , elm_size);
7 bbop_trsp_init(C , size , elm_size);
8 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9 // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 ...
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

Equivalent code using
SIMDRAM operations →

Code Using SIMD RAM Instructions

```
1 int size = 65536;
2 int elm_size = sizeof (uint8_t);
3 uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5 ...
6 for (int i = 0; i < size ; ++ i){
7     bool cond = A[i] > pred[i];
8     if (cond)
9         C [i] = A[i] + B[i];
10    else
11        C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub
with predicated execution

```
1 int size = 65536;
2 int elm_size = sizeof(uint8_t);
3 uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5 bbop_trsp_init(A , size , elm_size);
6 bbop_trsp_init(B , size , elm_size);
7 bbop_trsp_init(C , size , elm_size);
8 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9 // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 ...
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

Equivalent code using
SIMDRAM operations →

Code Using SIMD RAM Instructions

```
1 int size = 65536;
2 int elm_size = sizeof (uint8_t);
3 uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5 ...
6 for (int i = 0; i < size ; ++ i){
7     bool cond = A[i] > pred[i];
8     if (cond)
9         C [i] = A[i] + B[i];
10    else
11        C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub
with predicated execution

```
1 int size = 65536;
2 int elm_size = sizeof(uint8_t);
3 uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5 bbop_trsp_init(A , size , elm_size);
6 bbop_trsp_init(B , size , elm_size);
7 bbop_trsp_init(C , size , elm_size);
8 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9 // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 ...
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

Equivalent code using
SIMDRAM operations →

Code Using SIMD RAM Instructions

```
1 int size = 65536;
2 int elm_size = sizeof (uint8_t);
3 uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5 ...
6 for (int i = 0; i < size ; ++ i){
7     bool cond = A[i] > pred[i];
8     if (cond)
9         C [i] = A[i] + B[i];
10    else
11        C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub
with predicated execution

```
1 int size = 65536;
2 int elm_size = sizeof(uint8_t);
3 uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5 bbop_trsp_init(A , size , elm_size);
6 bbop_trsp_init(B , size , elm_size);
7 bbop_trsp_init(C , size , elm_size);
8 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9 // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 ...
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

Equivalent code using
SIMDRAM operations →

Code Using SIMD RAM Instructions

```
1 int size = 65536;
2 int elm_size = sizeof (uint8_t);
3 uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5 ...
6 for (int i = 0; i < size ; ++ i){
7     bool cond = A[i] > pred[i];
8     if (cond)
9         C [i] = A[i] + B[i];
10    else
11        C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub
with predicated execution

```
1 int size = 65536;
2 int elm_size = sizeof(uint8_t);
3 uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5 bbop_trsp_init(A , size , elm_size);
6 bbop_trsp_init(B , size , elm_size);
7 bbop_trsp_init(C , size , elm_size);
8 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9 // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 ...
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

Equivalent code using
SIMDRAM operations →

Code Using SIMD RAM Instructions

```
1 int size = 65536;
2 int elm_size = sizeof (uint8_t);
3 uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5 ...
6 for (int i = 0; i < size ; ++ i){
7     bool cond = A[i] > pred[i];
8     if (cond)
9         C [i] = A[i] + B[i];
10    else
11        C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub
with predicated execution

```
1 int size = 65536;
2 int elm_size = sizeof(uint8_t);
3 uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5 bbop_trsp_init(A , size , elm_size);
6 bbop_trsp_init(B , size , elm_size);
7 bbop_trsp_init(C , size , elm_size);
8 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9 // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 ...
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

Equivalent code using
SIMDRAM operations →

More in the Paper

SIMDRAM: An End-to-End Framework for Bit-Serial SIMD Computing in DRAM

*Nastaran Hajinazar^{1,2}

Nika Mansouri Ghiasi¹

*Geraldo F. Oliveira¹

Minesh Patel¹

Sven Gregorio¹

Mohammed Alser¹

João Dinis Ferreira¹

Saugata Ghose³

Juan Gómez-Luna¹

Onur Mutlu¹

¹ETH Zürich

²Simon Fraser University

³University of Illinois at Urbana–Champaign

coherence, and interrupts

Handling limited subarray size

Security implications

Limitations of our framework

Outline

1. Processing-using-DRAM

2. Background

3. SIMD RAM

Processing-using-DRAM Substrate
Framework

4. System Integration

5. Evaluation

6. Conclusion

Methodology: Experimental Setup

- **Simulator:** gem5
- **Baselines:**
 - A multi-core CPU (Intel Skylake)
 - A high-end GPU (NVidia Titan V)
 - Ambit: a state-of-the-art in-memory computing mechanism
- **Evaluated SIMD RAM configurations** (all using a DDR4 device):
 - **1-bank:** SIMD RAM exploits 65'536 SIMD lanes (an 8 kB row buffer)
 - **4-banks:** SIMD RAM exploits 262'144 SIMD lanes
 - **16-banks:** SIMD RAM exploits 1'048'576 SIMD lanes

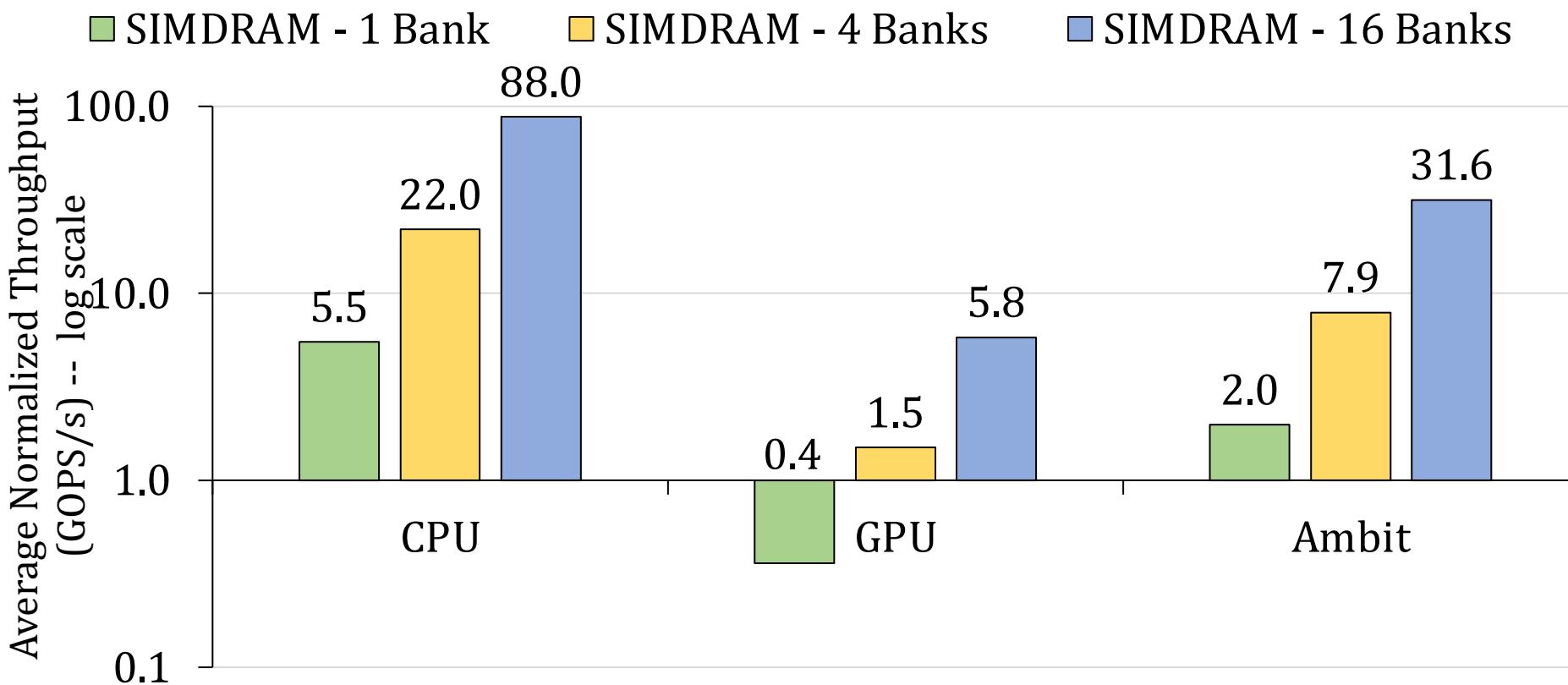
Methodology: Workloads

Evaluated:

- 16 complex in-DRAM operations:
 - Absolute
 - Addition/Subtraction
 - BitCount
 - Equality/ Greater/Greater Equal
 - Predication
 - ReLU
 - AND-/OR-/XOR-Reduction
 - Division/Multiplication
- 7 real-world applications
 - BitWeaving (databases)
 - TPH-H (databases)
 - kNN (machine learning)
 - LeNET (Neural Networks)
 - VGG-13/VGG-16 (Neural Networks)
 - brightness (graphics)

Throughput Analysis

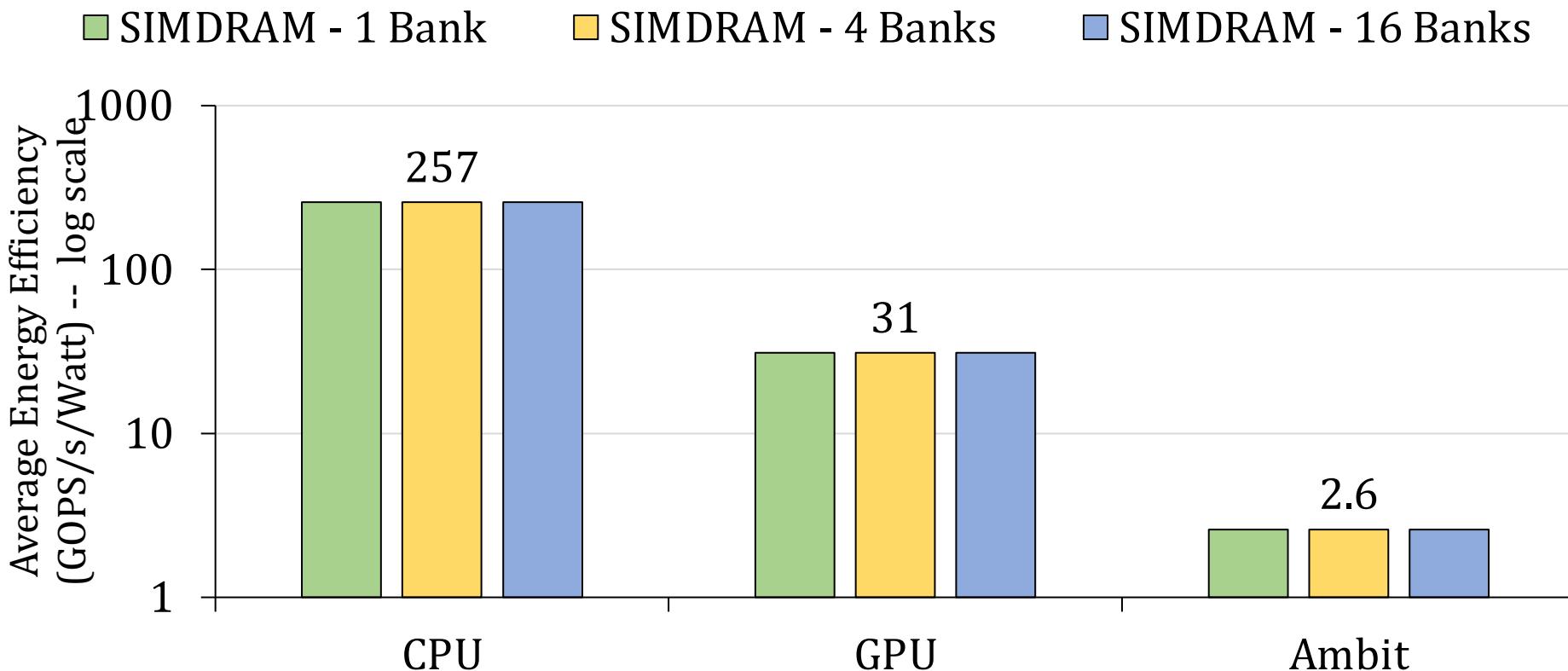
Average normalized throughput across all 16 SIMDRAM operations



**SIMDRAM significantly outperforms
all state-of-the-art baselines for a wide range of operations**

Energy Analysis

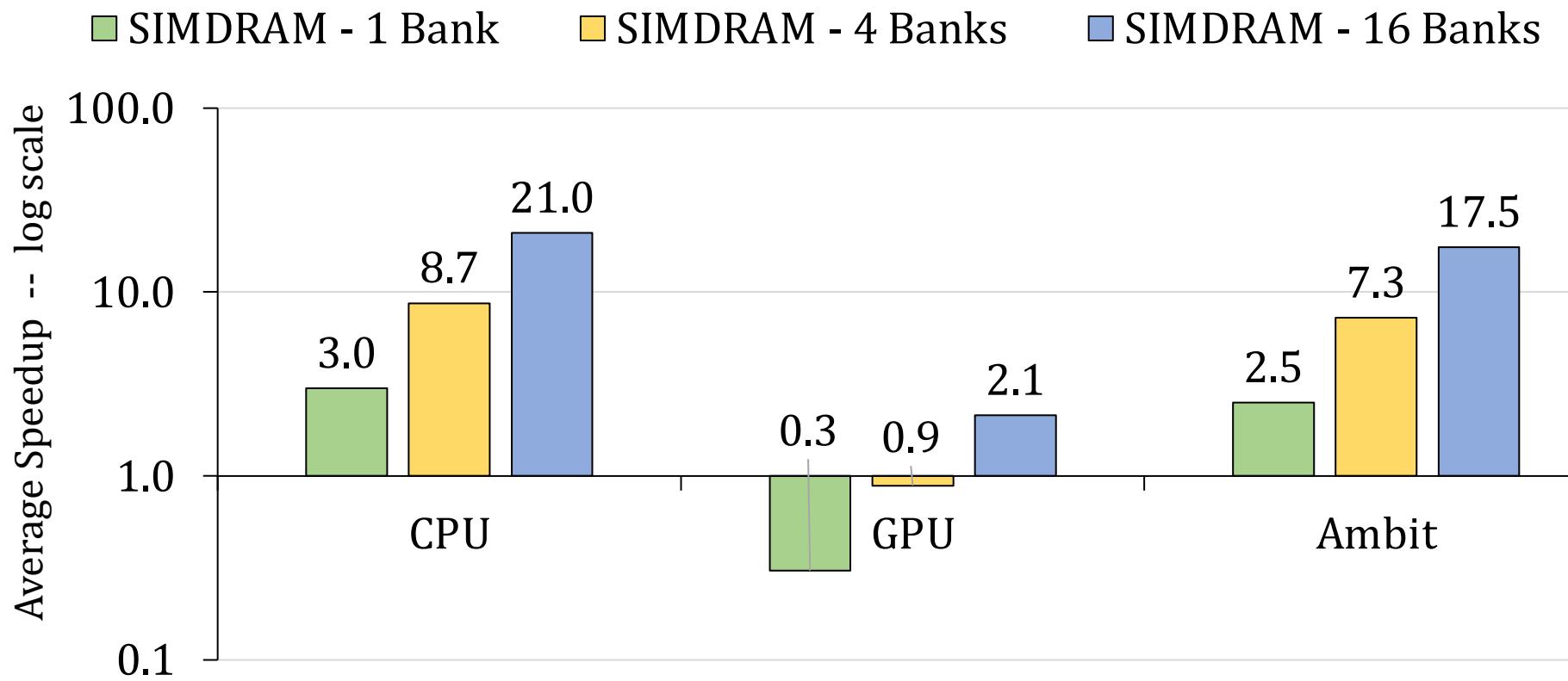
Average normalized energy efficiency across all 16 SIMDRAM operations



**SIMDRAM is more energy-efficient than
all state-of-the-art baselines for a wide range of operations**

Real-World Application

Average speedup across 7 real-world applications



SIMDRAM effectively and efficiently accelerates many commonly-used real-world applications

More in the Paper

- Evaluation:
 - Reliability
 - Data movement overhead
 - Data transposition overhead
 - Area overhead
 - Comparison to in-cache computing

More in the Paper

- Evaluation:

- Reliability

SIMDRAM: An End-to-End Framework for Bit-Serial SIMD Computing in DRAM

*Nastaran Hajinazar^{1,2}

Nika Mansouri Ghiasi¹

*Geraldo F. Oliveira¹

Minesh Patel¹

Juan Gómez-Luna¹

Sven Gregorio¹

Mohammed Alser¹

Onur Mutlu¹

João Dinis Ferreira¹

Saugata Ghose³

¹ETH Zürich

²Simon Fraser University

³University of Illinois at Urbana–Champaign

- Comparison to in-cache computing

Outline

1. Processing-using-DRAM

2. Background

3. SIMD RAM

Processing-using-DRAM Substrate
Framework

4. System Integration

5. Evaluation

6. Conclusion

Conclusion

- **SIMDRAM**: An end-to-end processing-using-DRAM framework that provides the programming interface, the ISA, and the hardware support for:
 1. Efficiently computing complex operations
 2. Providing the ability to implement arbitrary operations as required
 3. Using a massively-parallel in-DRAM SIMD substrate
- **Key Results**: SIMDRAM provides:
 - **88x** and **5.8x** the throughput and **257x** and **31x** the energy efficiency of a baseline CPU and a high-end GPU, respectively, for 16 in-DRAM operations
 - **21x** and **2.1x** the performance of the CPU and GPU over seven real-world applications
- **Conclusion**: SIMDRAM is a promising PuM framework
 - Can ease the adoption of processing-using-DRAM architectures
 - Improve the performance and efficiency of processing-using-DRAM architectures

SIMDRAM: A Framework for Bit-Serial SIMD Processing using DRAM

Computer Architecture, Lecture 24c
Fall 2021



Nastaran Hajinazar*

Geraldo F. Oliveira*

Sven Gregorio

Joao Ferreira

Nika Mansouri Ghiasi

Minesh Patel

Mohammed Alser

Saugata Ghose

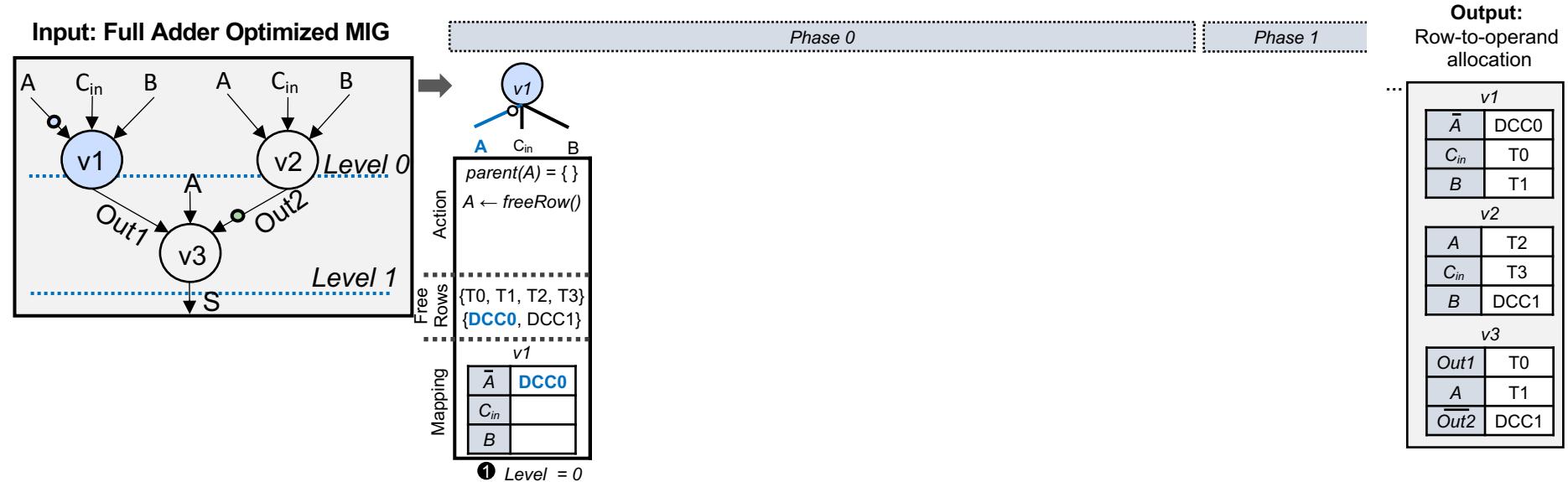
Juan Gómez-Luna

Onur Mutlu

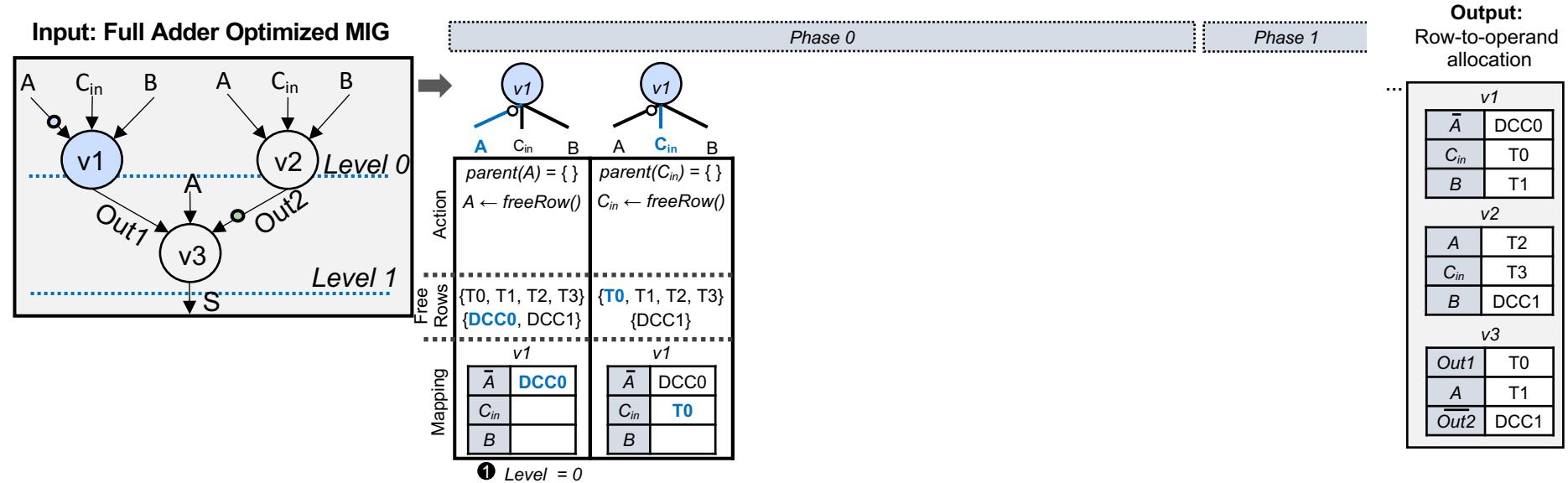
Task 1: Allocating DRAM Rows to Operands



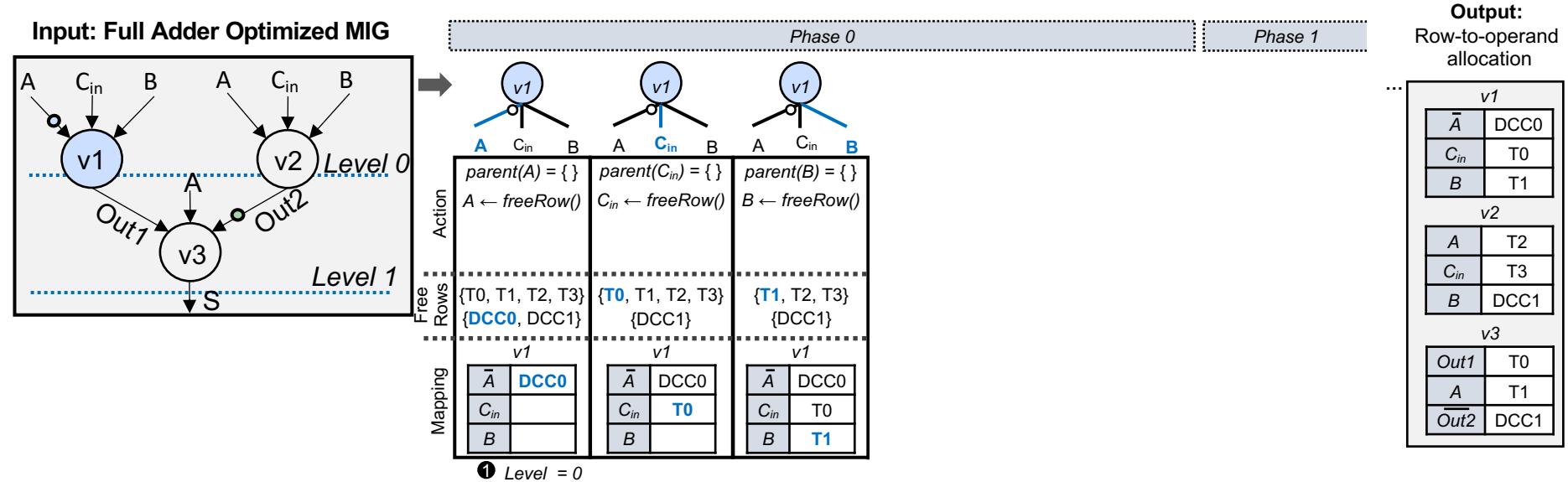
Task 1: Allocating DRAM Rows to Operands



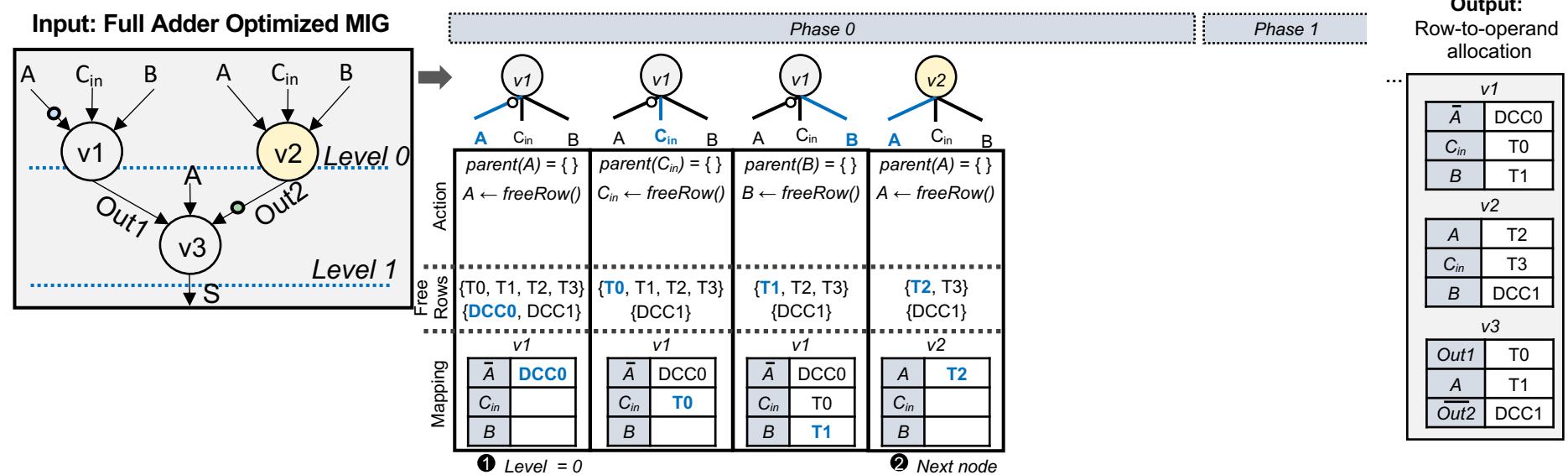
Task 1: Allocating DRAM Rows to Operands



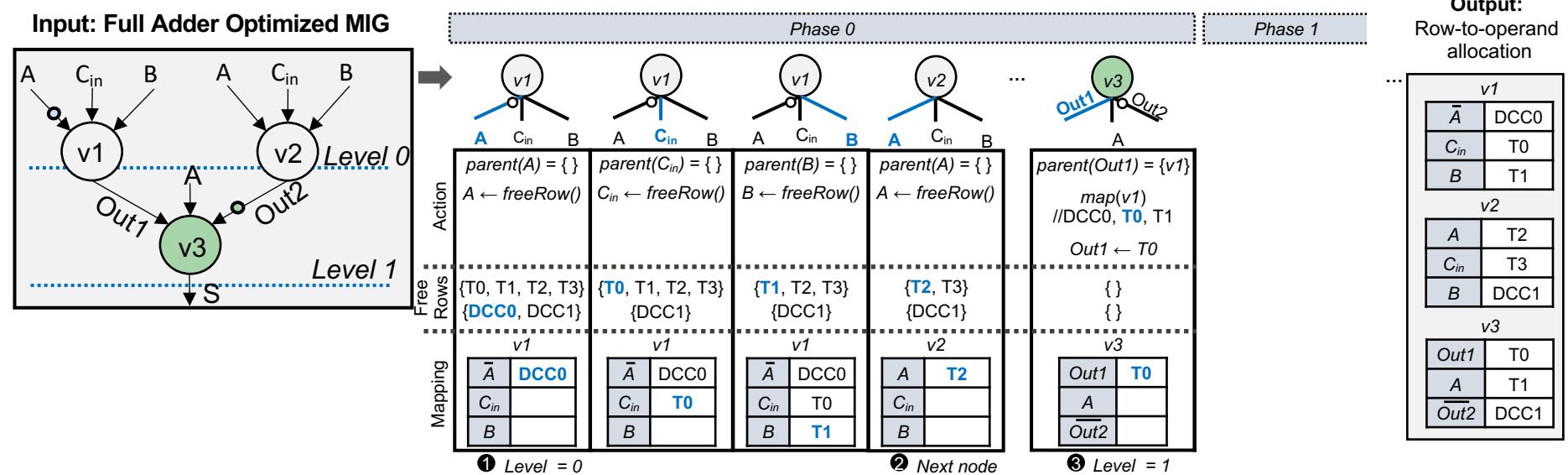
Task 1: Allocating DRAM Rows to Operands



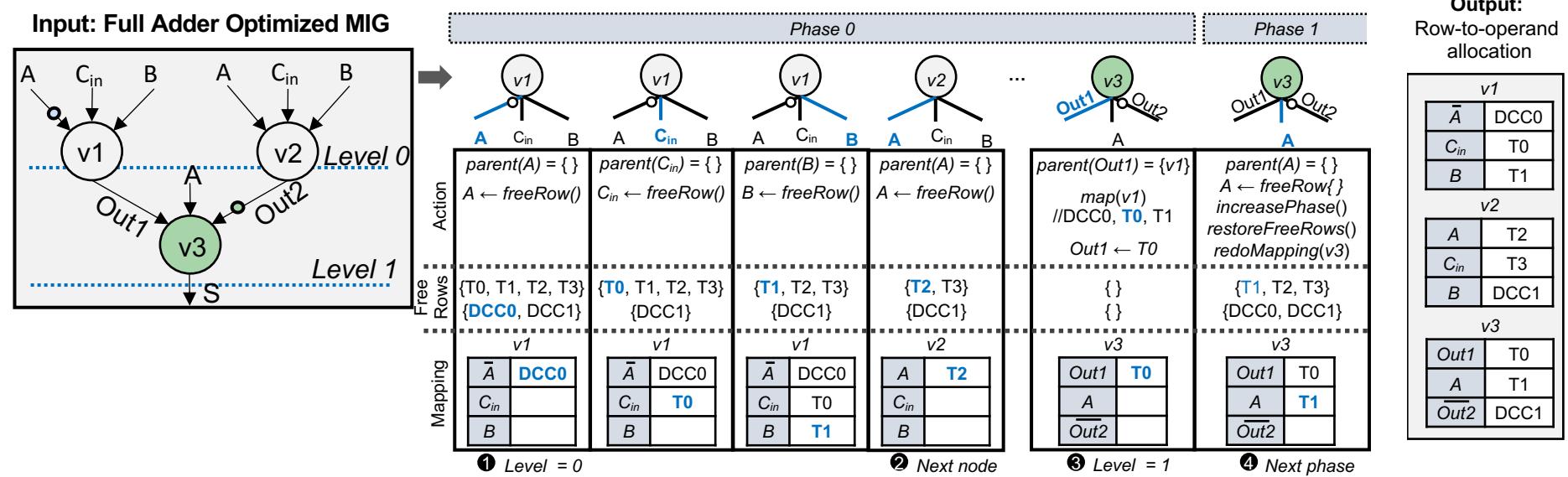
Task 1: Allocating DRAM Rows to Operands



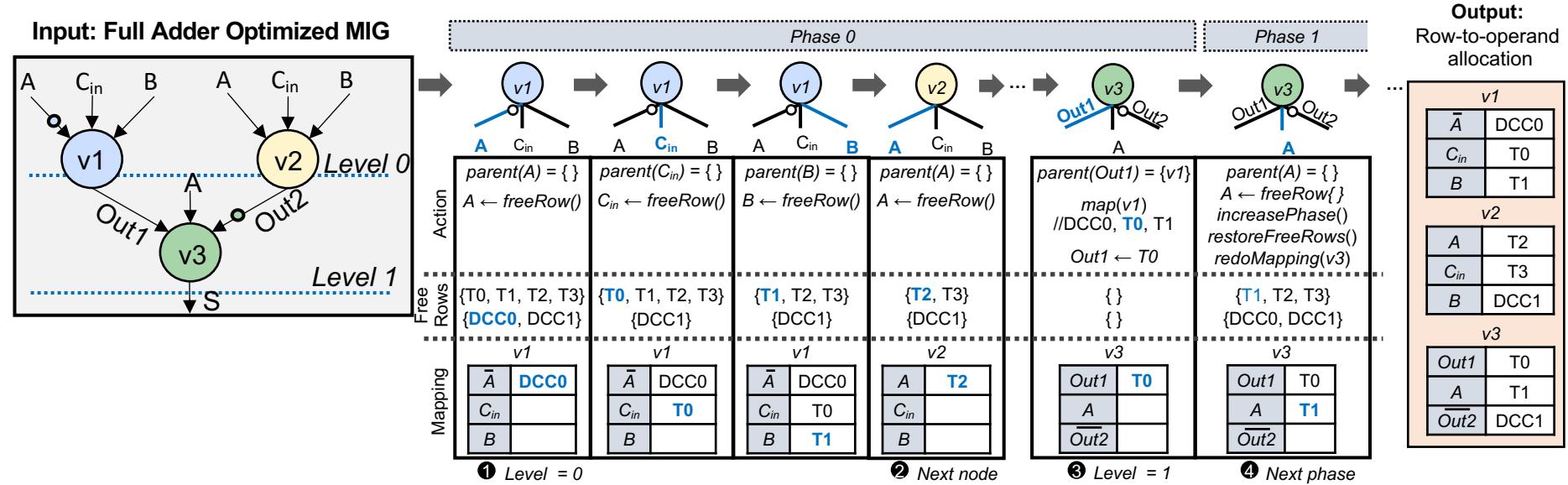
Task 1: Allocating DRAM Rows to Operands



Task 1: Allocating DRAM Rows to Operands

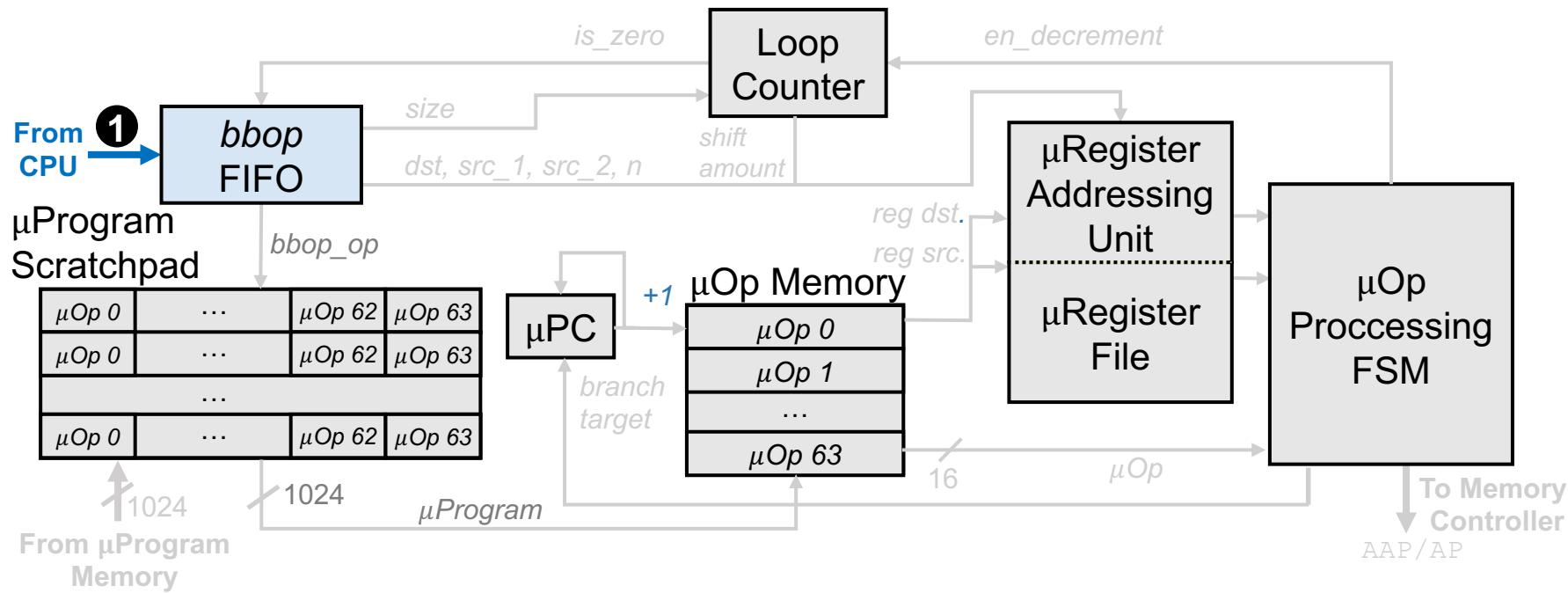


Task 1: Allocating DRAM Rows to Operands



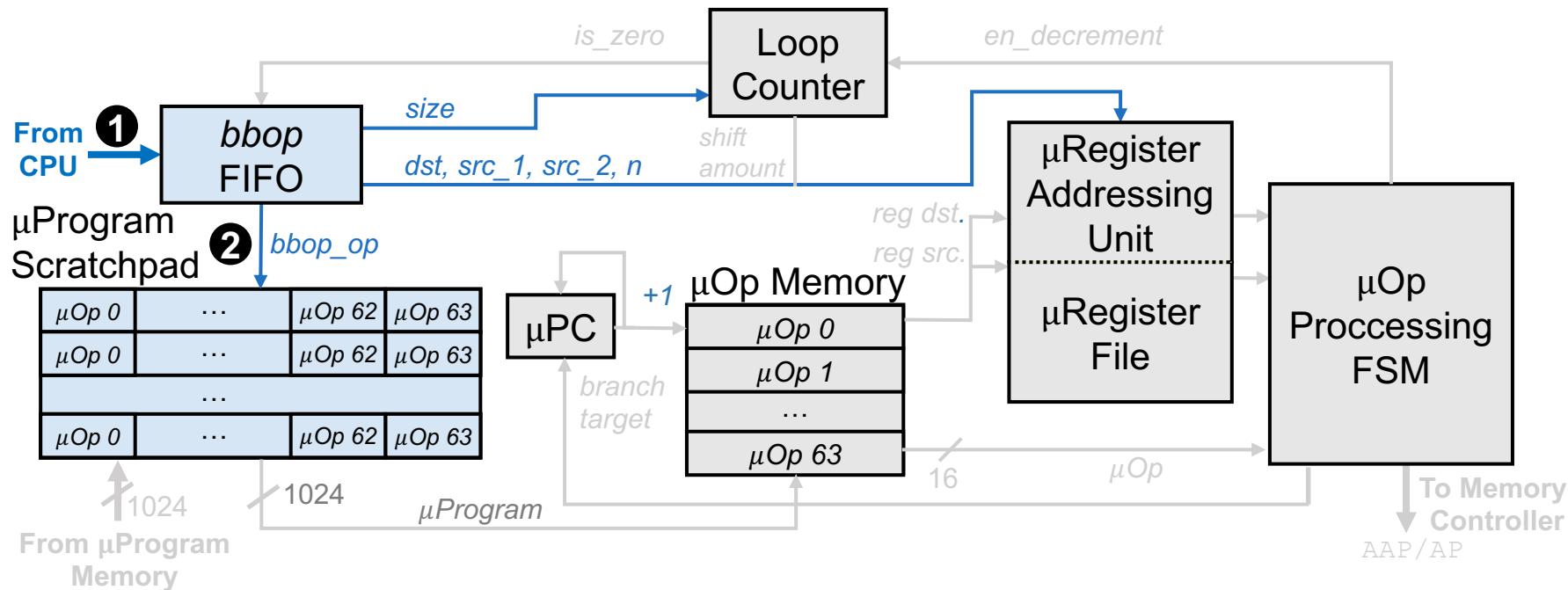
Step 3: Operation Execution

- The SIMDRAAM control unit works as follows:



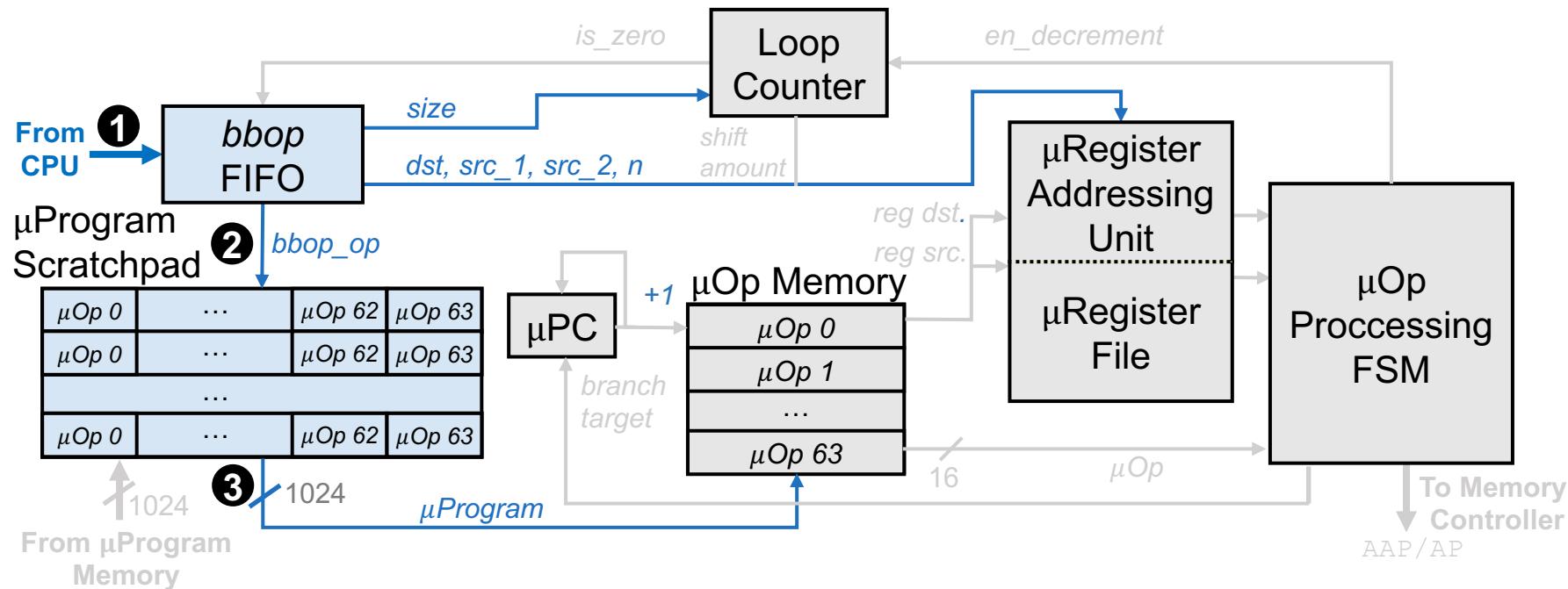
Step 3: Operation Execution

- The SIMDRAAM control unit works as follows:



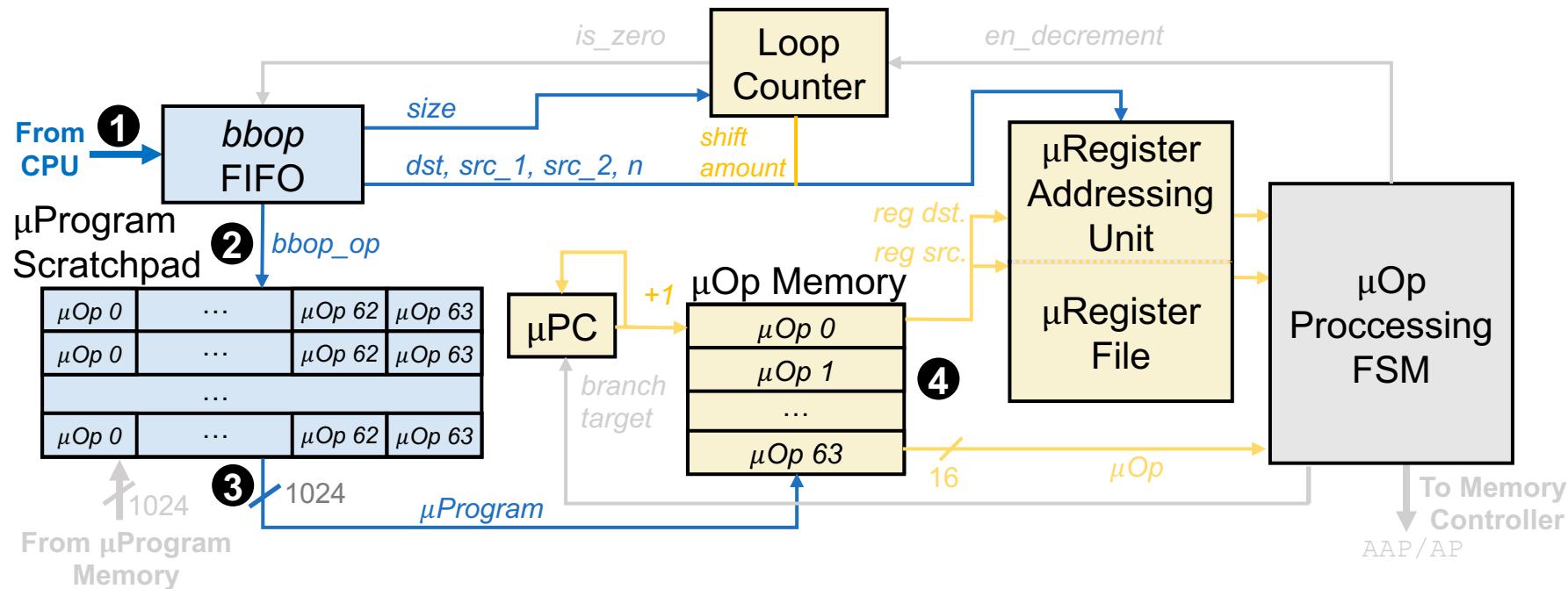
Step 3: Operation Execution

- The SIMDRAAM control unit works as follows:



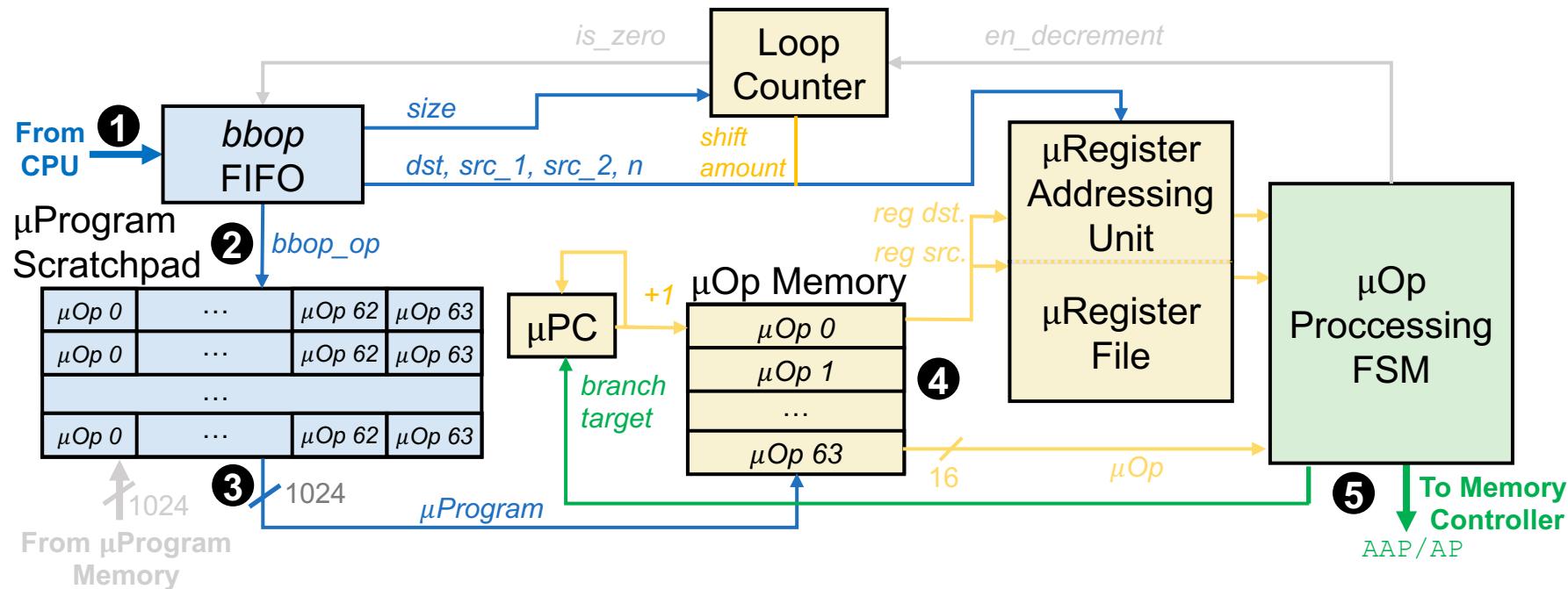
Step 3: Operation Execution

- The SIMDRAAM control unit works as follows:



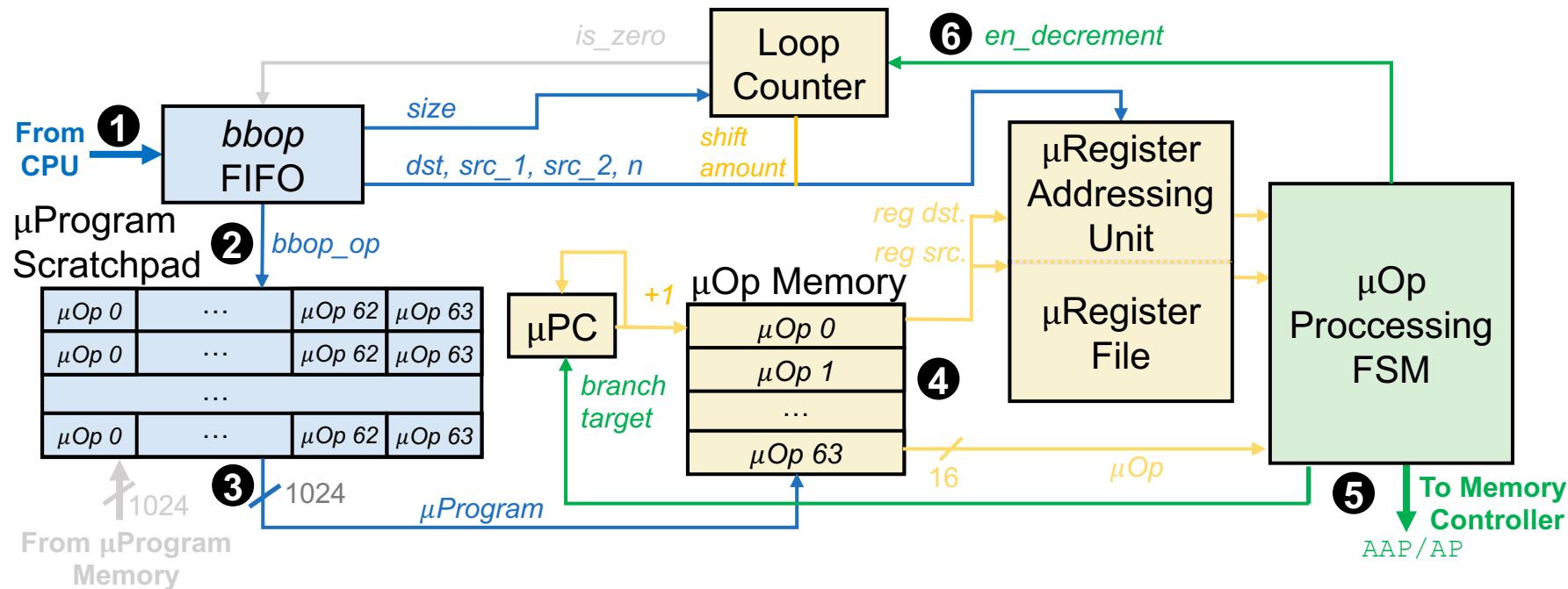
Step 3: Operation Execution

- The SIMD RAM control unit works as follows:



Step 3: Operation Execution

- The SIMDRAAM control unit works as follows:



Step 3: Operation Execution

- The SIMDRAAM control unit works as follows:

