

# P&S Processing-in-Memory

Bit-Serial SIMD Processing  
using DRAM

Dr. Juan Gómez Luna

Prof. Onur Mutlu

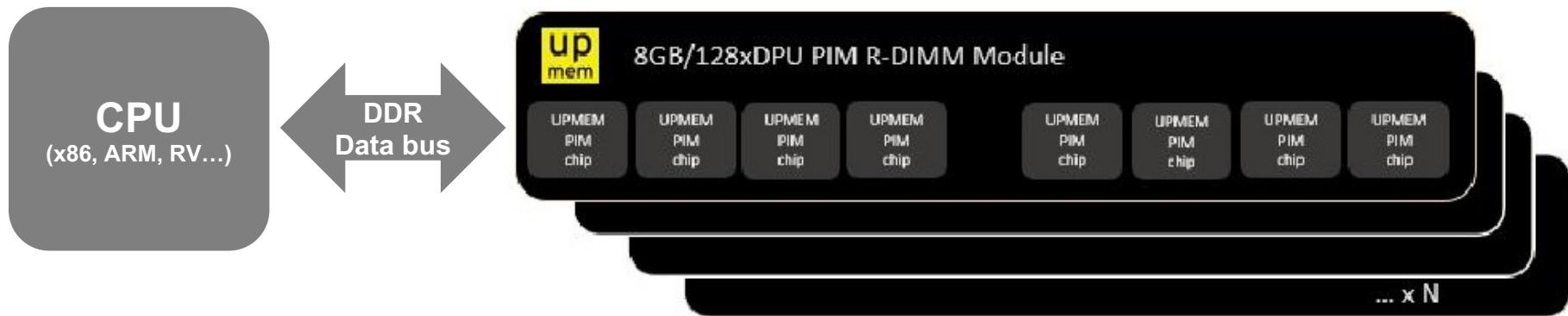
ETH Zürich

Fall 2021

7 December 2021

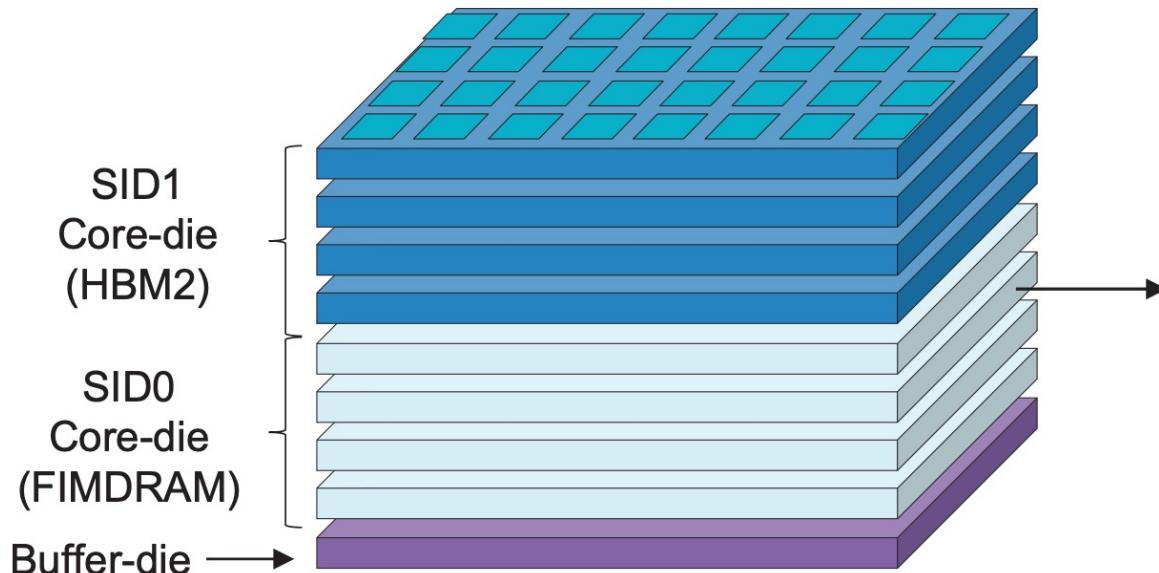
# UPMEM Processing-in-DRAM Engine (2019)

- Processing in DRAM Engine
- Includes **standard DIMM modules**, with a **large number of DPU processors** combined with DRAM chips.
- Replaces **standard** DIMMs
  - DDR4 R-DIMM modules
    - 8GB+128 DPUs (16 PIM chips)
    - Standard 2x-nm DRAM process
  - **Large amounts of** compute & memory bandwidth



# Samsung Function-in-Memory DRAM (2021)

## ■ FIMDRAM based on HBM2



[3D Chip Structure of HBM with FIMDRAM]

### Chip Specification

128DQ / 8CH / 16 banks / BL4

32 PCU blocks (1 FIM block/2 banks)

1.2 TFLOPS (4H)

**FP16 ADD /  
Multiply (MUL) /  
Multiply-Accumulate (MAC) /  
Multiply-and- Add (MAD)**

ISSCC 2021 / SESSION 25 / DRAM / 25.4

25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications

Young-Cheon Kwon<sup>1</sup>, Suk Han Lee<sup>1</sup>, Jaehoon Lee<sup>1</sup>, Sang-Hyuk Kwon<sup>1</sup>, Je Min Ryu<sup>1</sup>, Jong-Pil Son<sup>1</sup>, Seongil Oh<sup>1</sup>, Hak-Soo Yu<sup>1</sup>, Haesuk Lee<sup>1</sup>, Soo Young Kim<sup>1</sup>, Younghmin Cho<sup>1</sup>, Jin Guk Kim<sup>1</sup>, Jongyoon Choi<sup>1</sup>, Hyun-Sung Shin<sup>1</sup>, Jin Kim<sup>1</sup>, BengSeng Phua<sup>2</sup>, Hyo young Min Kim<sup>1</sup>, Myeong Jun Song<sup>1</sup>, Ahn Choi<sup>1</sup>, Daeho Kim<sup>1</sup>, Soo Young Kim<sup>1</sup>, Eun-Bong Kim<sup>1</sup>, David Wang<sup>2</sup>, Shinhwa Kang<sup>1</sup>, Yuhwan Ro<sup>3</sup>, Seungwoo Seo<sup>3</sup>, JoonHo Song<sup>3</sup>, Jaeyoun Youn<sup>1</sup>, Kyomin Sohn<sup>1</sup>, Nam Sung Kim<sup>1</sup>

<sup>1</sup>Samsung Electronics, Hwaseong, Korea

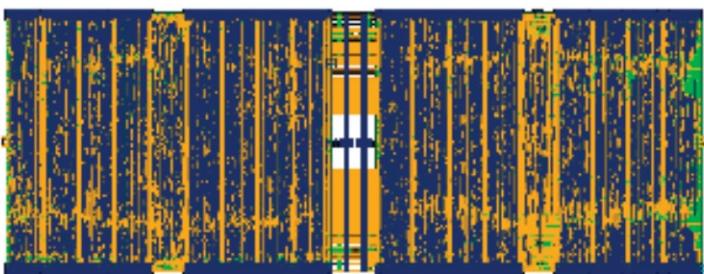
<sup>2</sup>Samsung Electronics, San Jose, CA

<sup>3</sup>Samsung Electronics, Suwon, Korea

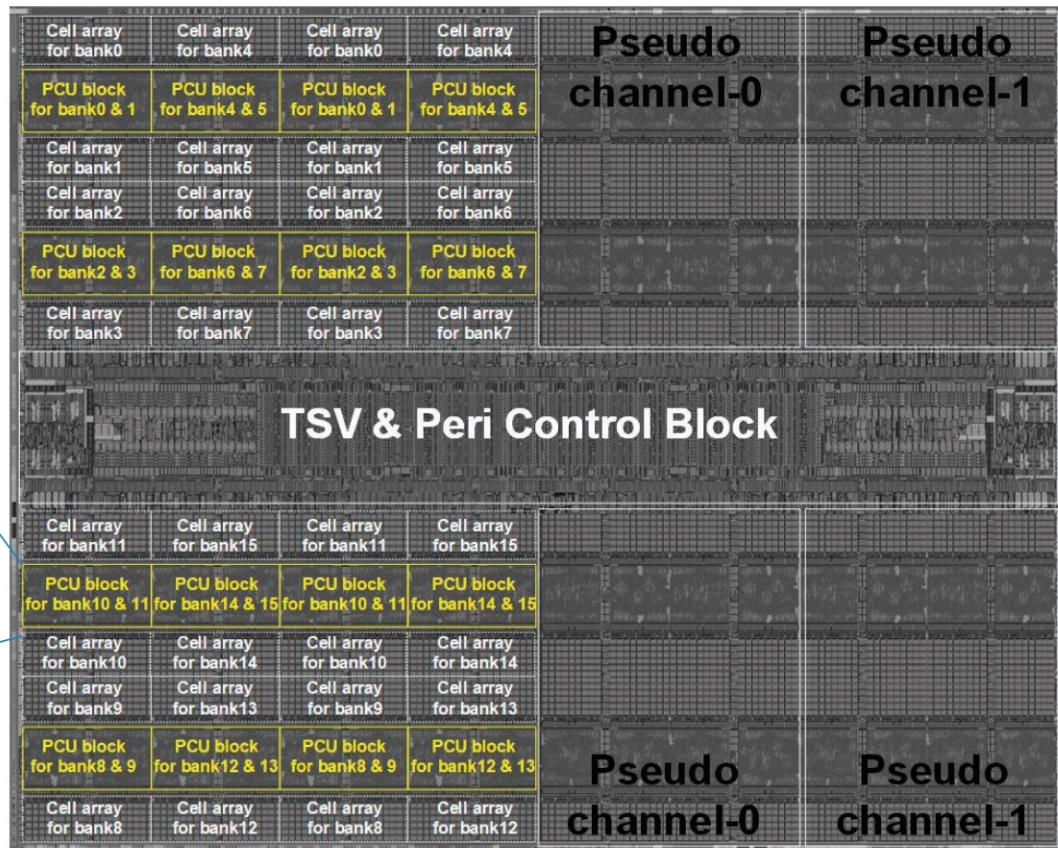
# Samsung Function-in-Memory DRAM (2021)

## Chip Implementation

- Mixed design methodology to implement FIMDRAM
  - Full-custom + Digital RTL



[Digital RTL design for PCU block]



ISSCC 2021 / SESSION 25 / DRAM / 25.4

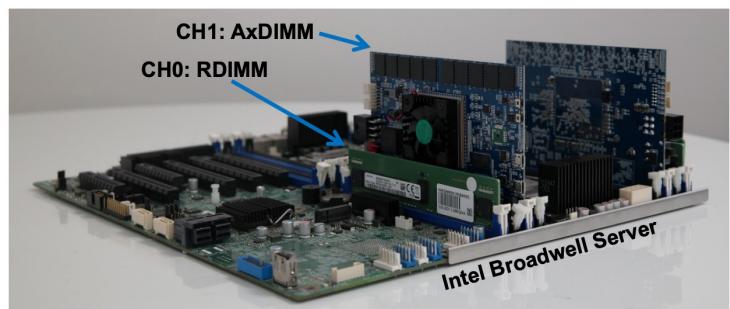
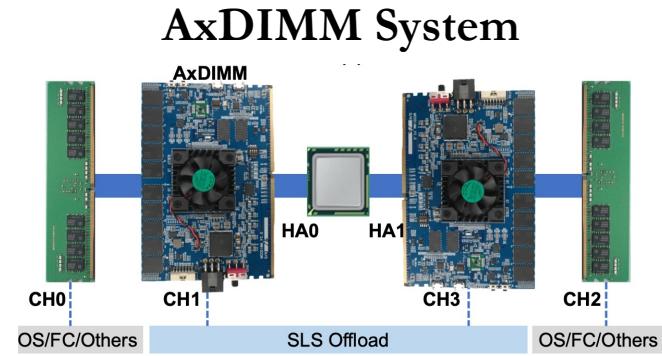
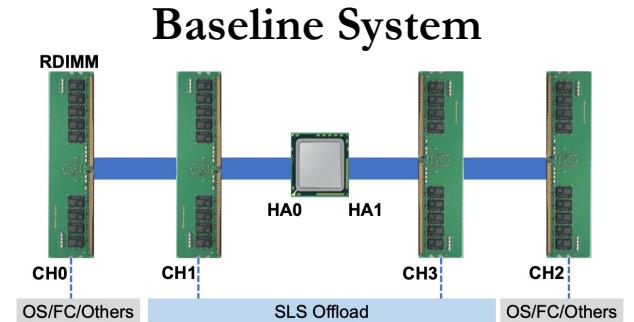
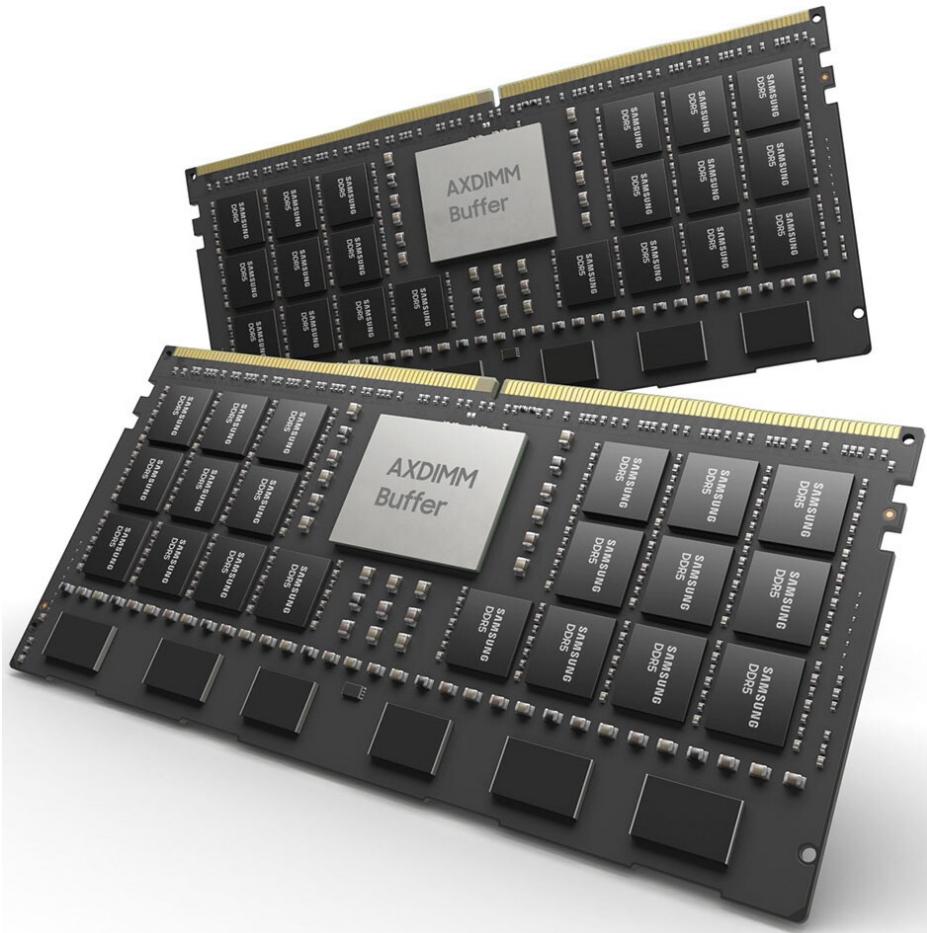
25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications

Young-Cheon Kwon<sup>1</sup>, Suk Han Lee<sup>1</sup>, Jaehoon Lee<sup>1</sup>, Sang-Hyuk Kwon<sup>1</sup>, Je Min Ryu<sup>1</sup>, Jong-Pil Son<sup>1</sup>, Seongil O<sup>2</sup>, Hak-Soo Yu<sup>1</sup>, Haesuk Lee<sup>1</sup>, Soo Young Kim<sup>1</sup>, Youngmin Cho<sup>1</sup>, Jin Guk Kim<sup>1</sup>, Jongyoon Choi<sup>1</sup>, Hyun-Sung Shin<sup>1</sup>, Jin Kim<sup>1</sup>, BengSeng Phua<sup>3</sup>, HyoungMin Kim<sup>1</sup>, Myeong Jun Song<sup>1</sup>, Ahn Choi<sup>1</sup>, Daeho Kim<sup>1</sup>, SooYoung Kim<sup>1</sup>, Eun-Bong Kim<sup>1</sup>, David Wang<sup>1</sup>, Shinhwang Kang<sup>1</sup>, Yuhwan Ro<sup>1</sup>, Seungwoo Seo<sup>1</sup>, JoonHo Song<sup>1</sup>, Jaeyoun Youn<sup>1</sup>, Kyomin Sohn<sup>1</sup>, Nam Sung Kim<sup>1</sup>

<sup>1</sup>Samsung Electronics, Hwasung, Korea  
<sup>2</sup>Samsung Electronics, San Jose, CA  
<sup>3</sup>Samsung Electronics, Suwon, Korea

# Samsung AxDIMM (2021)

- DIMM-based PIM
  - DLRM recommendation system



# Processing in Memory: Two Approaches

1. Processing near Memory
2. Processing using Memory

# Two PIM Approaches

---

## 5.2. Two Approaches: Processing Using Memory (PUM) vs. Processing Near Memory (PNM)

Many recent works take advantage of the memory technology innovations that we discuss in Section 5.1 to enable and implement PIM. We find that these works generally take one of two approaches, which are categorized in Table 1: (1) *processing using memory* or (2) *processing near memory*. We briefly describe each approach here. Sections 6 and 7 will provide example approaches and more detail for both.

Table 1: Summary of enabling technologies for the two approaches to PIM used by recent works. Adapted from [309].

Approach	Enabling Technologies
Processing Using Memory	SRAM
	DRAM
	Phase-change memory (PCM)
	Magnetic RAM (MRAM) Resistive RAM (RRAM)/memristors
Processing Near Memory	Logic layers in 3D-stacked memory
	Silicon interposers
	Logic in memory controllers

**Processing using memory (PUM)** exploits the existing memory architecture and the operational principles of the memory circuitry to enable operations within main memory with minimal changes. PUM makes use

Onur Mutlu, Saugata Ghose, Juan Gomez-Luna, and Rachata Ausavarungnirun,

[\*\*"A Modern Primer on Processing in Memory"\*\*](#)

*Invited Book Chapter in [Emerging Computing: From Devices to Systems - Looking Beyond Moore and Von Neumann](#), Springer, to be published in 2021.*

[\[Tutorial Video on "Memory-Centric Computing Systems"\]](#) (1 hour 51 minutes)]

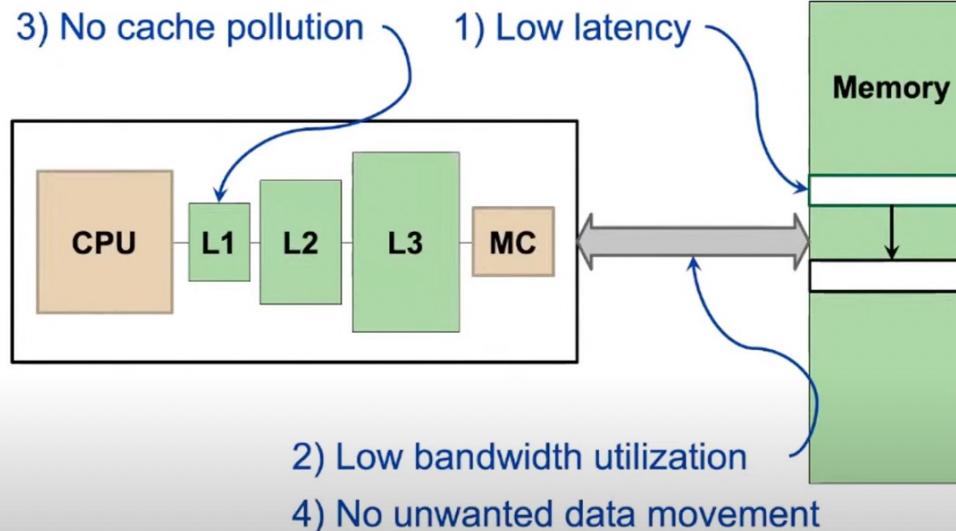
# Processing Using Memory

---

- Take advantage of operational principles of memory to perform **bulk data movement and computation in memory**
  - Can **exploit internal connectivity** to move data
  - Can **exploit analog computation capability**
  - ...
- Examples: RowClone, In-DRAM AND/OR, Gather/Scatter DRAM
  - [RowClone: Fast and Efficient In-DRAM Copy and Initialization of Bulk Data](#) (Seshadri et al., MICRO 2013)
  - [Fast Bulk Bitwise AND and OR in DRAM](#) (Seshadri et al., IEEE CAL 2015)
  - [Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-unit Strided Accesses](#) (Seshadri et al., MICRO 2015)
  - ["Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology"](#) (Seshadri et al., MICRO 2017)

# Lecture on Processing Using DRAM (I)

## Future Systems: In-Memory Copy



~~1046ns, 3.6uJ → 90ns, 0.04uJ~~

ETH ZURICH D-ITET

Processing in Memory Course: Meeting 1: Exploring the PIM Paradigm for Future Systems - Fall'21

1,210 views • Streamed live on Oct 5, 2021

43 DISLIKE SHARE SAVE ...



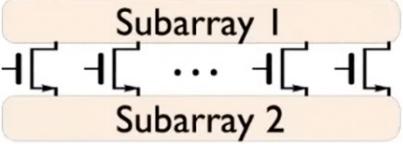
Onur Mutlu Lectures  
20.5K subscribers

SUBSCRIBED



# Lecture on Processing Using DRAM (II)

## Key Idea and Applications

- **Low-cost Inter-linked subarrays (LISA)**
  - Fast bulk data movement between subarrays
  - Wide datapath via isolation transistors: 0.8% DRAM chip area
- LISA is a **versatile substrate** → new applications
  - Fast bulk data copy: Copy latency 1.363ms → 0.148ms (9.2x)  
→ 66% speedup, -55% DRAM energy
  - In-DRAM caching: Hot data access latency 48.7ns → 21.5ns (2.2x)  
→ 5% speedup

ETH ZURICH D-ITET

Computer Architecture - Lecture 6: Processing using Memory (Fall 2021)

880 views • Streamed live on Oct 15, 2021

30 DISLIKE SHARE SAVE ...



Onur Mutlu Lectures  
20.5K subscribers

SUBSCRIBED



# Processing-using-Memory in Real DRAM Chips

---

## ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs

Fei Gao

feig@princeton.edu

Department of Electrical Engineering  
Princeton University

Georgios Tziantzioulis

georgios.tziantzioulis@princeton.edu

Department of Electrical Engineering  
Princeton University

David Wentzlaff

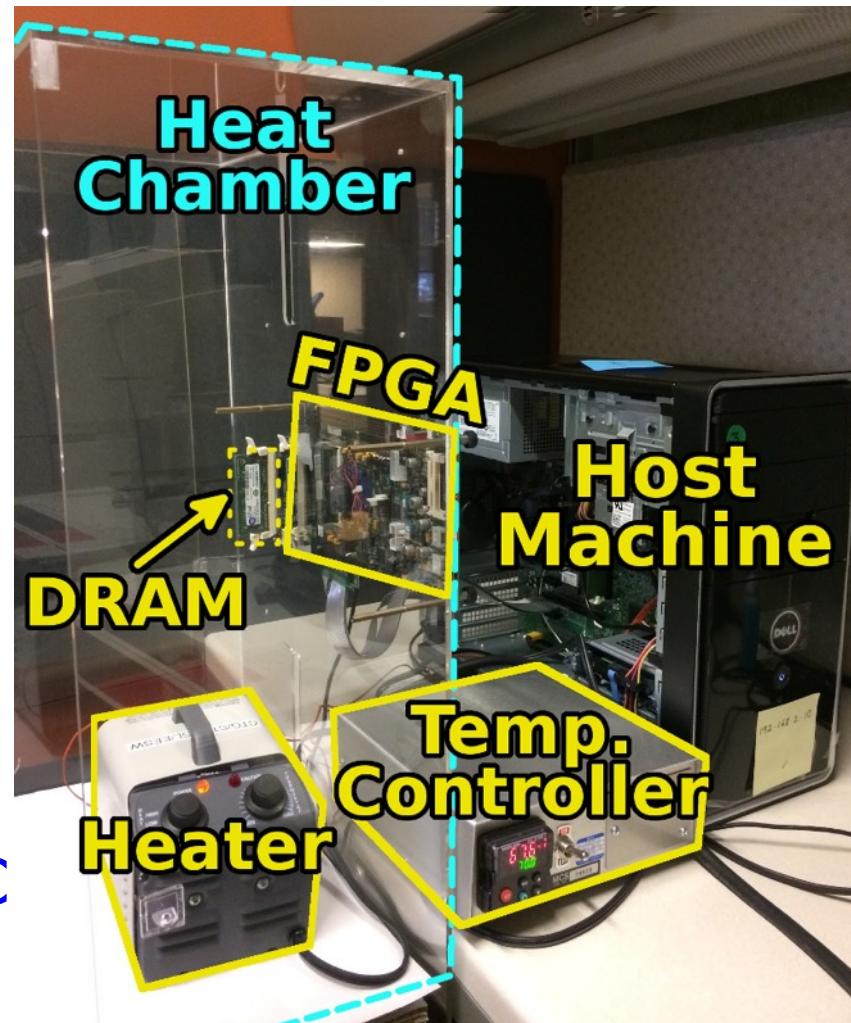
wentzlaf@princeton.edu

Department of Electrical Engineering  
Princeton University

# SoftMC: Open Source DRAM Infrastructure

- Hasan Hassan et al., “**SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies,**” HPCA 2017

- Flexible
- Easy to Use (C++ API)
- Open-source  
[github.com/CMU-SAFARI/SoftMC](https://github.com/CMU-SAFARI/SoftMC)



# RowClone & Bitwise Ops in Real DRAM Chips

MICRO-52, October 12–16, 2019, Columbus, OH, USA

Gao et al.

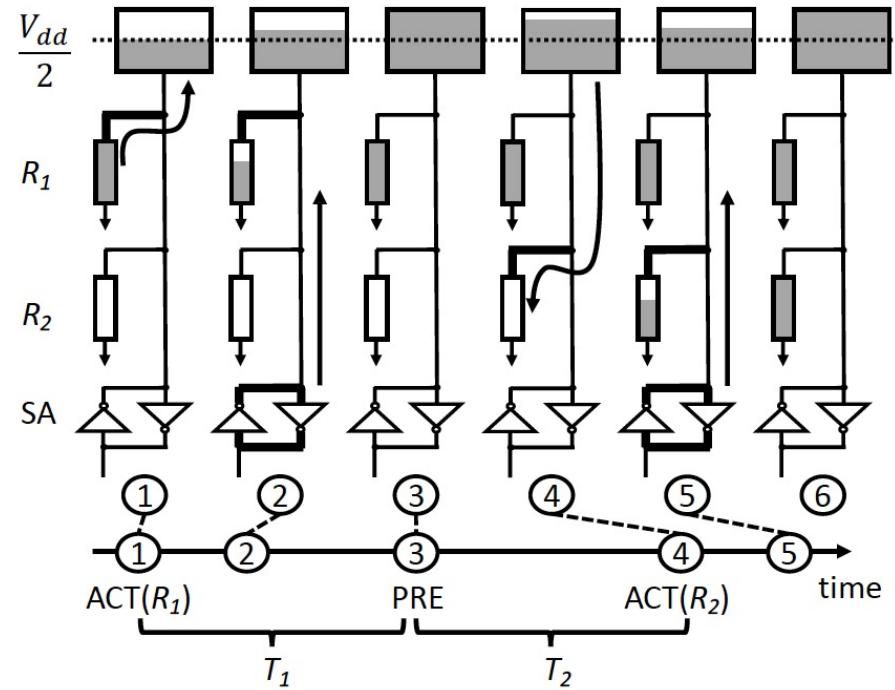


Figure 4: Timeline for a single bit of a column in a row copy operation. The data in  $R_1$  is loaded to the bit-line, and overwrites  $R_2$ .

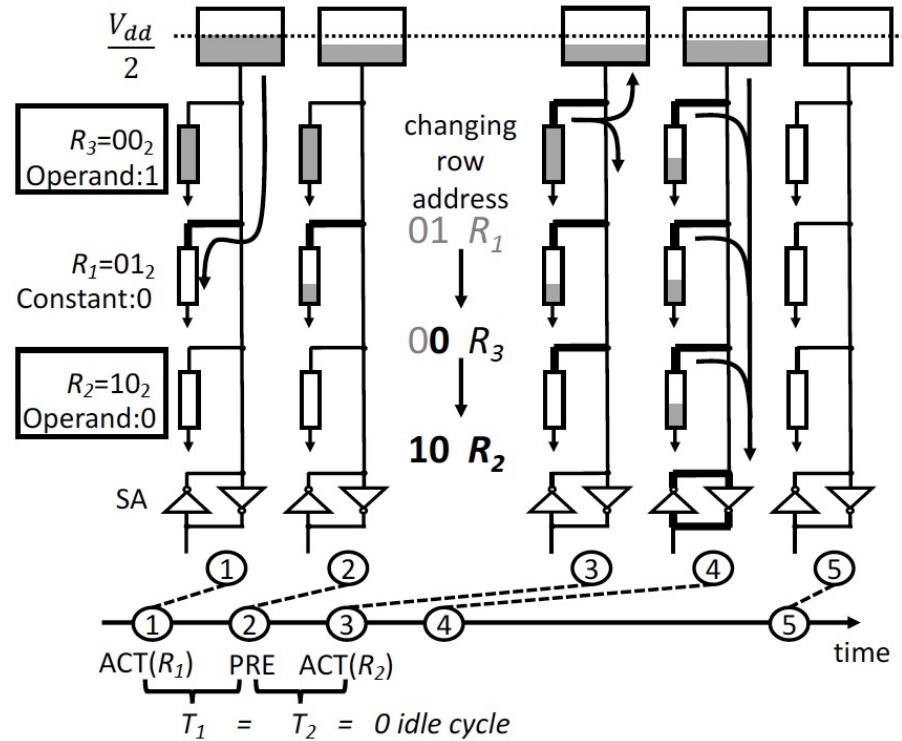


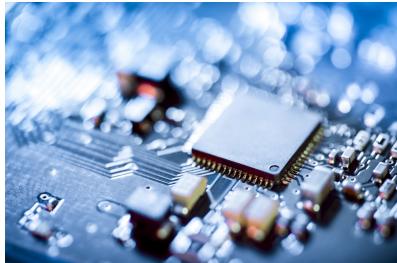
Figure 5: Logical AND in ComputeDRAM.  $R_1$  is loaded with constant zero, and  $R_2$  and  $R_3$  store operands (0 and 1). The result ( $0 = 1 \wedge 0$ ) is finally set in all three rows.

# PiDRAM

**Goal:** Develop a **flexible** platform to explore end-to-end implementations of PuM techniques

- Enable rapid integration via key components

## Hardware



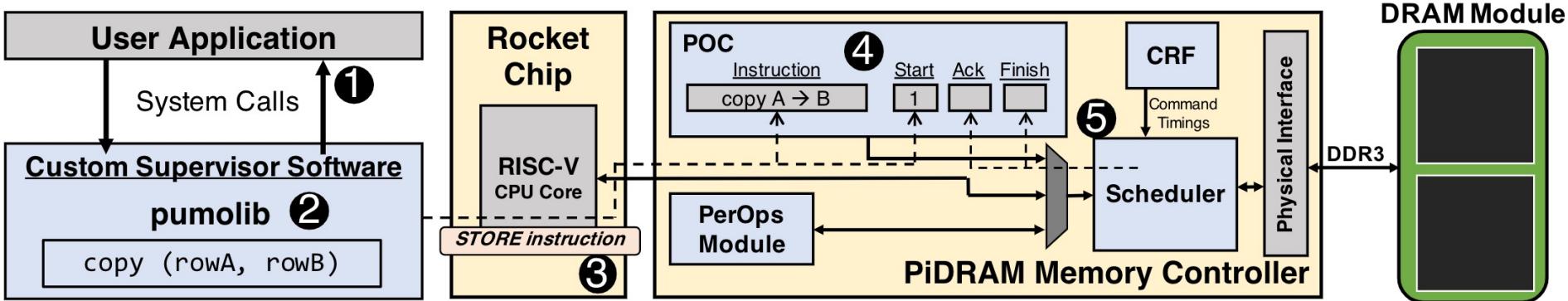
- ① Easy-to-extend Memory Controller
- ② ISA-transparent PuM Controller

## Software



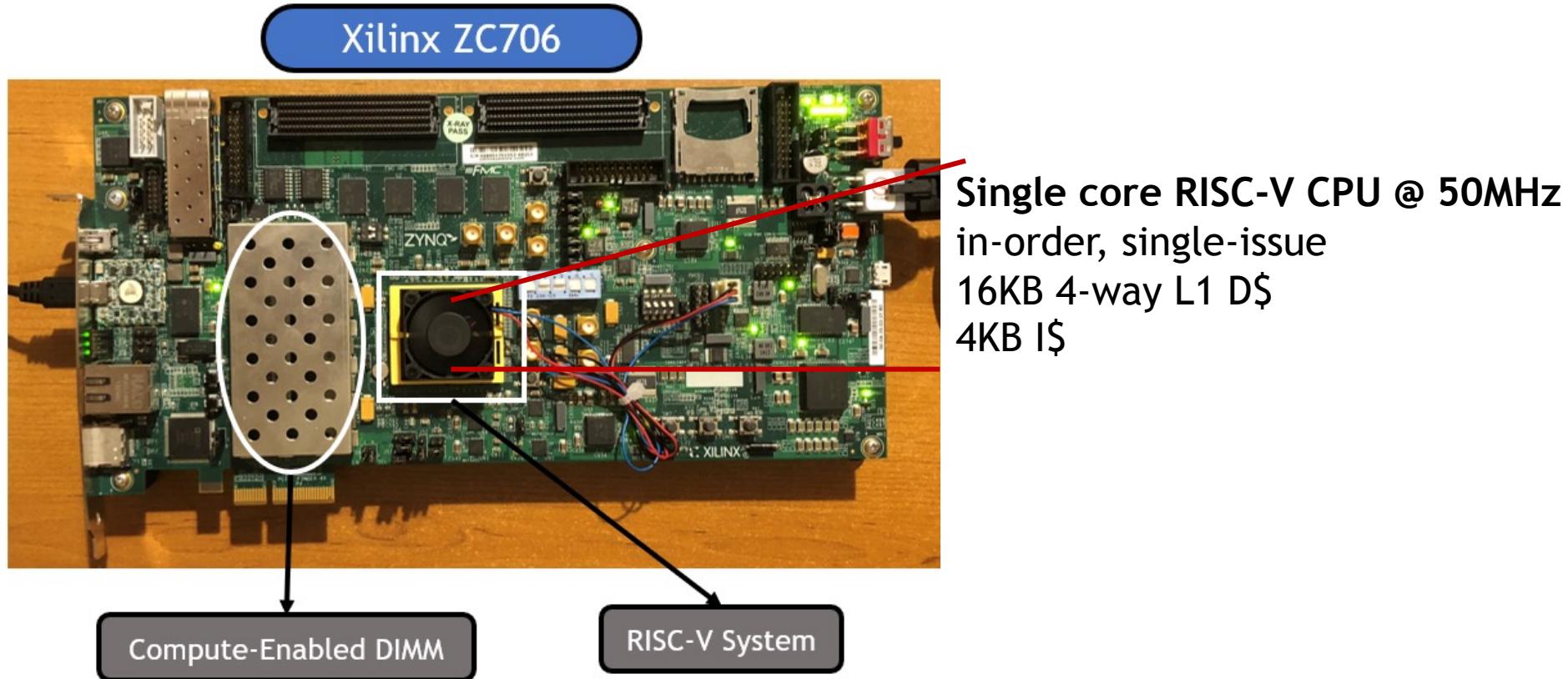
- ① Extensible Software Library
- ② Custom Supervisor Software

# PiDRAM Workflow



- 1- User application interfaces with the OS via system calls
- 2- OS uses PuM Operations Library (pumolib) to convey operation related information to the hardware using
- 3- STORE instructions that target the memory mapped registers of the PuM Operations Controller (POC)
- 4- POC oversees the execution of a PuM operation (e.g., RowClone, bulk bitwise operations)
- 5- Scheduler arbitrates between regular (load, store) and PuM operations and issues DRAM commands with custom timings

# PiDRAM FPGA Prototype



# Lecture on PiDRAM

## PiDRAM: PuM Controller (I)

### PuM Operations Controller (POC)

Provide ISA-transparent control for PuM operations

- Connected as a memory-mapped module
  - Hierarchically, resides within the memory controller
- Simple Interface: Offload 128-bit instructions

The diagram illustrates the architecture of the PiDRAM Memory Controller. It starts with a **Rocket Chip** containing a **RISC-V CPU Core**. A **STORE instruction** is issued from the CPU. This instruction is processed by a **PerOps Module** within the **PiDRAM Memory Controller**. The PerOps Module generates a **copy A → B** instruction for the **PuM Operations Controller**. The PuM Operations Controller handles the instruction and sends **Start**, **Ack**, and **Finish** signals back to the PerOps Module. Simultaneously, the PuM Operations Controller interacts with a **Scheduler** and a **CRF** (Command Response Filter) via a **Physical Interface**. The Scheduler manages the execution of operations, and the CRF filters commands based on timing requirements.

Processing in Memory Course: Meeting 6: End-to-end Framework for Processing-using-Memory - Fall'21

431 views • Streamed live on Nov 9, 2021

20 DISLIKE SHARE SAVE ...



Onur Mutlu Lectures

20.5K subscribers

SUBSCRIBED



# PiDRAM Paper and Repo

## PiDRAM: A Holistic End-to-end FPGA-based Framework for Processing-in-DRAM

Ataberk Olgun<sup>§†</sup>

Juan Gómez Luna<sup>§</sup>

Konstantinos Kanellopoulos<sup>§</sup>

Behzad Salami<sup>§\*</sup>

Hasan Hassan<sup>§</sup>

Oğuz Ergin<sup>†</sup>

Onur Mutlu<sup>§</sup>

<sup>§</sup>ETH Zürich

<sup>†</sup>TOBB ETÜ

<sup>\*</sup>BSC

<https://arxiv.org/pdf/2111.00082.pdf>

The screenshot shows the GitHub repository page for CMU-SAFARI / PiDRAM. The repository is public. The code tab is selected. There is 1 branch and 0 tags. Recent commits include:

- olgunataberk Fix small mistake in README 46522cc 23 hours ago 11 commits
- controller-hardware Add files via upload 25 days ago
- fpga-zynq Adds instructions to reproduce two key results 23 hours ago
- README.md Fix small mistake in README 23 hours ago

A preview of the README.md file is shown, containing the title "PiDRAM" and a description of the repository as the first flexible end-to-end framework for real Processing-using-Memory (Pum) techniques.

<https://github.com/CMU-SAFARI/PiDRAM>

PiDRAM is the first flexible end-to-end framework that enables system integration studies and evaluation of real Processing-using-Memory (Pum) techniques. PiDRAM, at a high level, comprises a RISC-V system and a custom memory controller that can perform Pum operations in real DDR3 chips. This repository contains all sources required to build PiDRAM and develop its prototype on the Xilinx ZC706 FPGA boards.

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

**Nastaran Hajinazar\***

Sven Gregorio

Joao Ferreira

**Geraldo F. Oliveira\***

Nika Mansouri Ghiasi

Minesh Patel

Mohammed Alser

Saugata Ghose

Juan Gómez-Luna

Onur Mutlu

**SAFARI**



SIMON FRASER  
UNIVERSITY

**ETH** Zürich



UNIVERSITY OF  
**ILLINOIS**  
URBANA-CHAMPAIGN

# 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

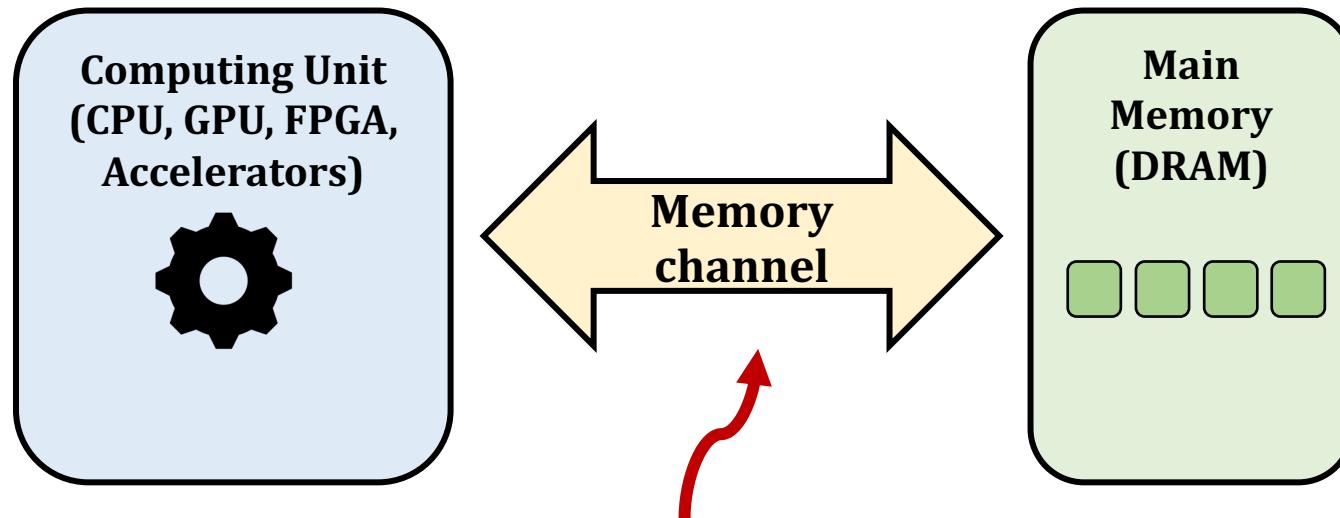
## 5. Evaluation

## 6. Conclusion

# Data Movement Bottleneck

- Data movement is a major bottleneck

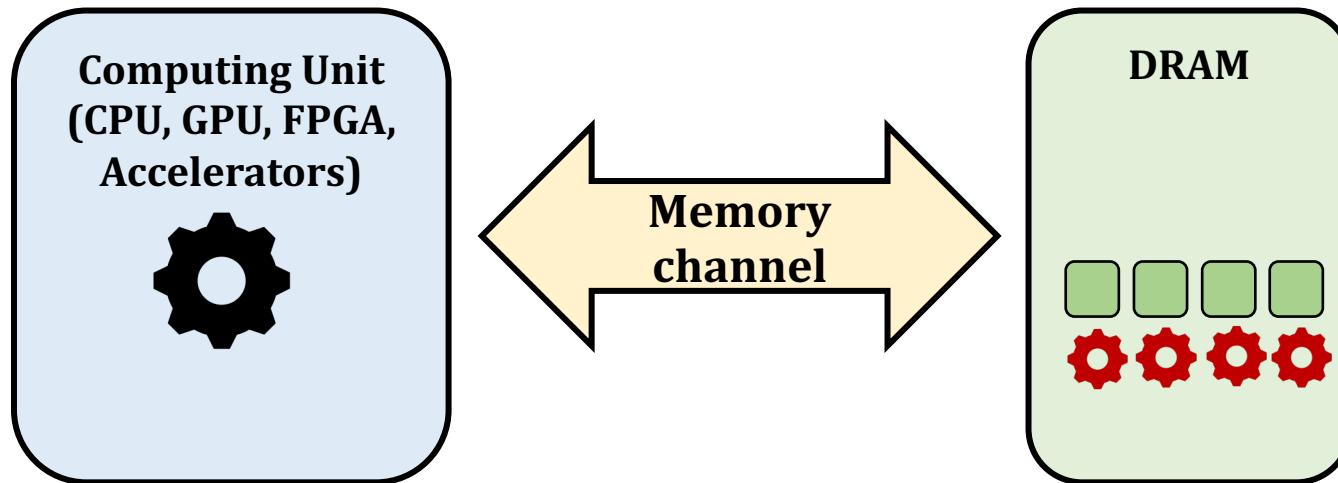
More than **60%** of the total system energy  
is spent on **data movement**<sup>1</sup>



**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

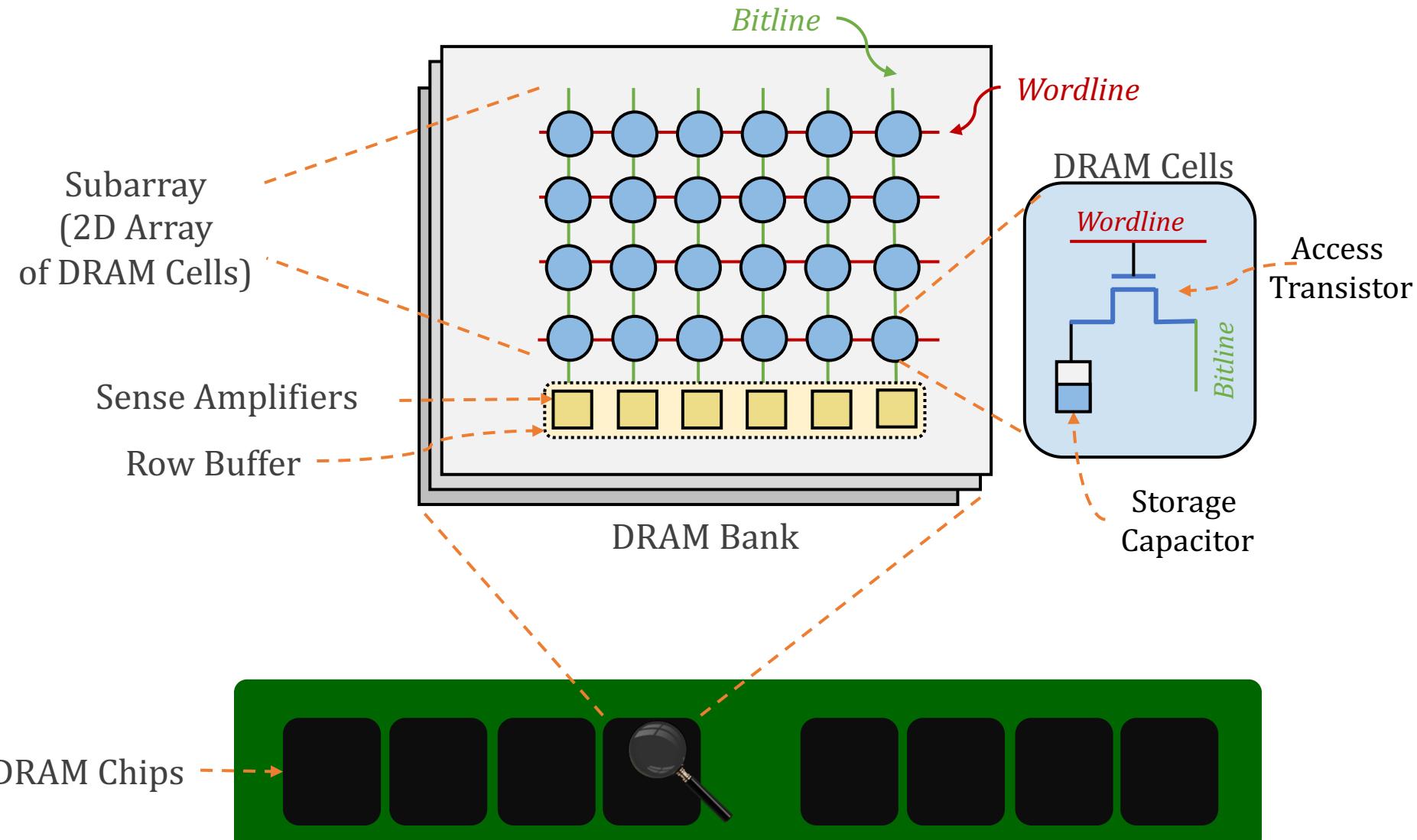
Processing-using-DRAM Substrate  
Framework

## 4. System Integration

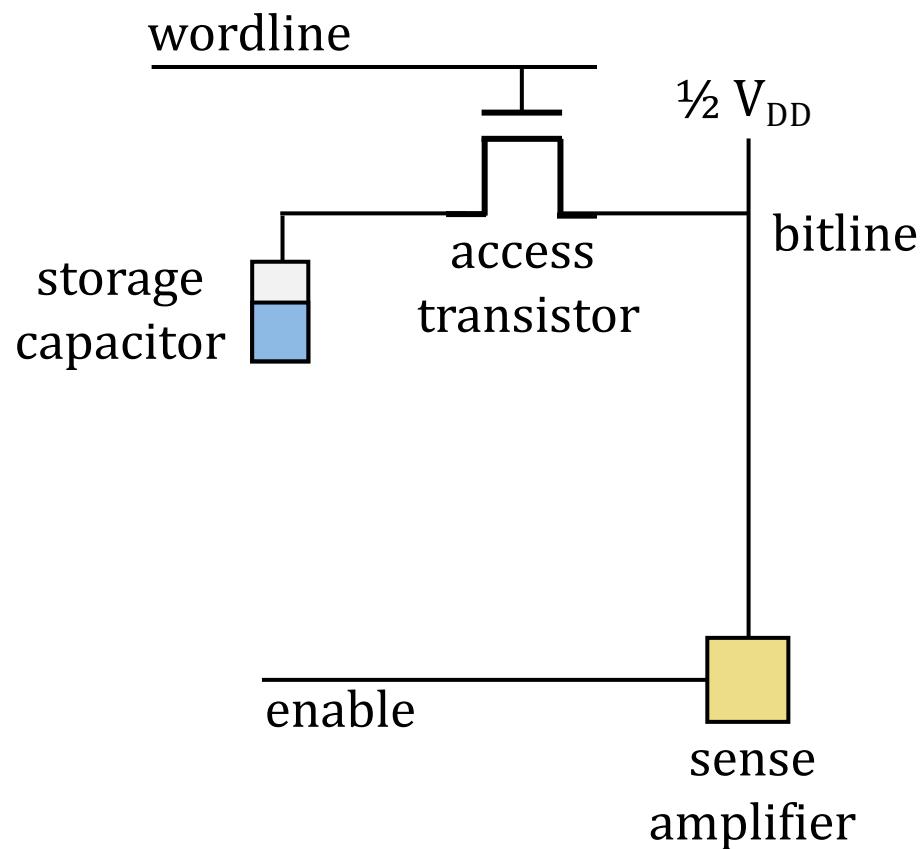
## 5. Evaluation

## 6. Conclusion

# Inside a DRAM Chip



# DRAM Cell Operation

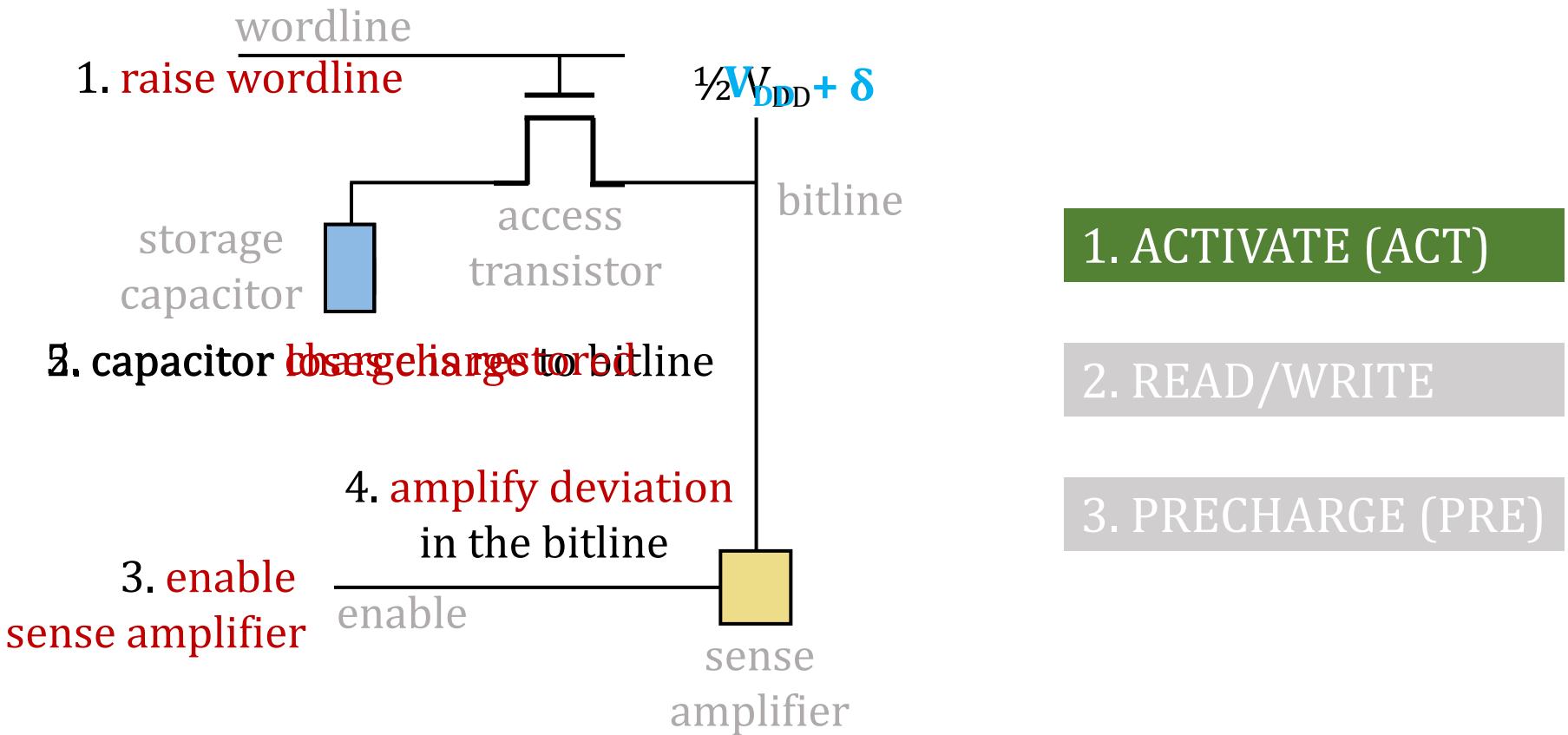


1. ACTIVATE (ACT)

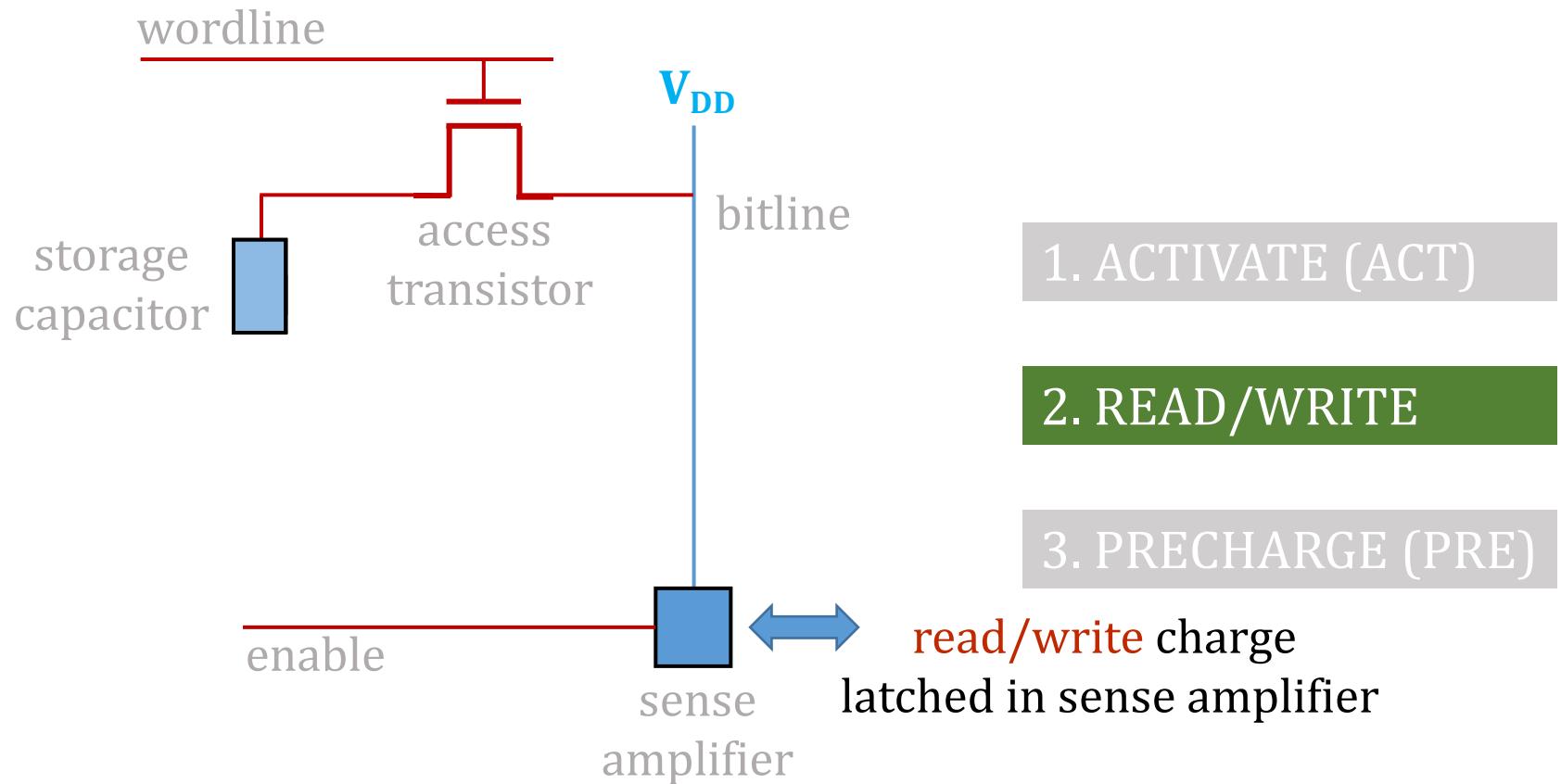
2. READ/WRITE

3. PRECHARGE (PRE)

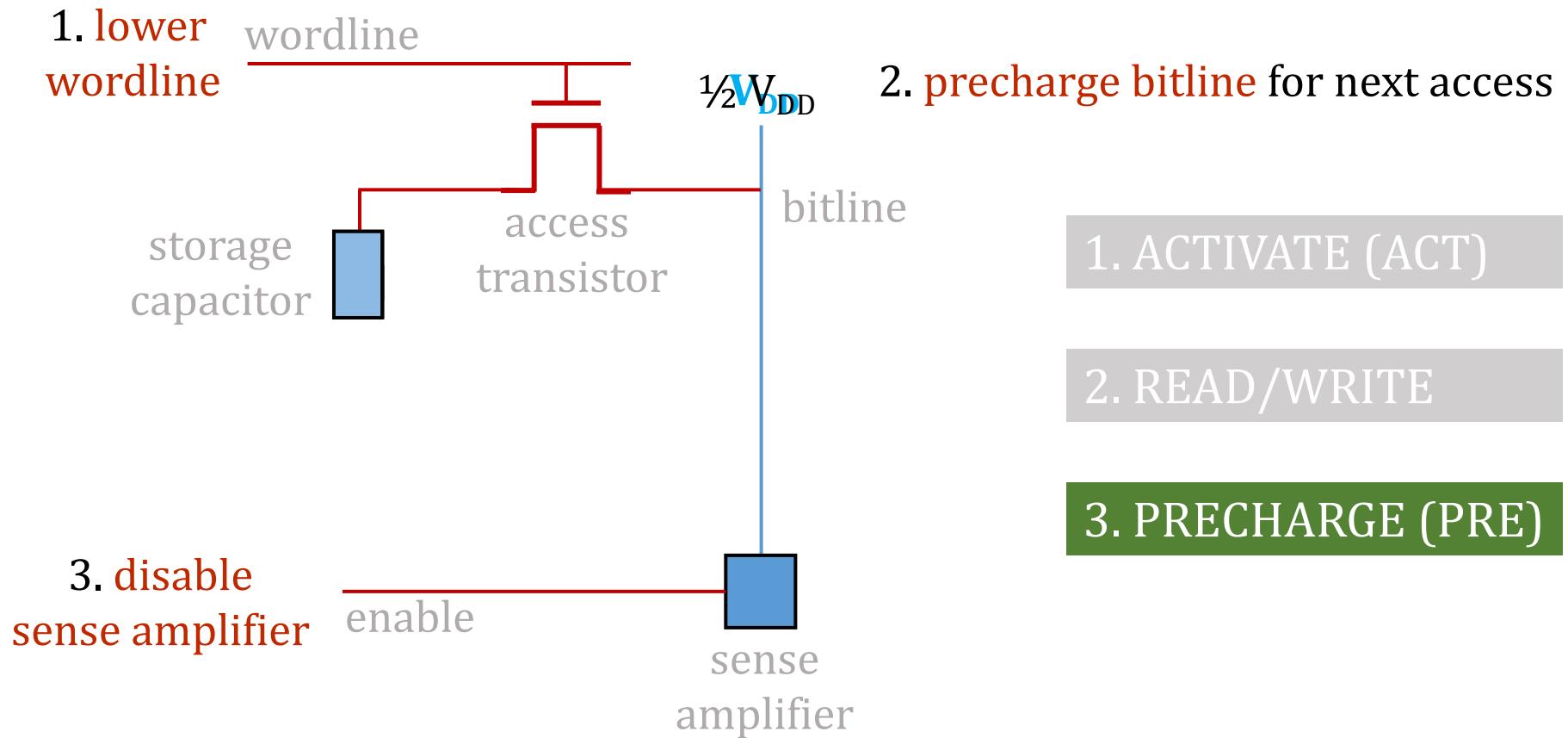
# DRAM Cell Operation (1/3)



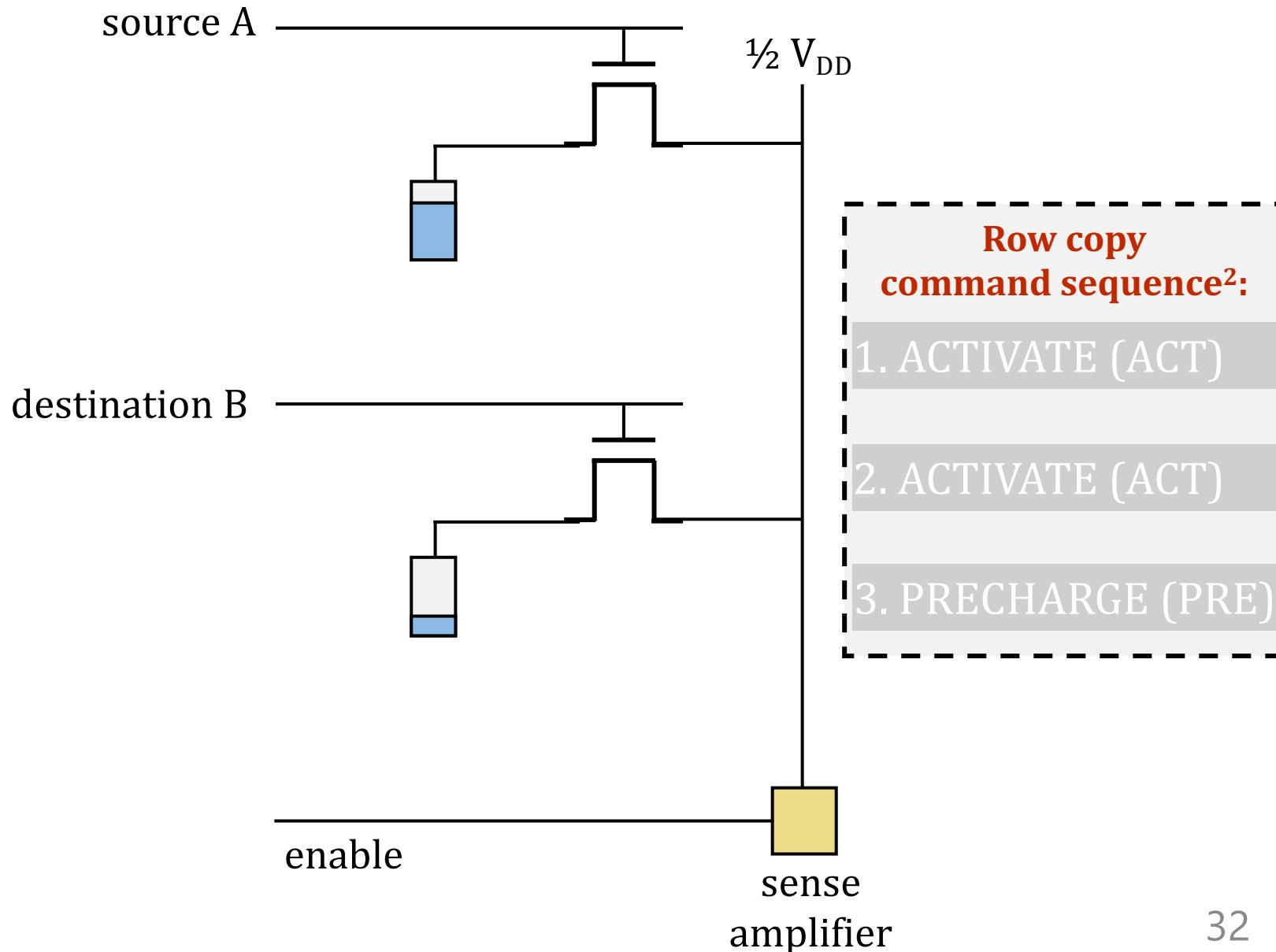
# DRAM Cell Operation (2/3)



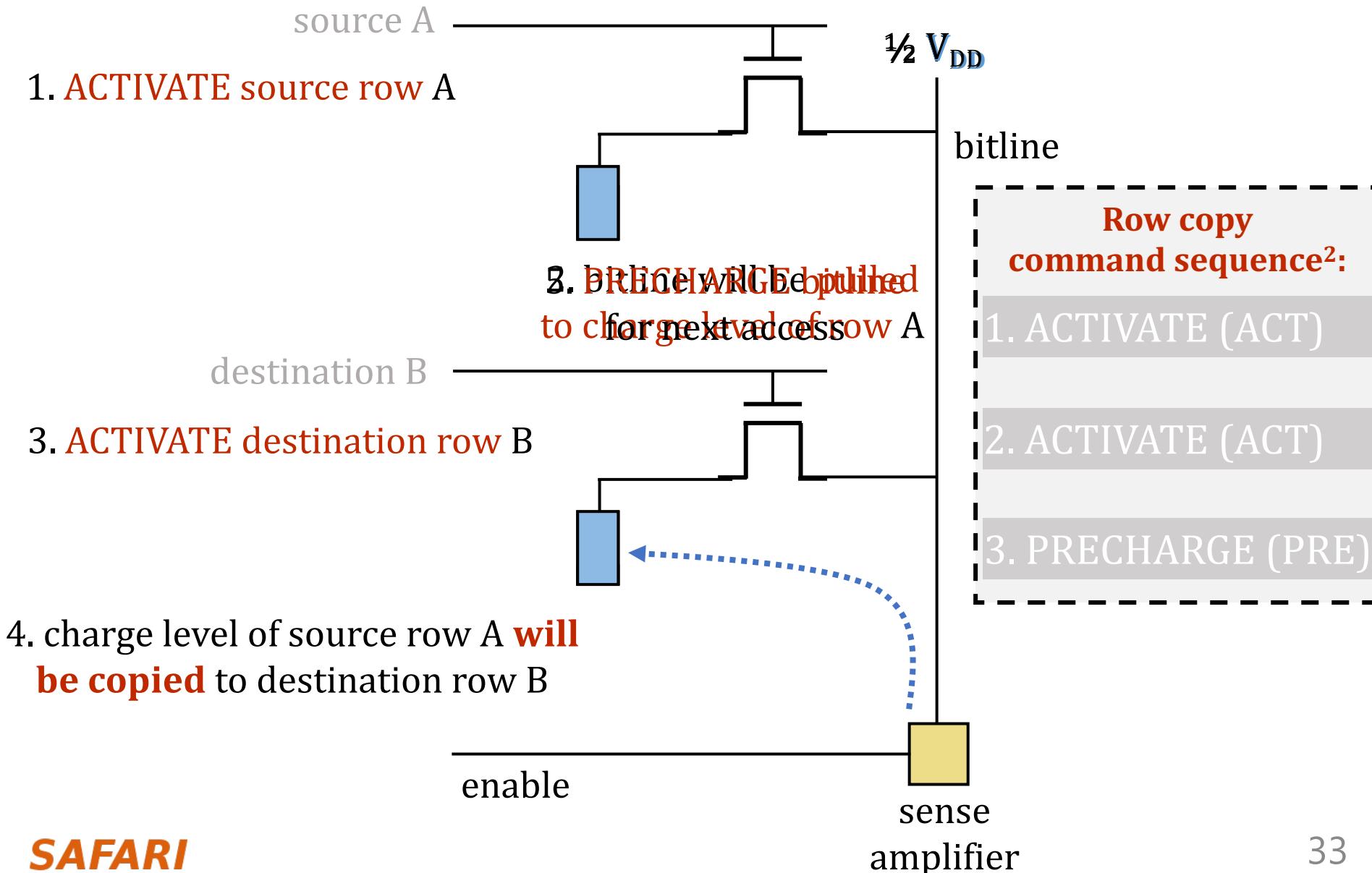
# DRAM Cell Operation (3/3)



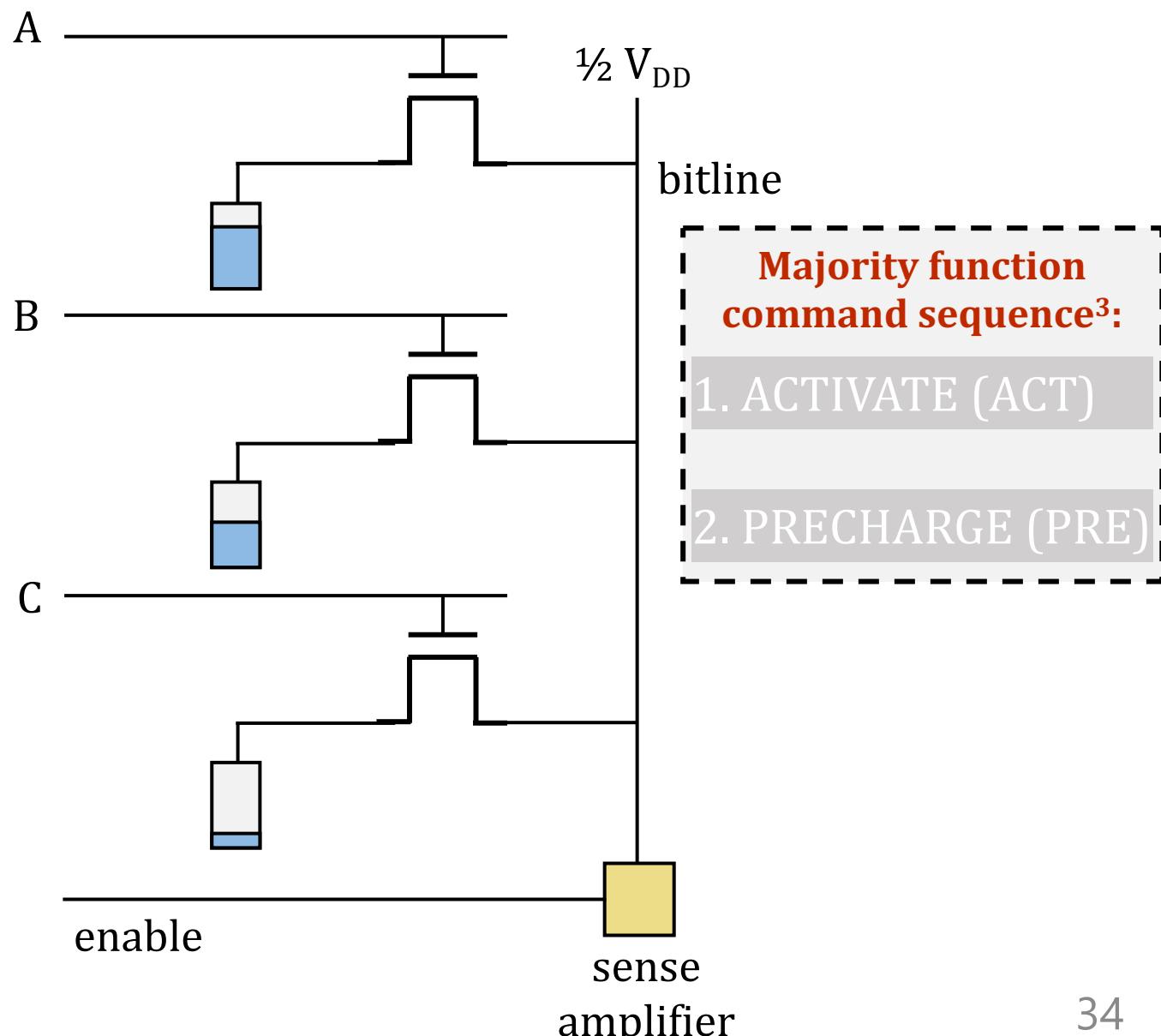
# RowClone: In-DRAM Row Copy (1/2)



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



# Triple-Row Activation: Majority Function



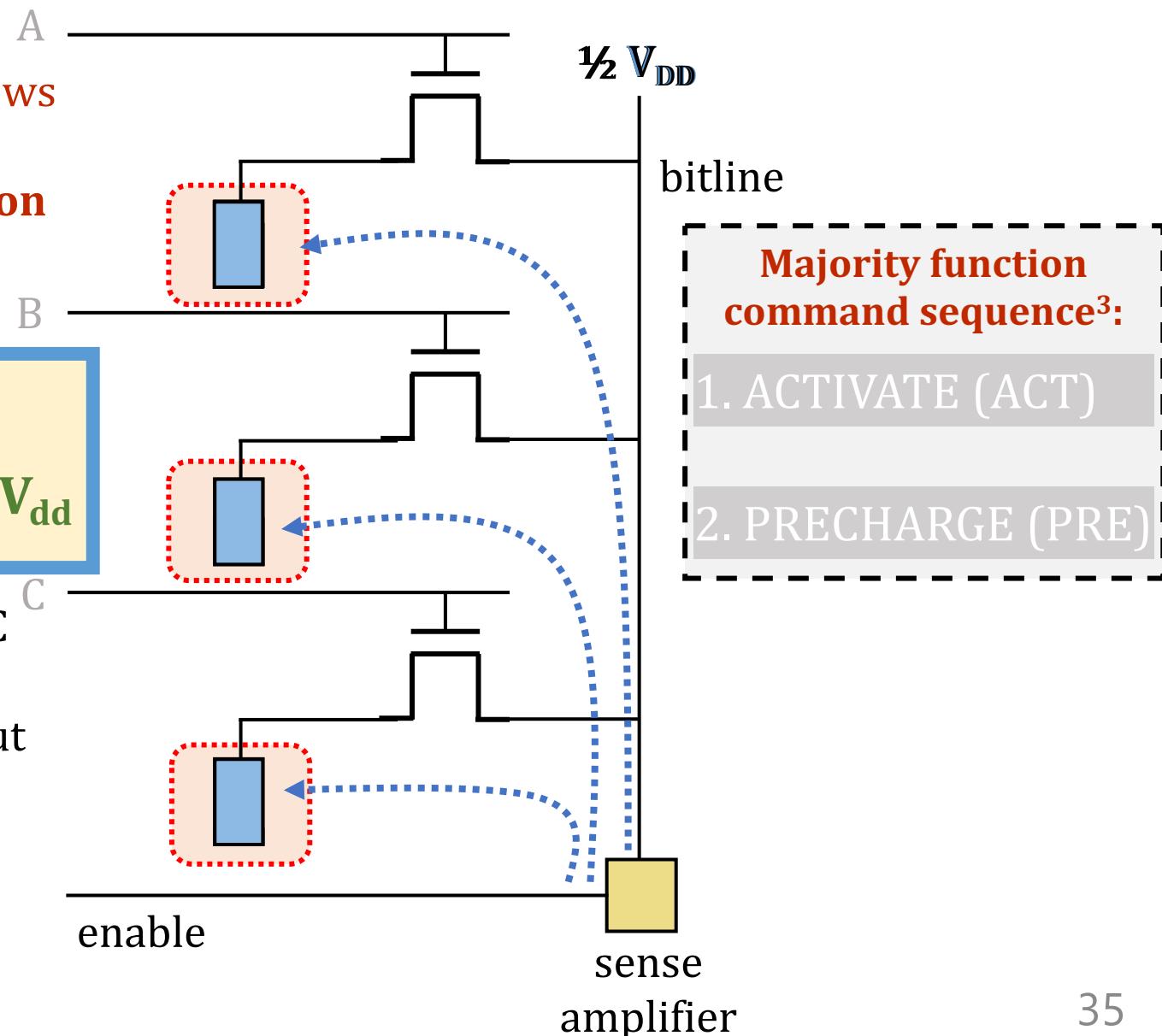
# Triple-Row Activation: Majority Function

1. ACTIVATE three rows simultaneously  
→ triple-row activation

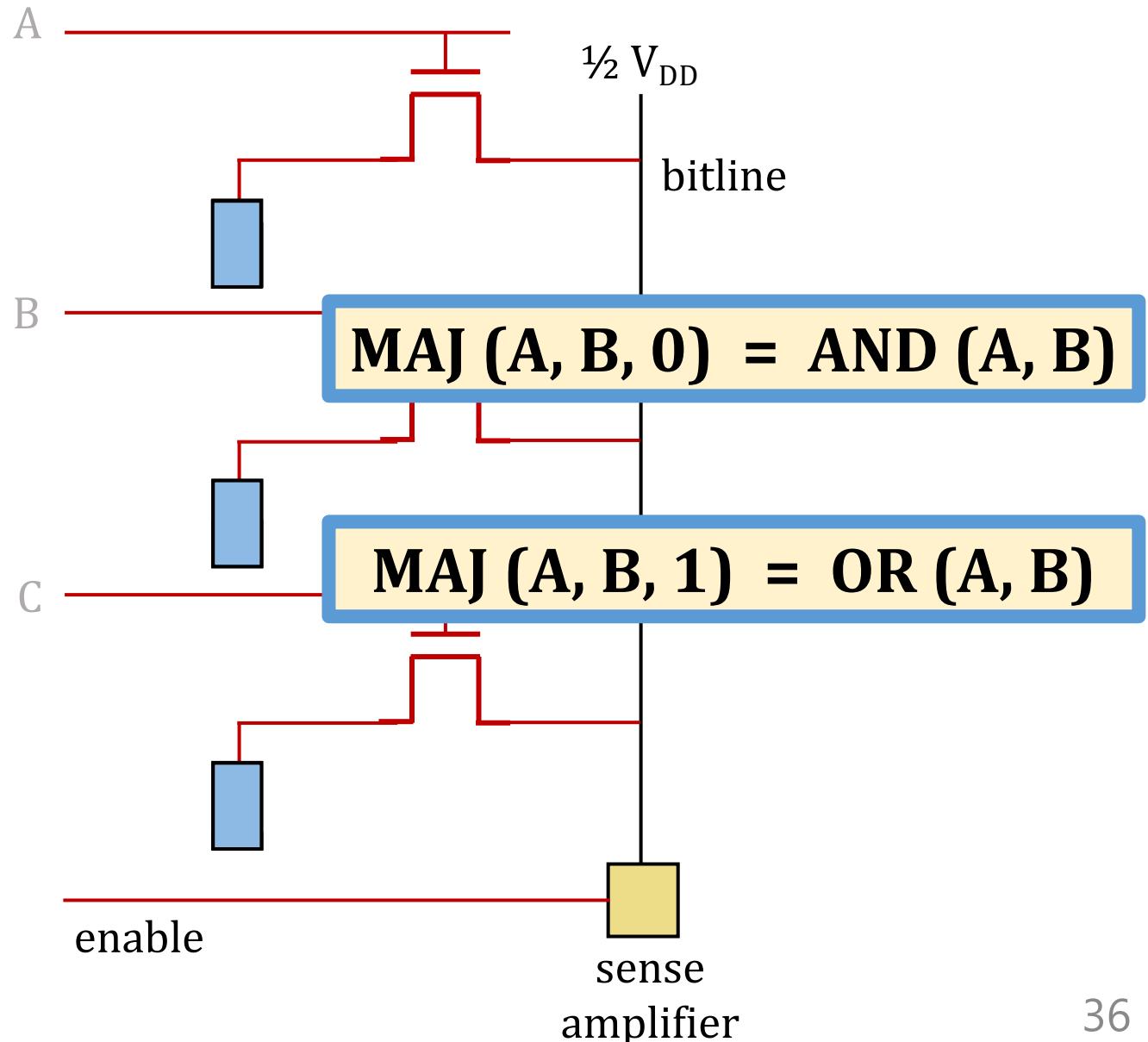
$$\text{MAJ}(A, B, C) =$$
$$\text{MAJ}(V_{dd}, V_{dd}, 0) = V_{dd}$$

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

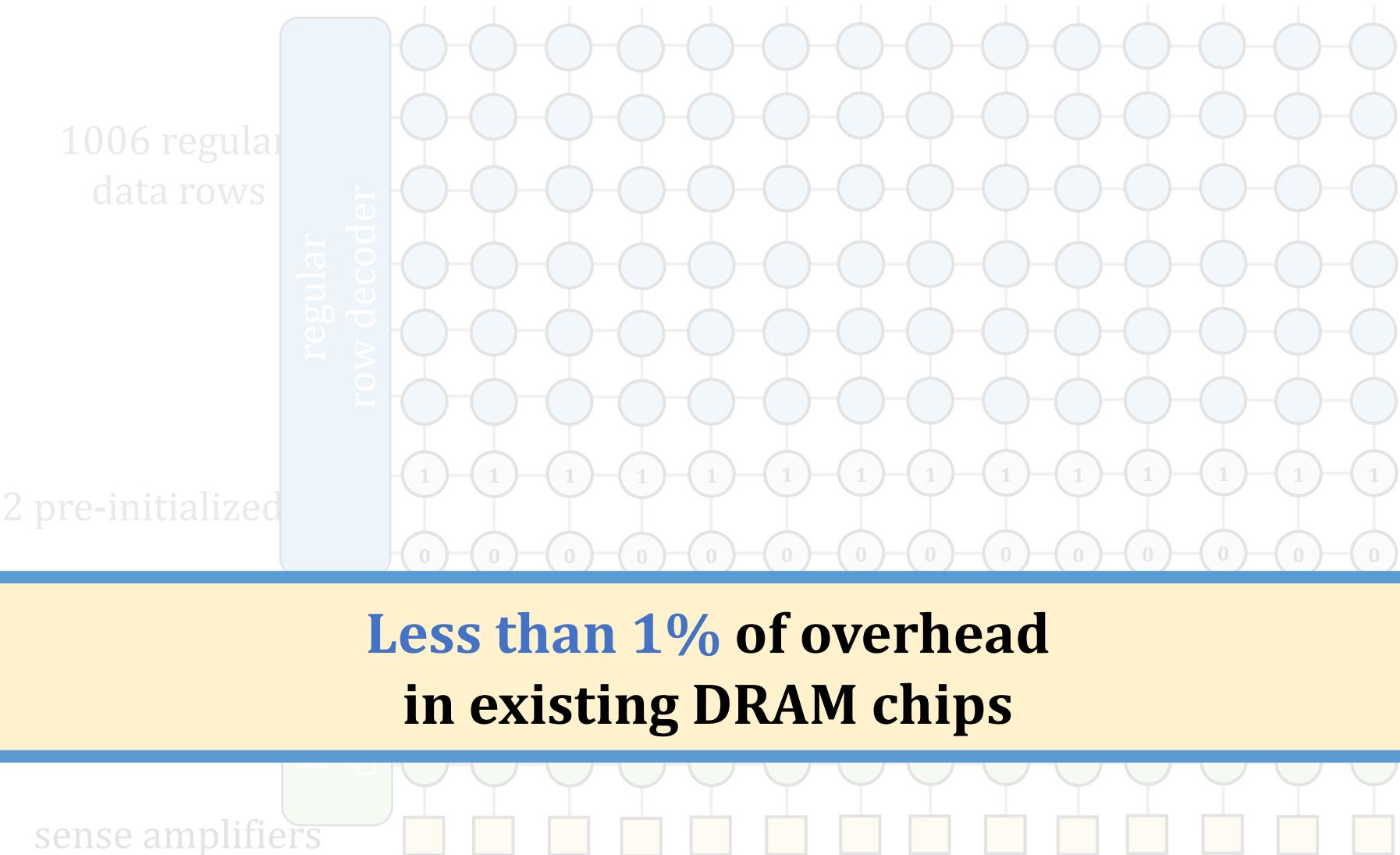
4. PRECHARGE bitline for next access



# 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

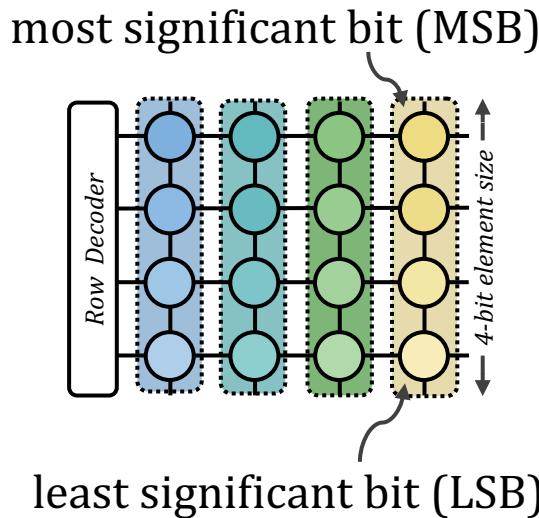
5. Evaluation

6. Conclusion

# SIMDRAM: PuM Substrate

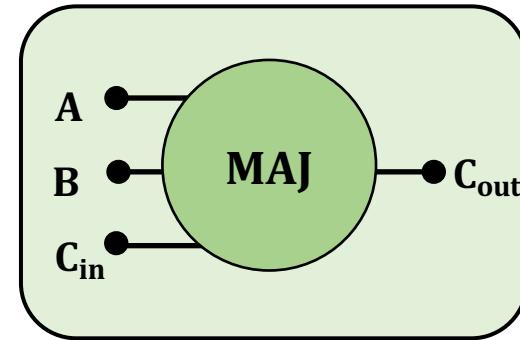
- SIMDRAM framework is built around a DRAM substrate that enables two techniques:

## (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

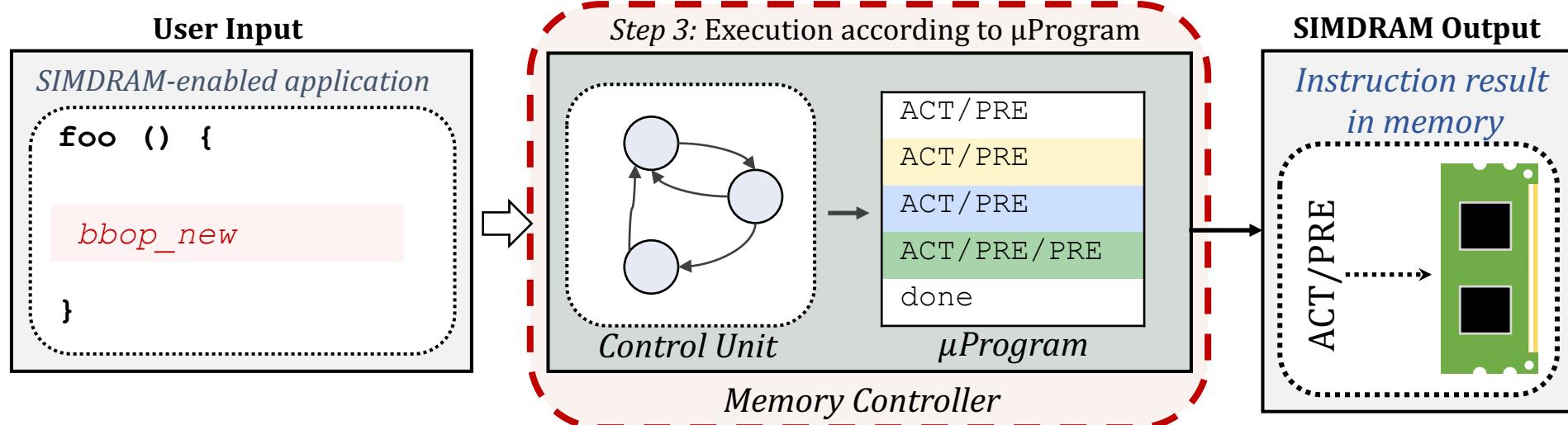
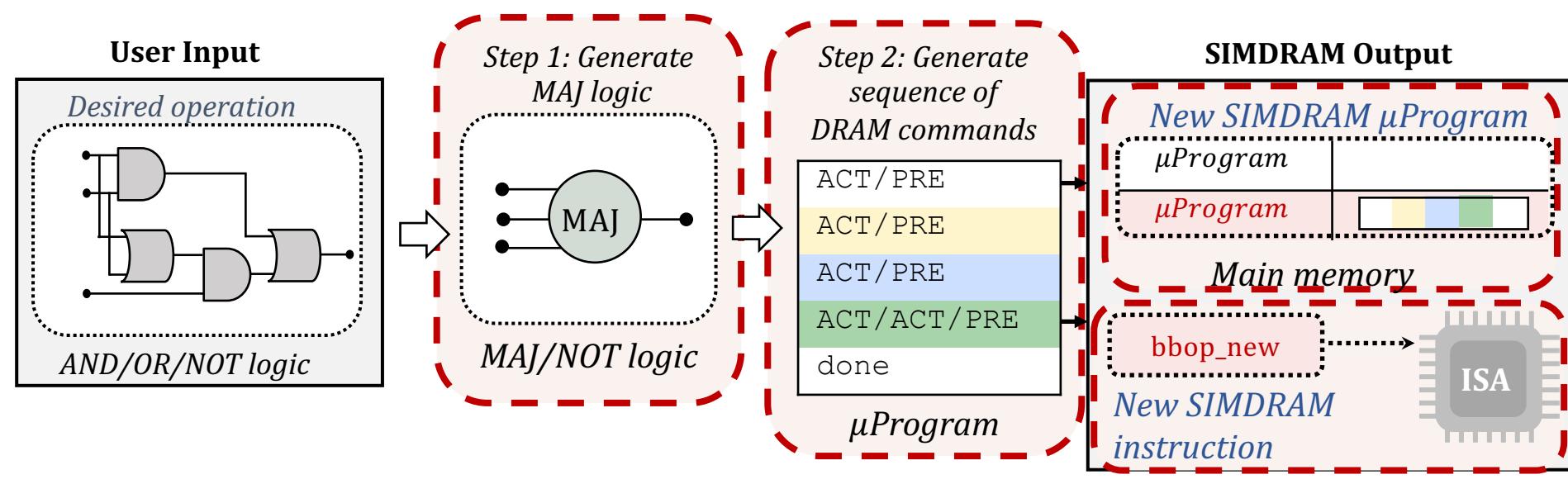
Processing-using-DRAM Substrate  
Framework

4. System Integration

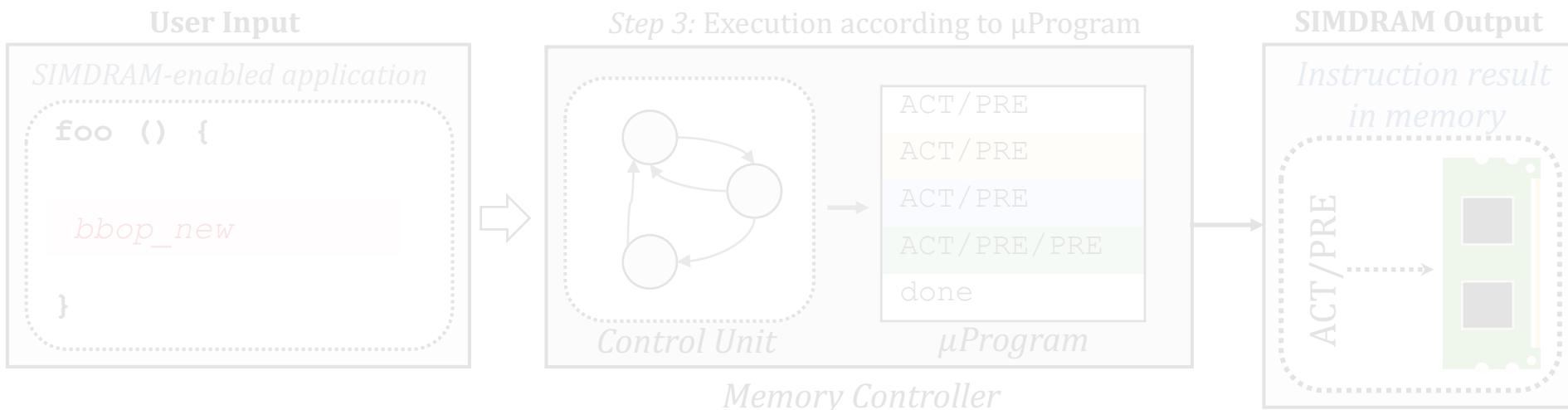
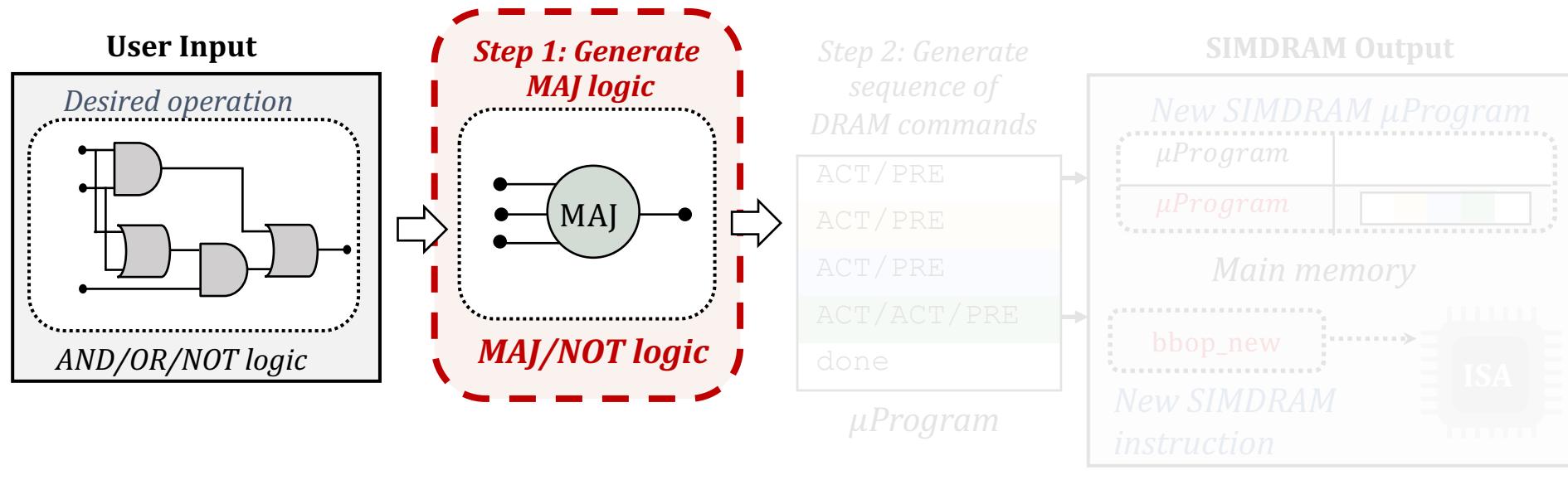
5. Evaluation

6. Conclusion

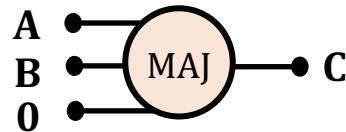
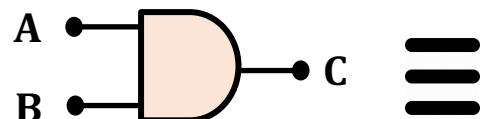
# SIMDRAM Framework



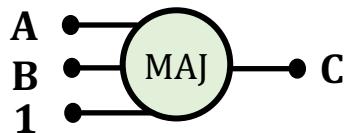
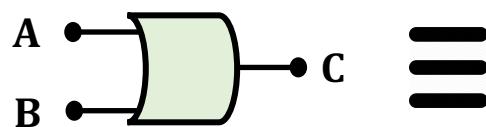
# SIMDRAM Framework: Step 1



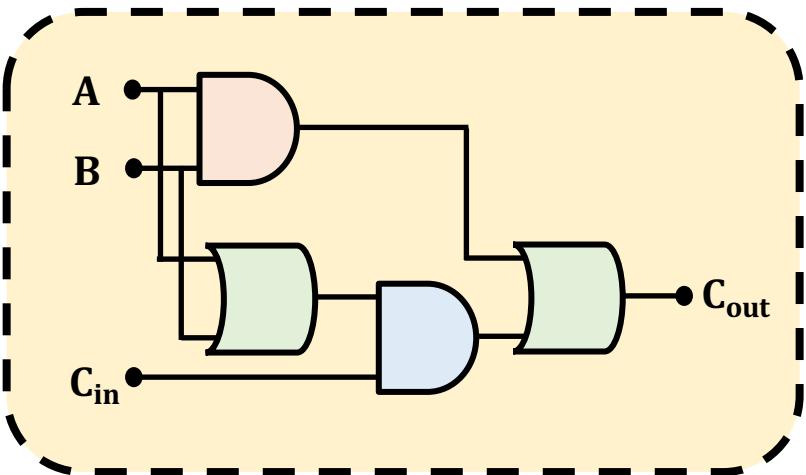
# Step 1: Naïve MAJ/NOT Implementation



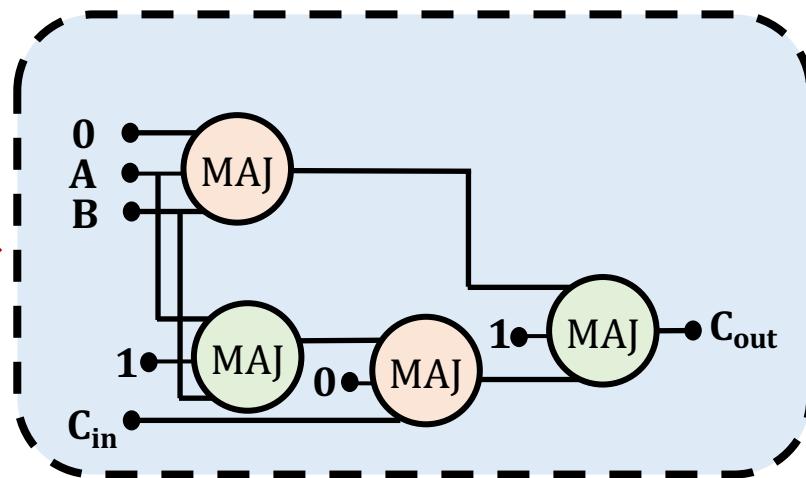
output is “1” only when  $A = B = “1”$



output is “0” only when  $A = B = “0”$

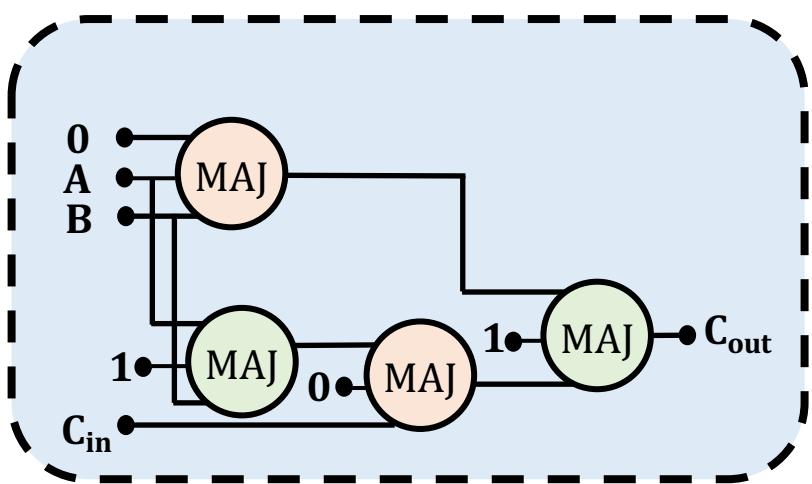


Part 1



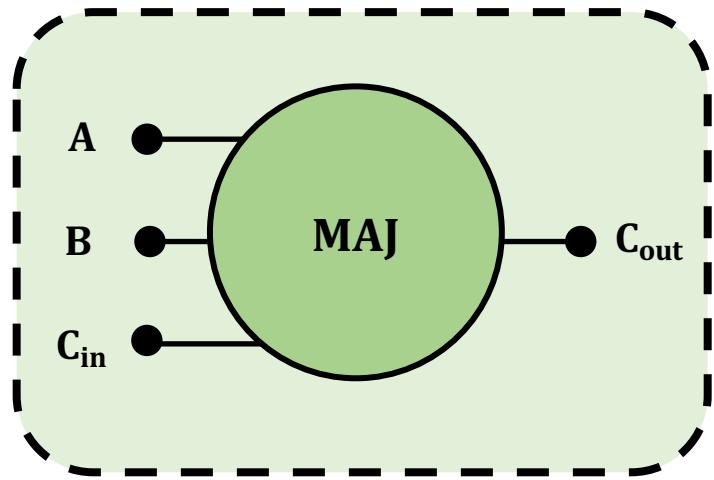
**Naïvely converting AND/OR/NOT-implementation to MAJ/NOT-implementation leads to an unoptimized circuit**

# Step 1: Efficient MAJ/NOT Implementation



Greedy  
optimization  
algorithm<sup>4</sup>

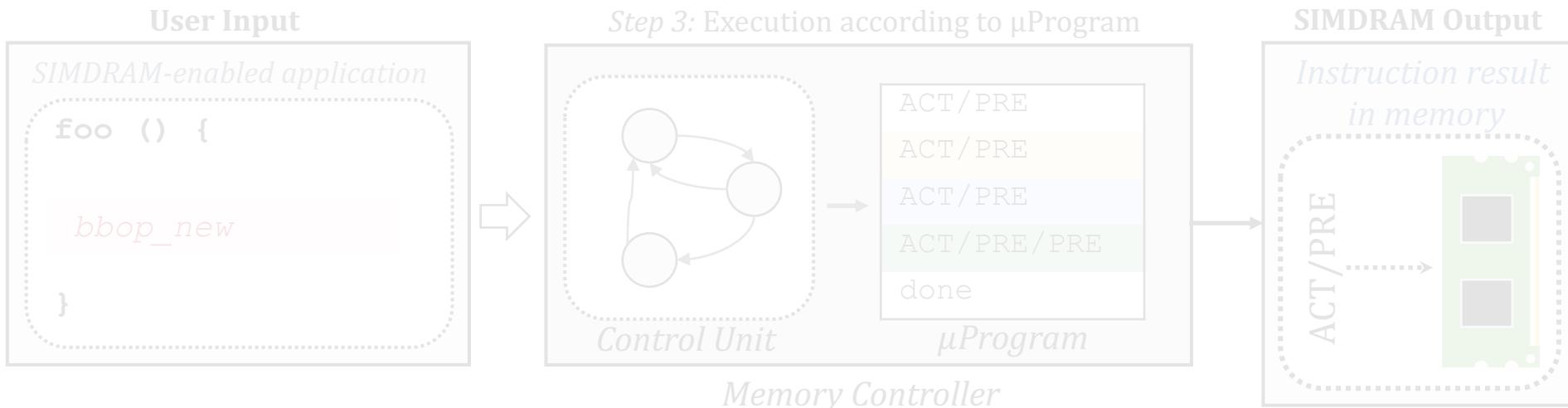
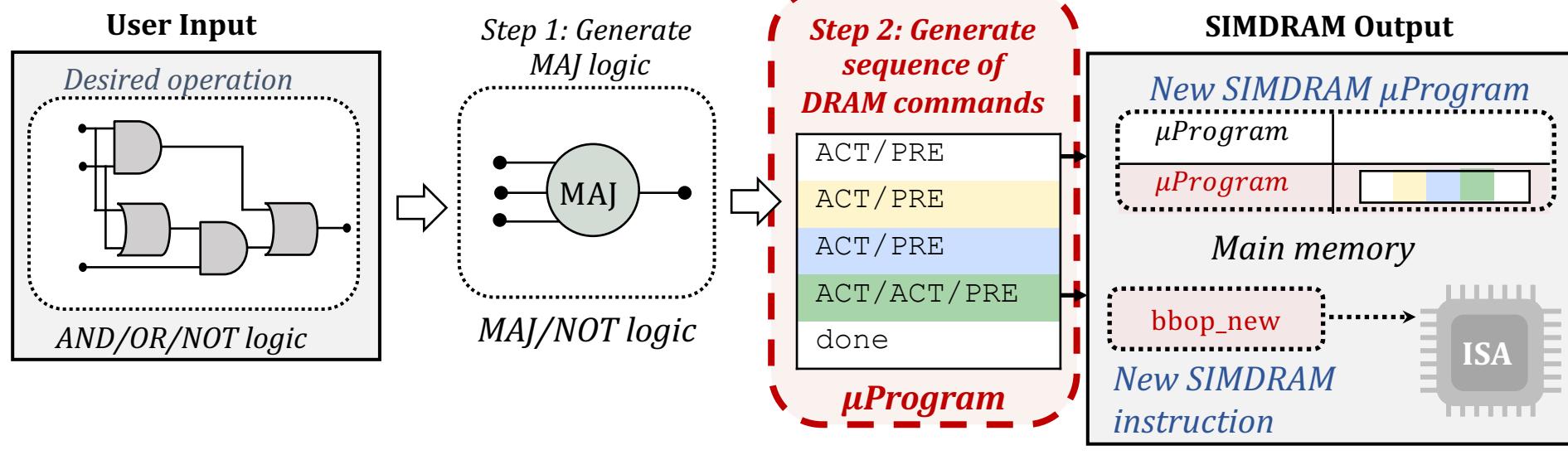
Part 2



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

<sup>4</sup> 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

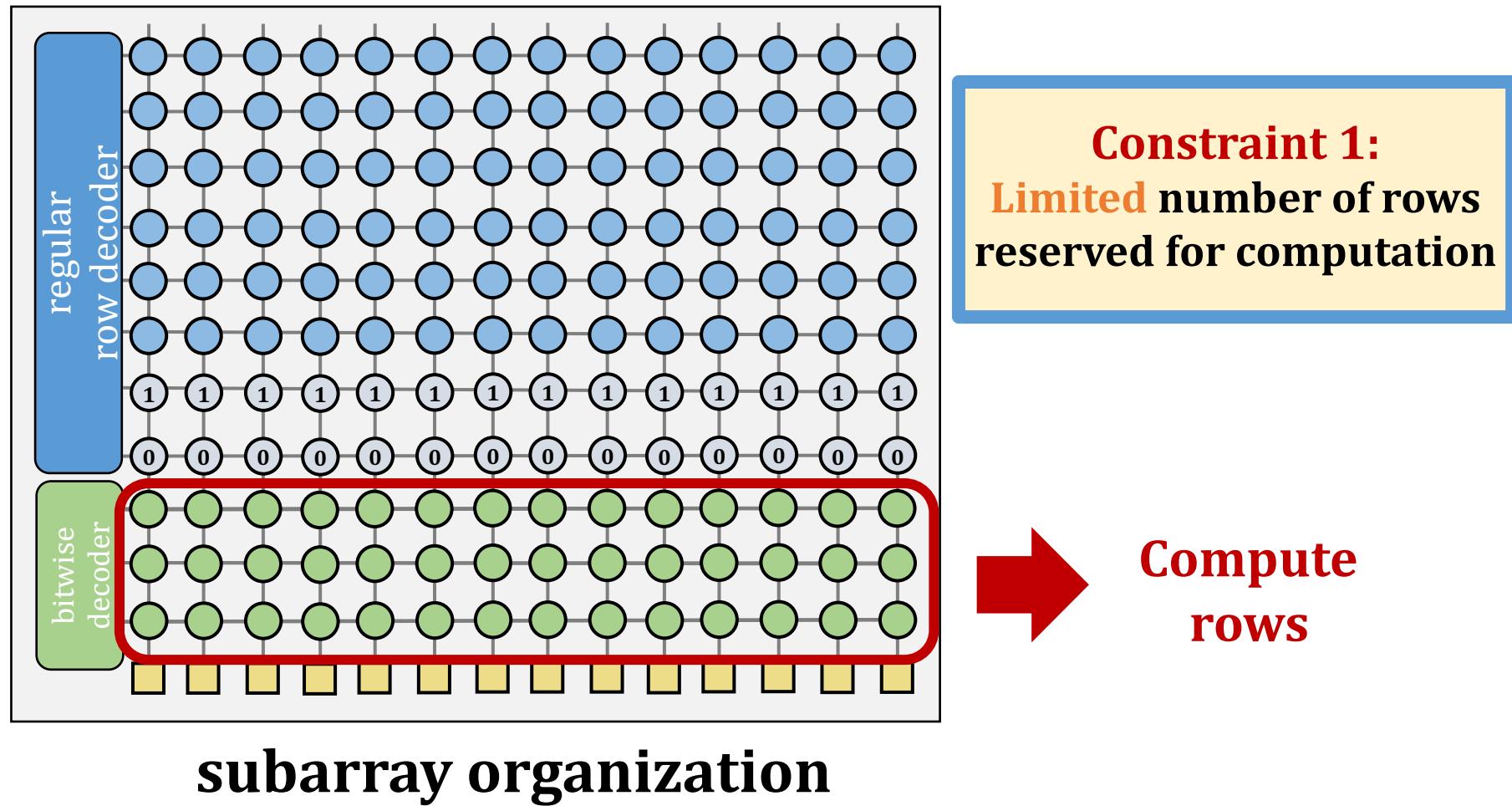
- **μProgram:** A series of microarchitectural operations (e.g., ACT/PRE) that SIMDRAM uses to execute SIMDRAM operation in DRAM
- **Goal of Step 2:** To generate the μProgram that executes the desired SIMDRAM operation in DRAM

Task 1: Allocate DRAM rows to the operands

Task 2: Generate μProgram

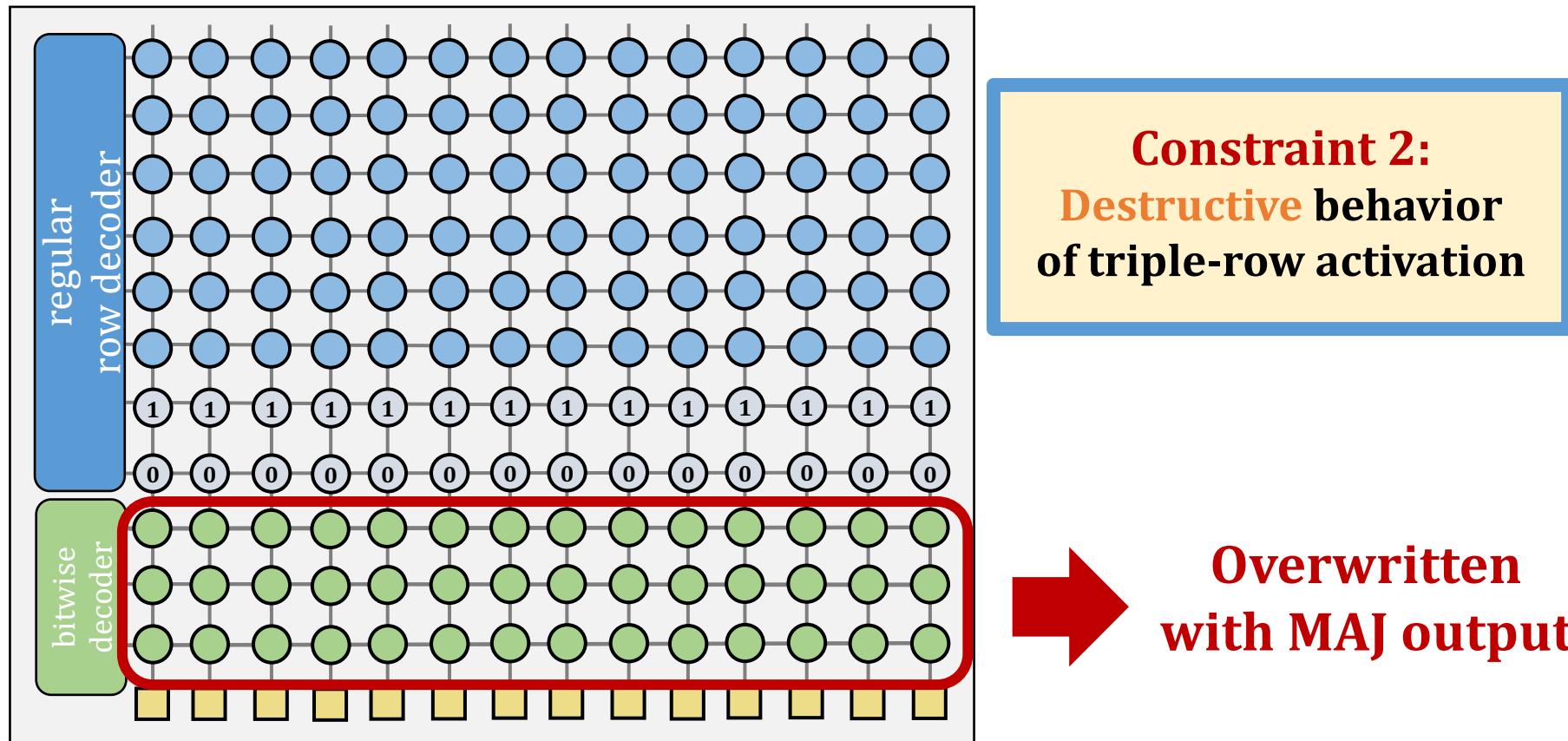
# Task 1: Allocating DRAM Rows to Operands

- 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

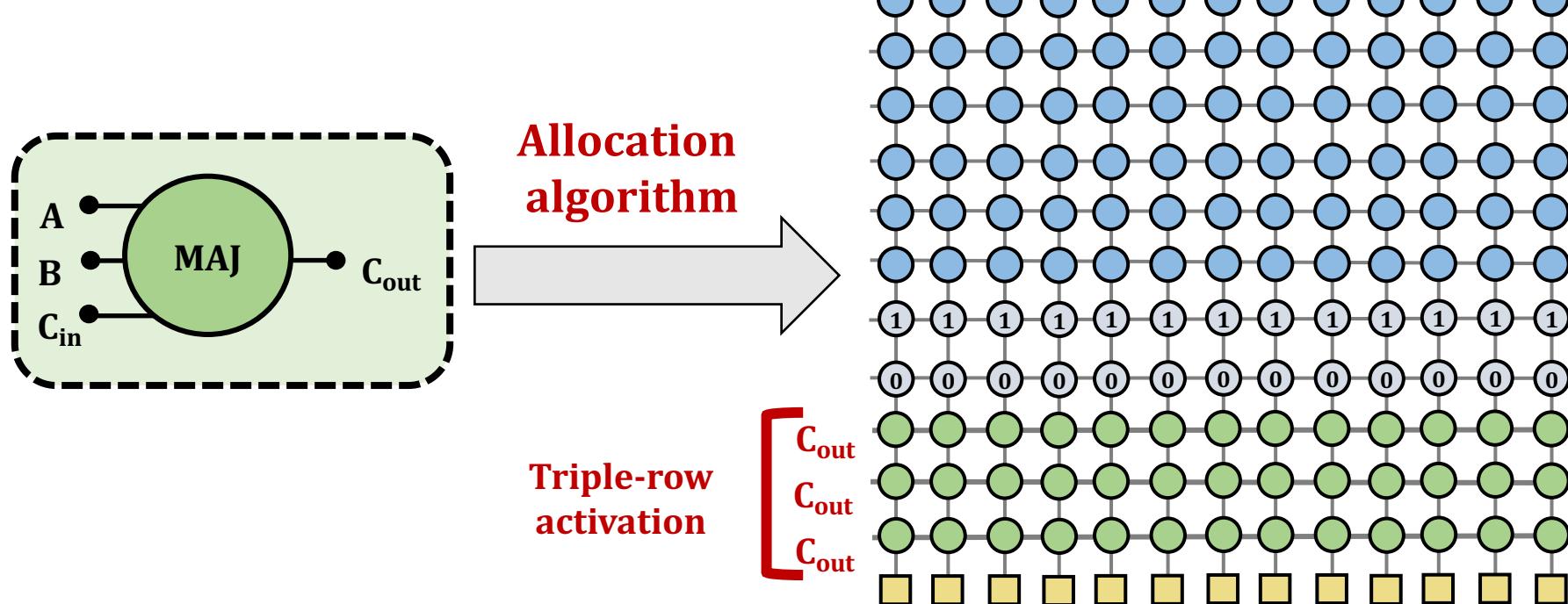


subarray organization

# Task 1: Allocating DRAM Rows to Operands

- Allocation algorithm:

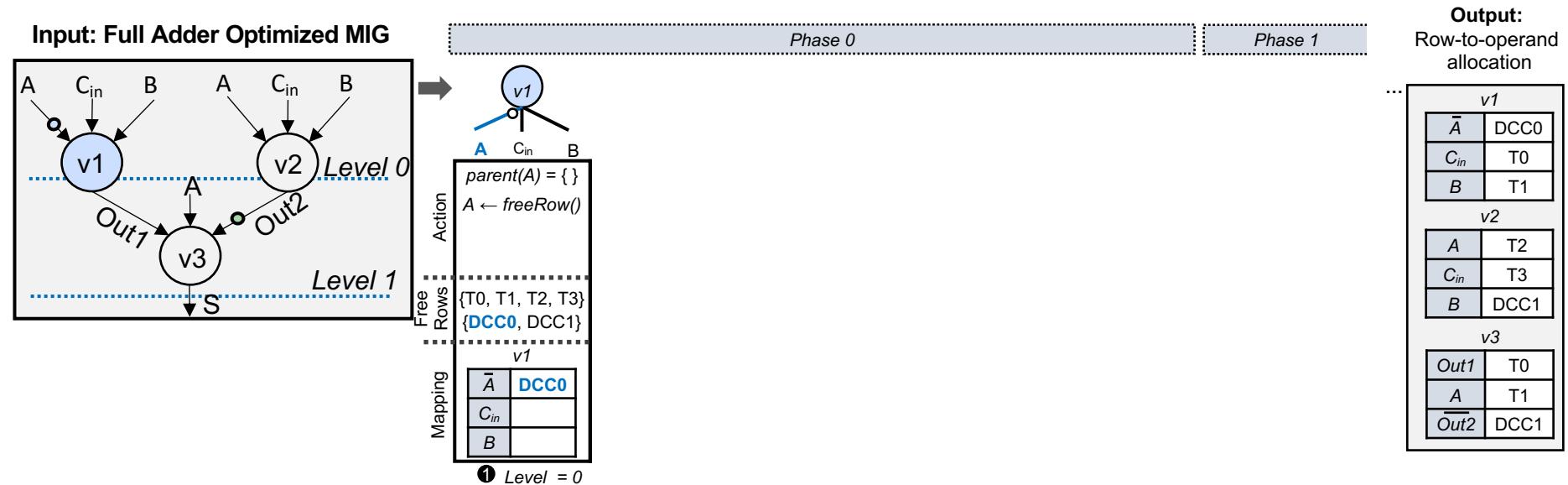
- Assigns as many inputs as the number of free compute rows
- All three input rows contain the MAJ output and can be reused



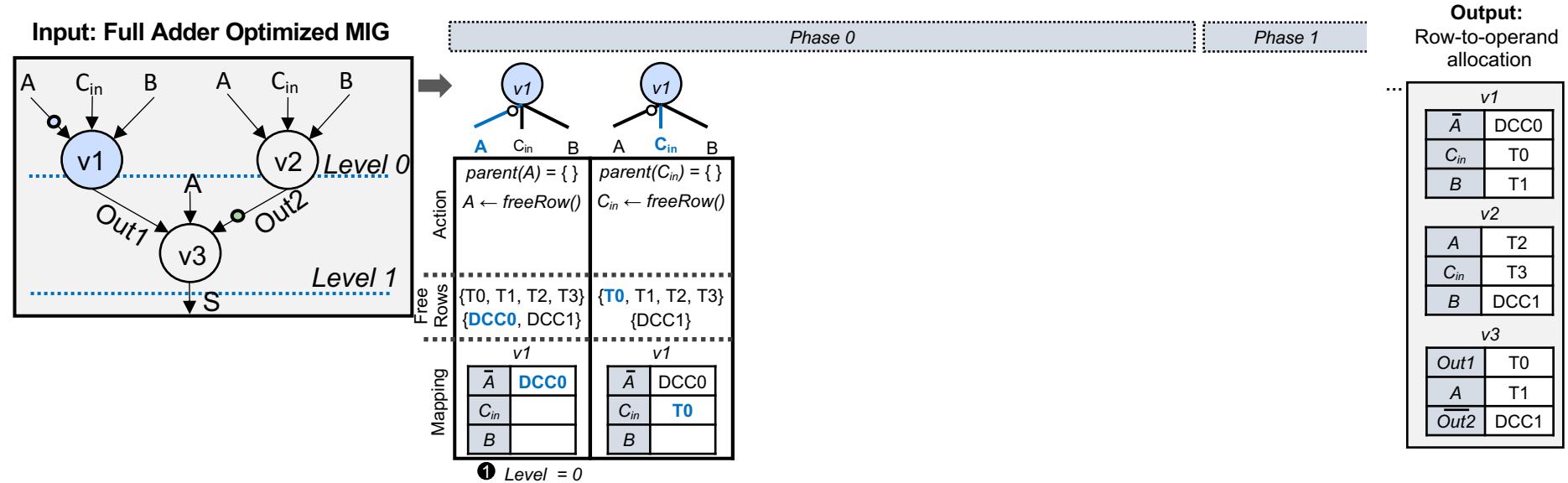
# Task 1: Allocating DRAM Rows to Operands



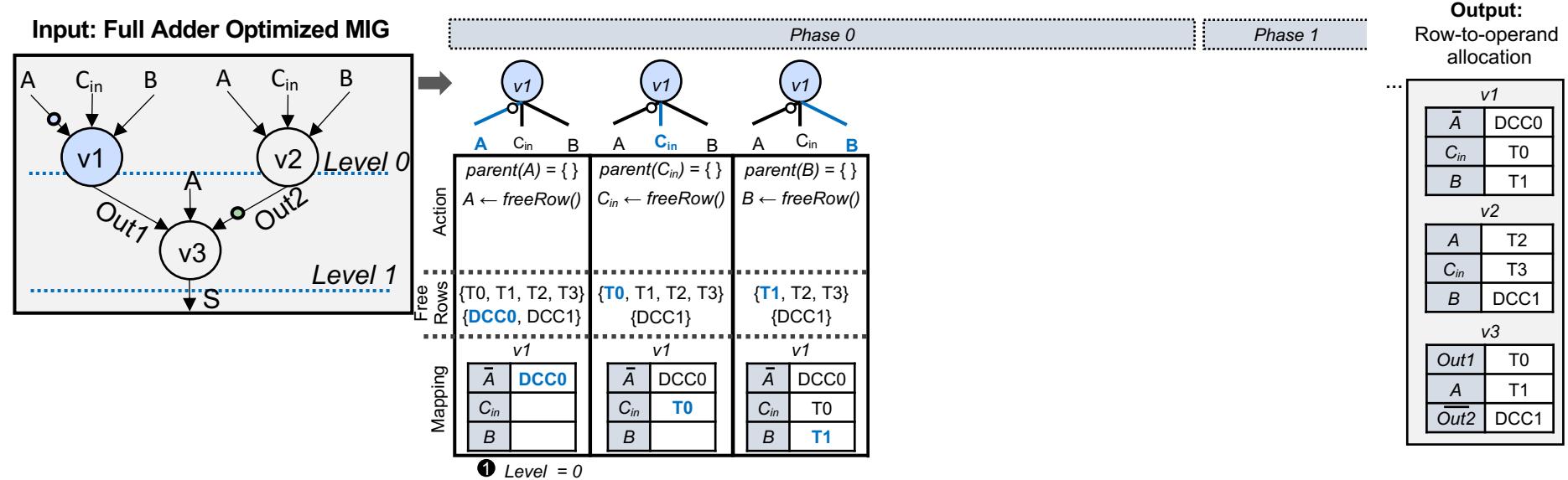
# Task 1: Allocating DRAM Rows to Operands



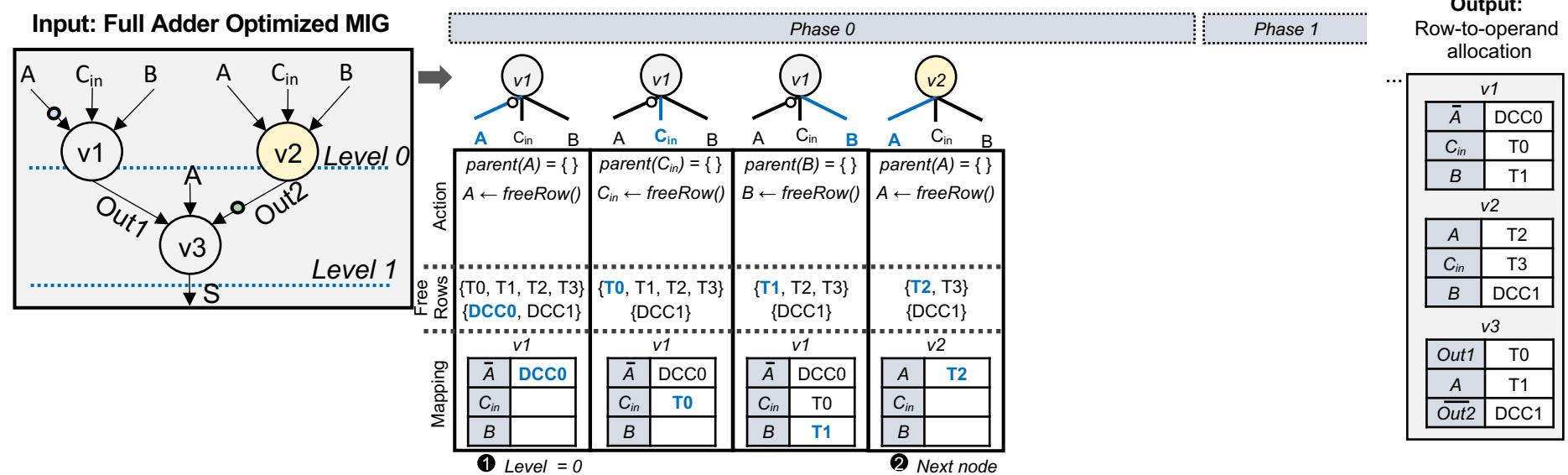
# Task 1: Allocating DRAM Rows to Operands



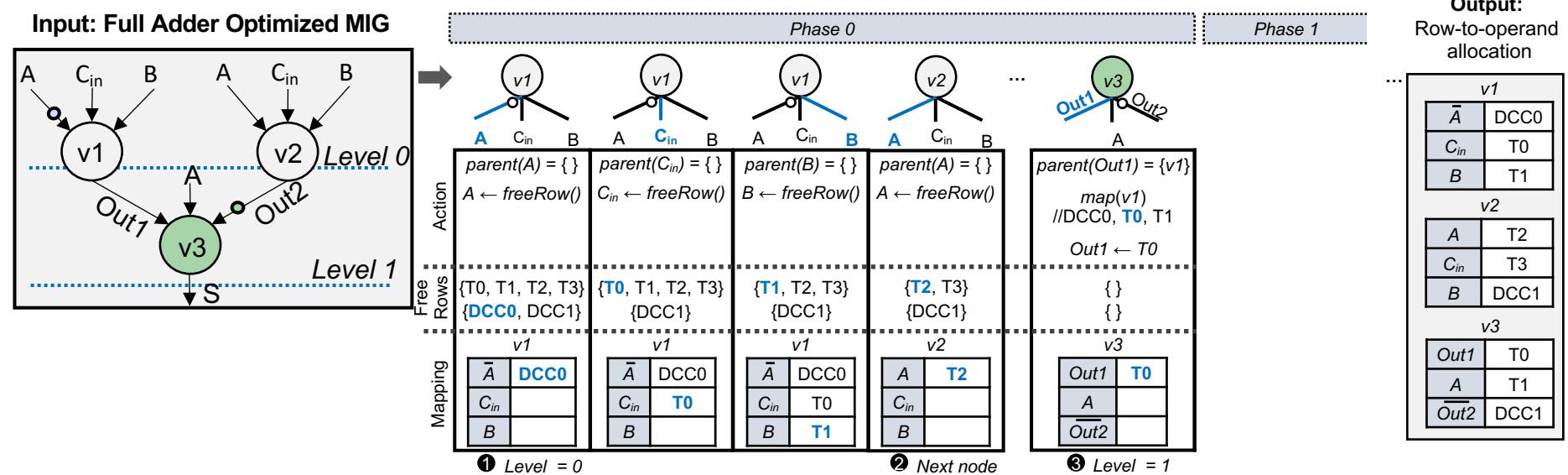
# Task 1: Allocating DRAM Rows to Operands



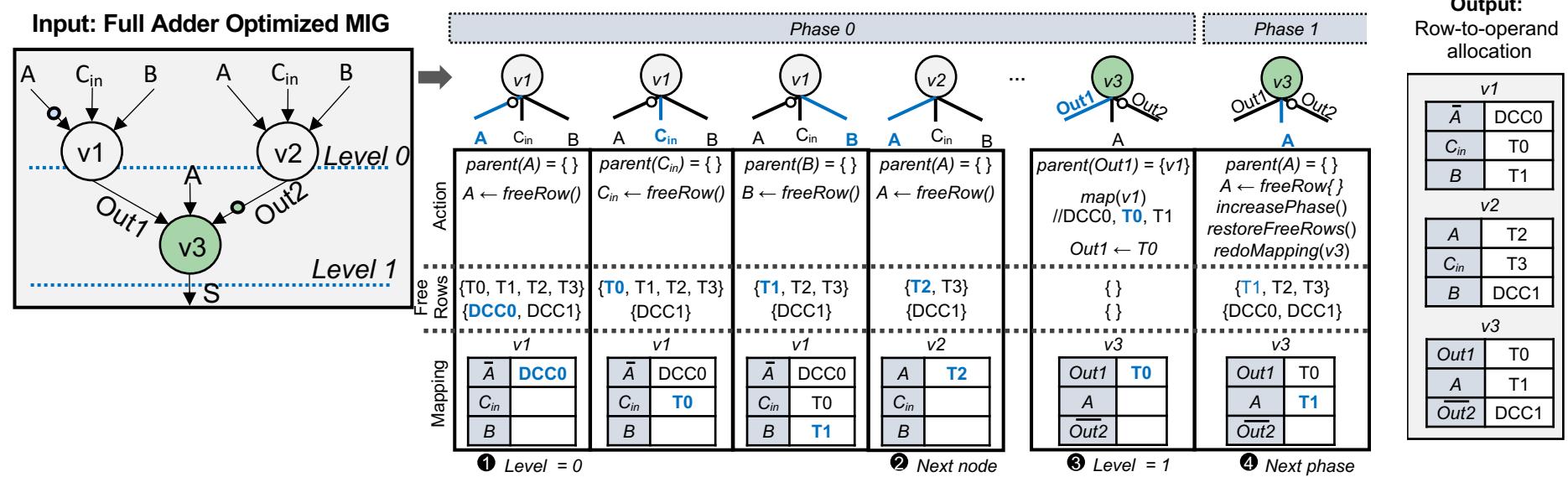
# Task 1: Allocating DRAM Rows to Operands



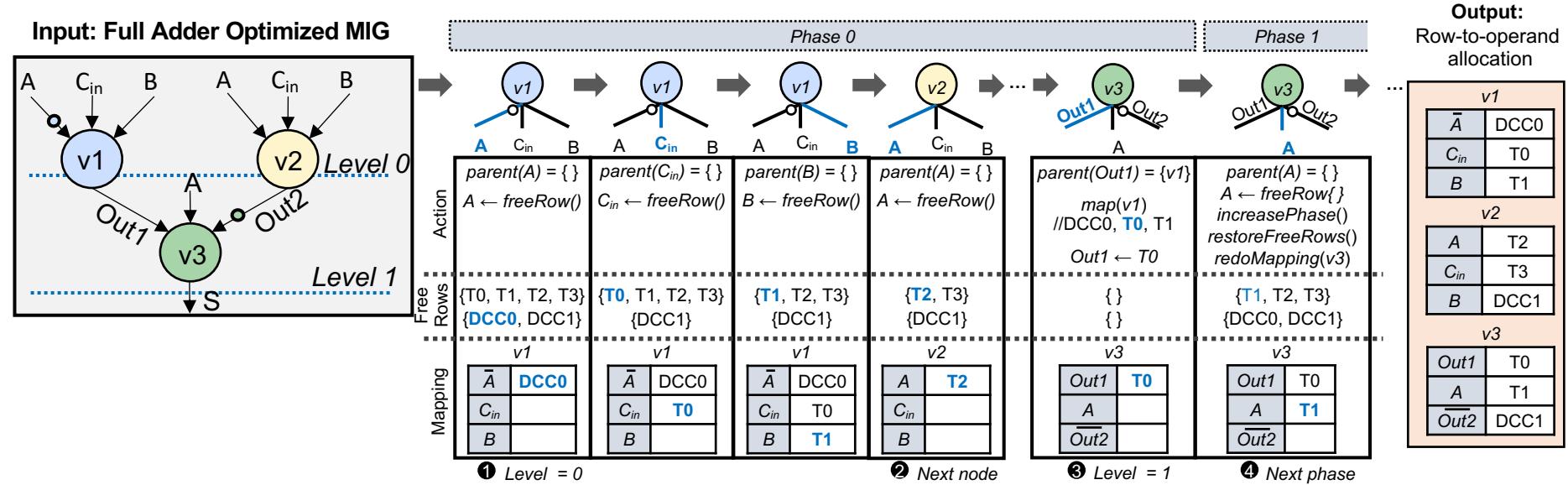
# Task 1: Allocating DRAM Rows to Operands



# Task 1: Allocating DRAM Rows to Operands



# Task 1: Allocating DRAM Rows to Operands



# Step 2: μProgram Generation

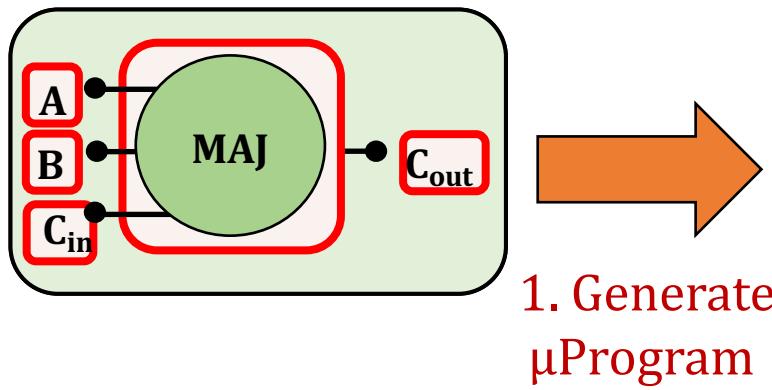
- **μProgram:** A series of microarchitectural operations (e.g., ACT/PRE) that SIMDRAM uses to execute SIMDRAM operation in DRAM
- **Goal of Step 2:** To generate the μProgram that executes the desired SIMDRAM operation in DRAM

Task 1: Allocate DRAM rows to the operands

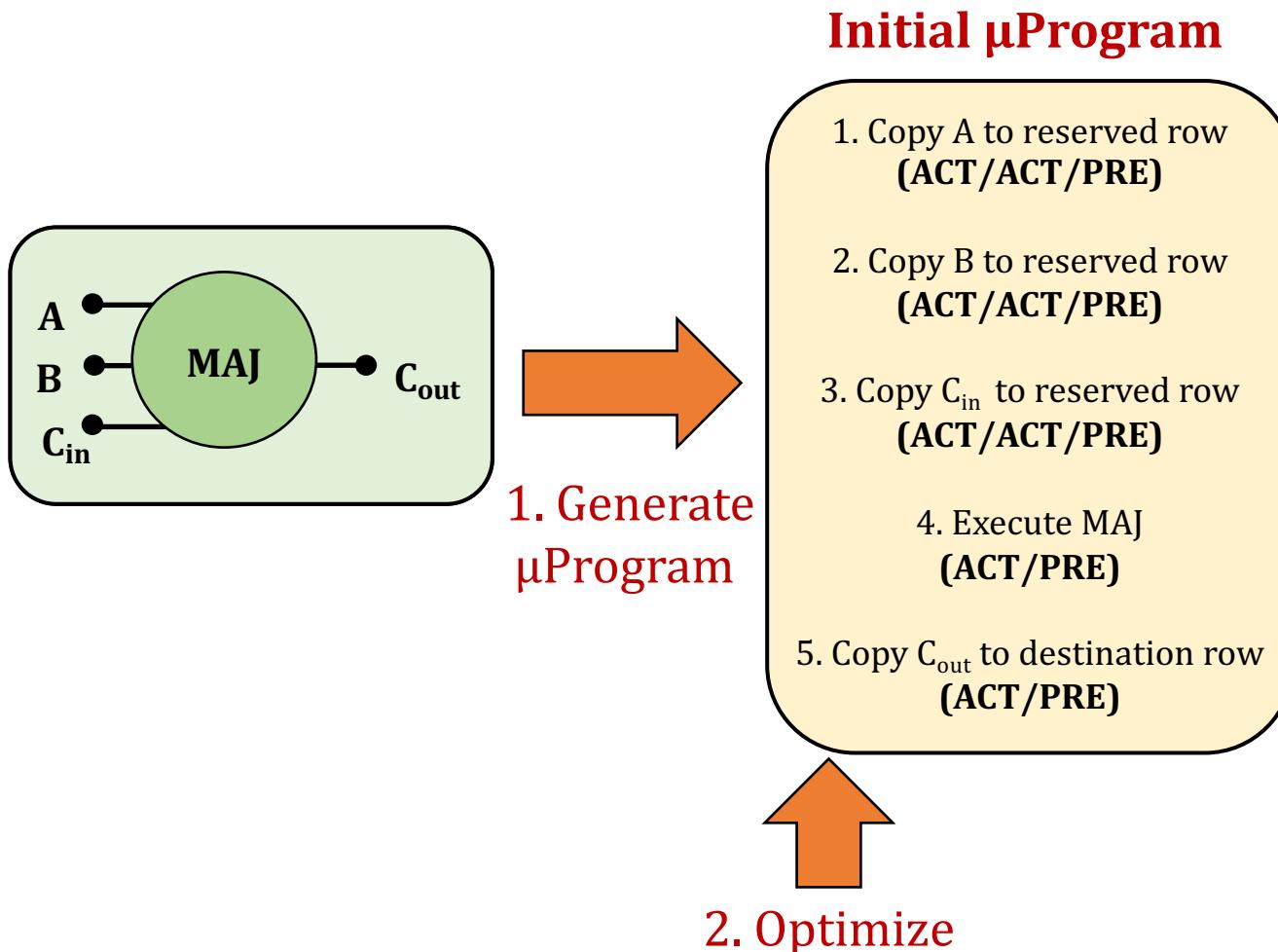
Task 2: Generate μProgram

# Task 2: Generate an initial $\mu$ Program

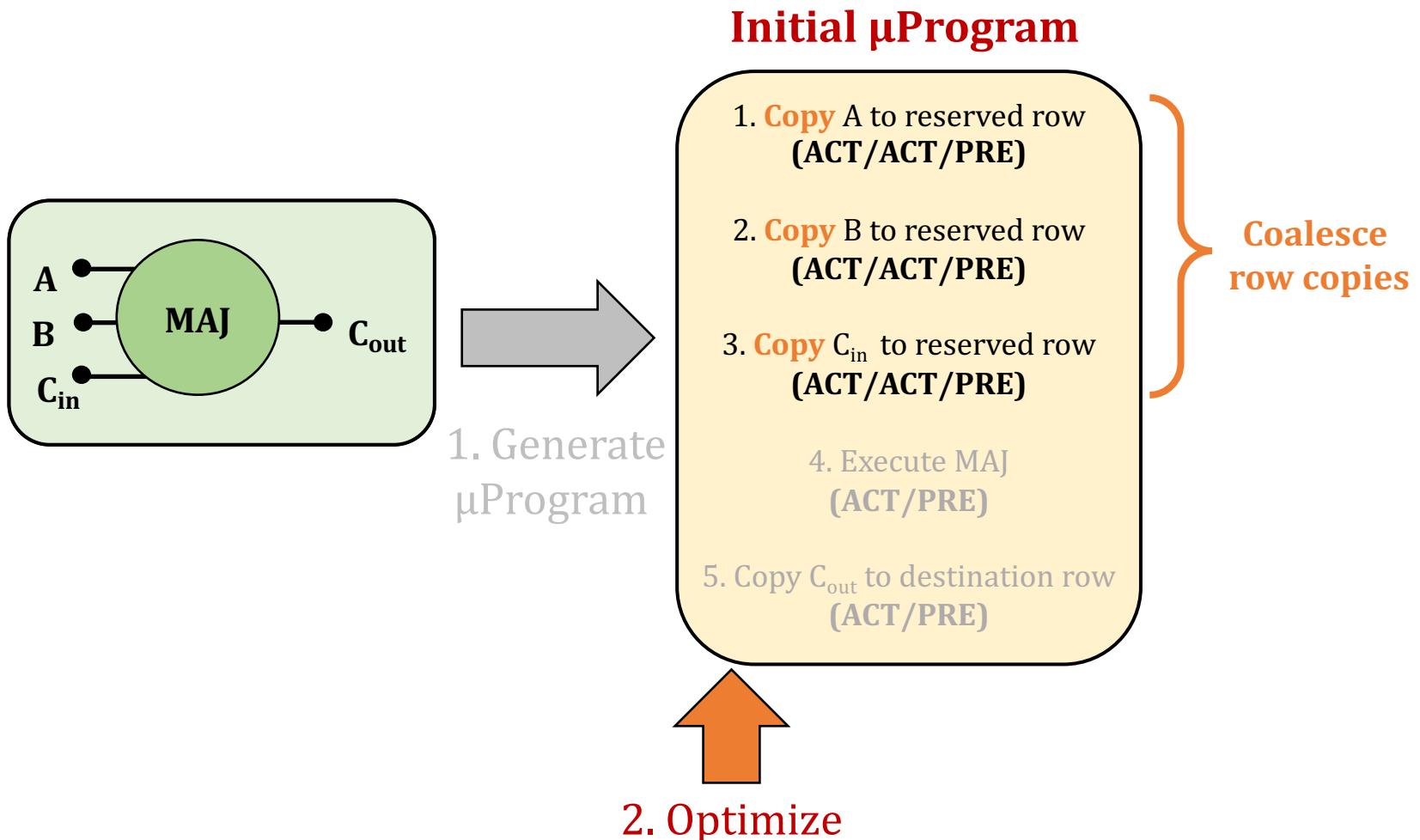
—



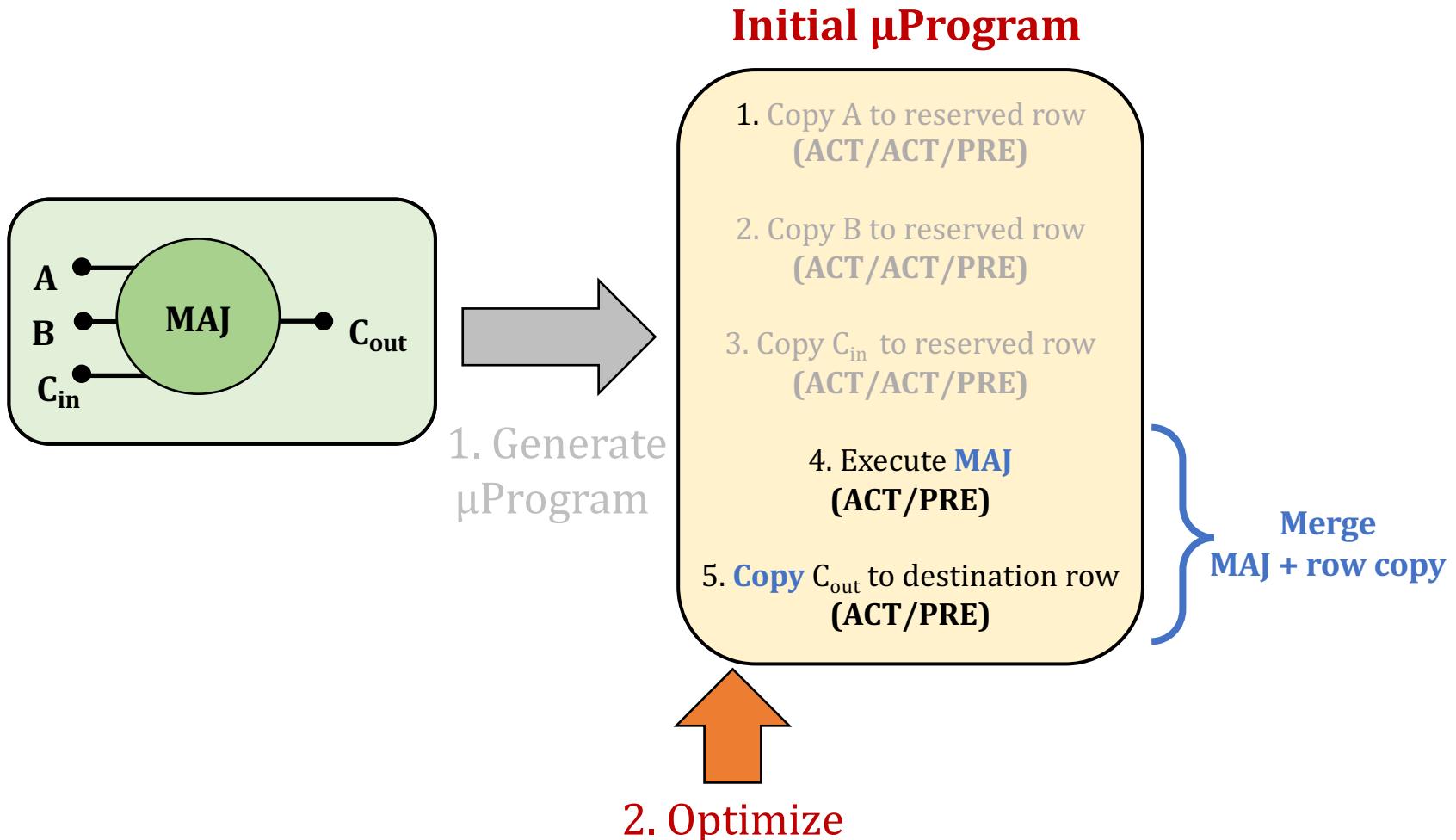
# Task 2: Optimize the $\mu$ Program



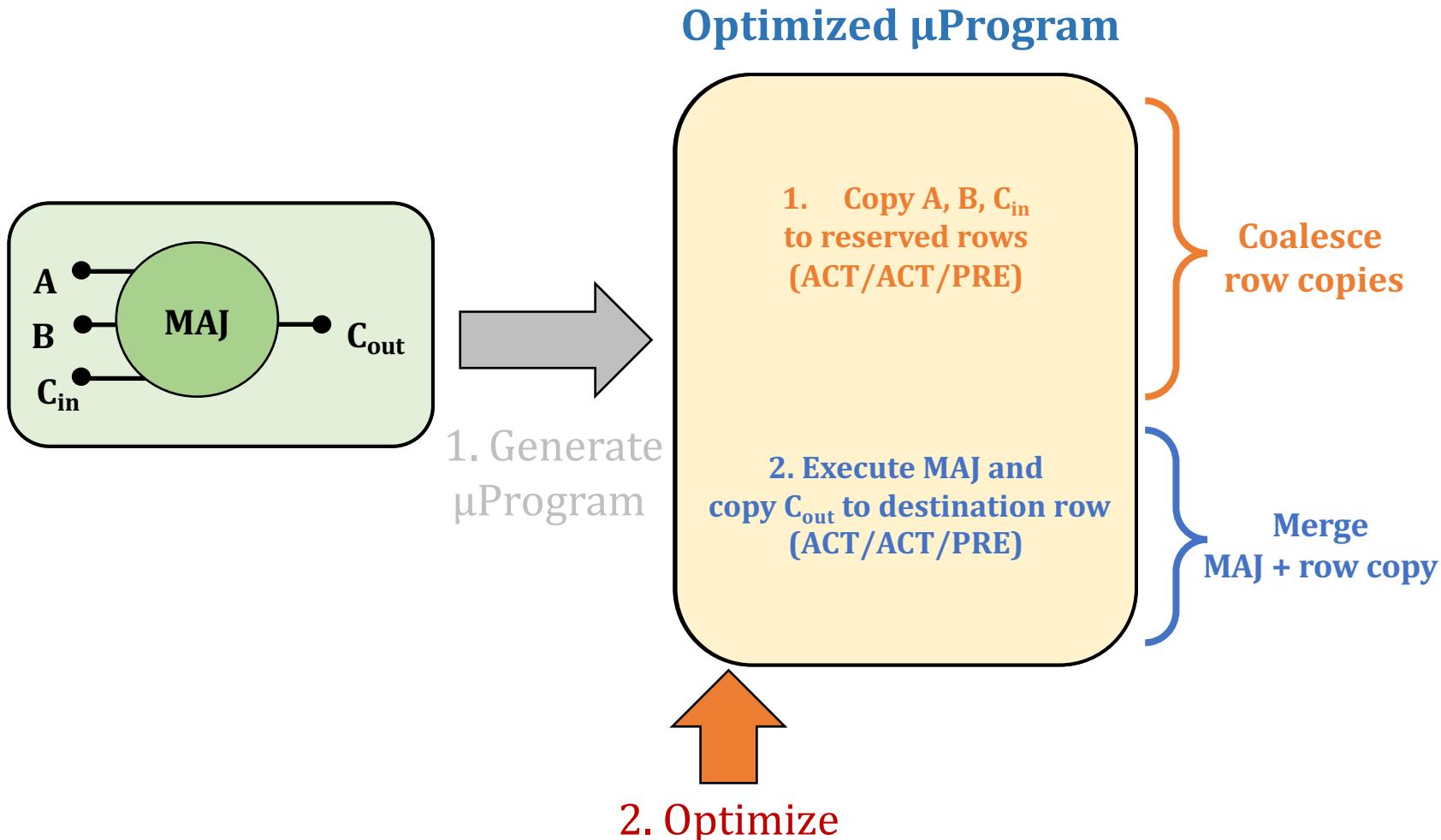
# Task 2: Optimize the $\mu$ Program



# Task 2: Optimize the $\mu$ Program

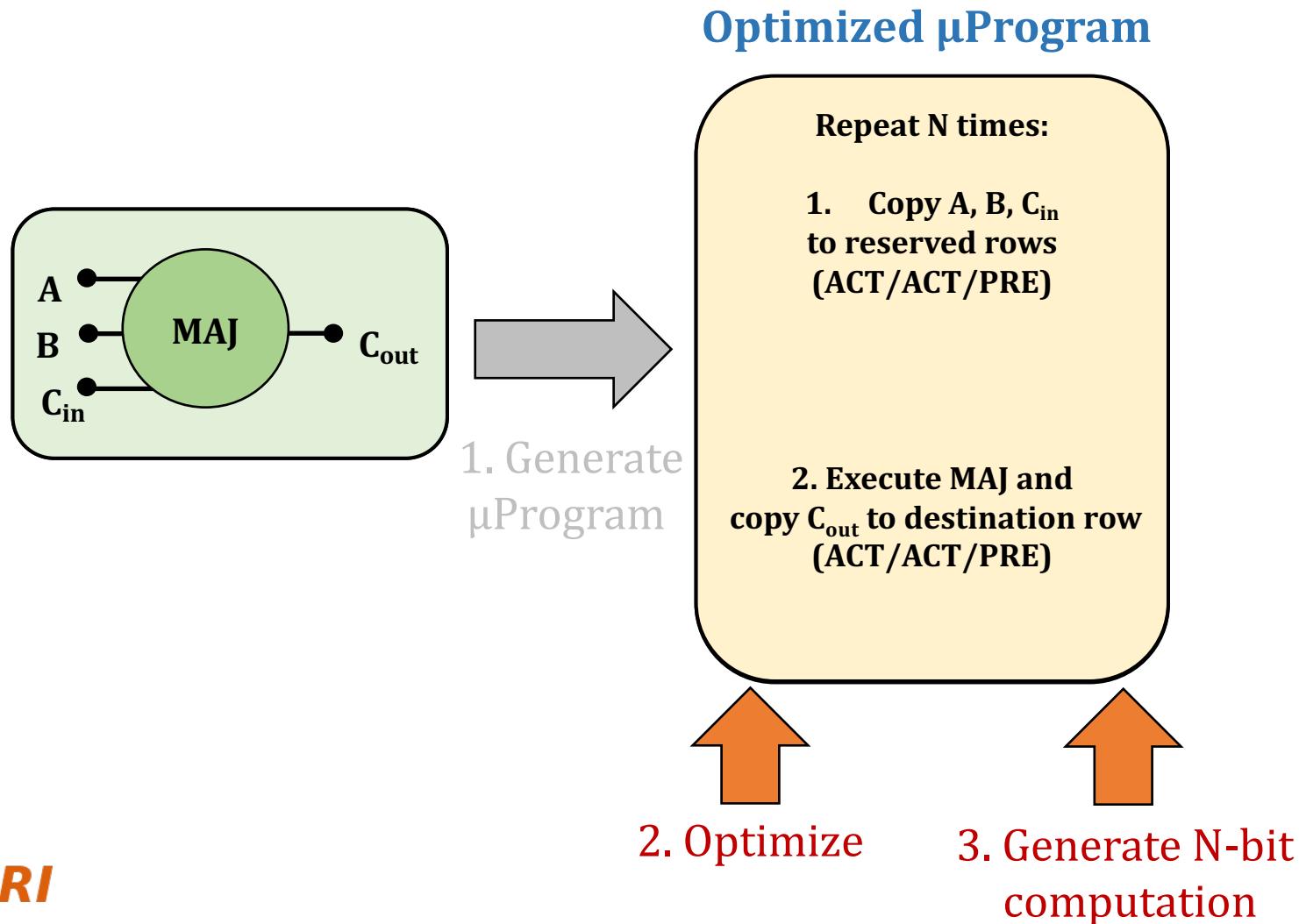


# Task 2: Optimize the $\mu$ Program



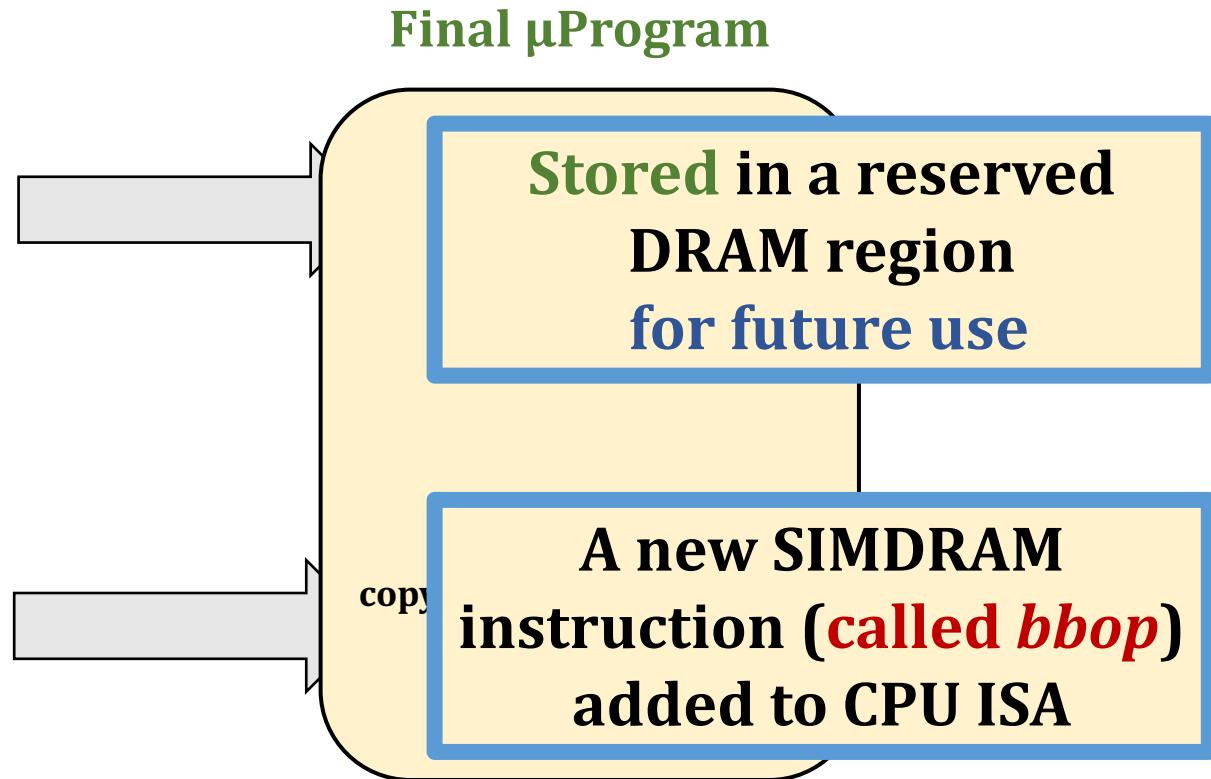
# Task 2: Generate N-bit Computation

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

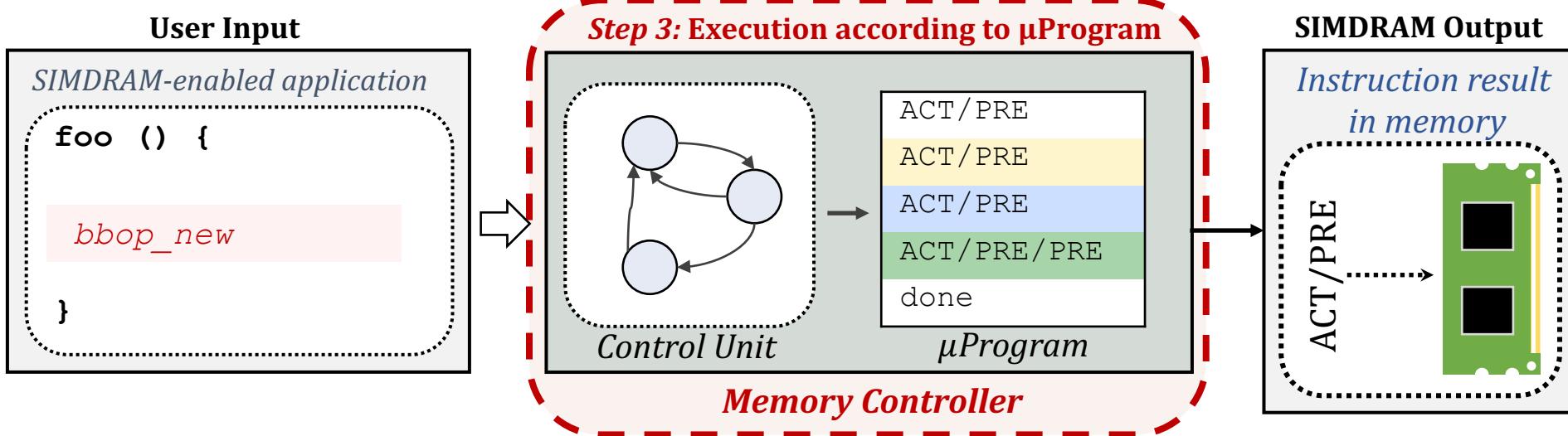
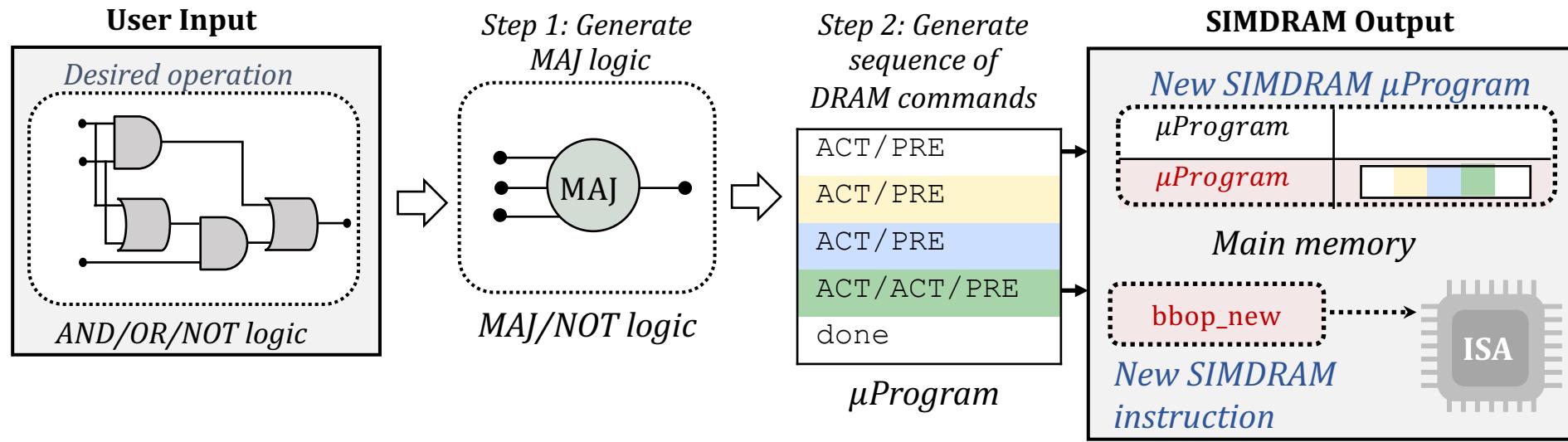


# Task 2: Generate μProgram

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

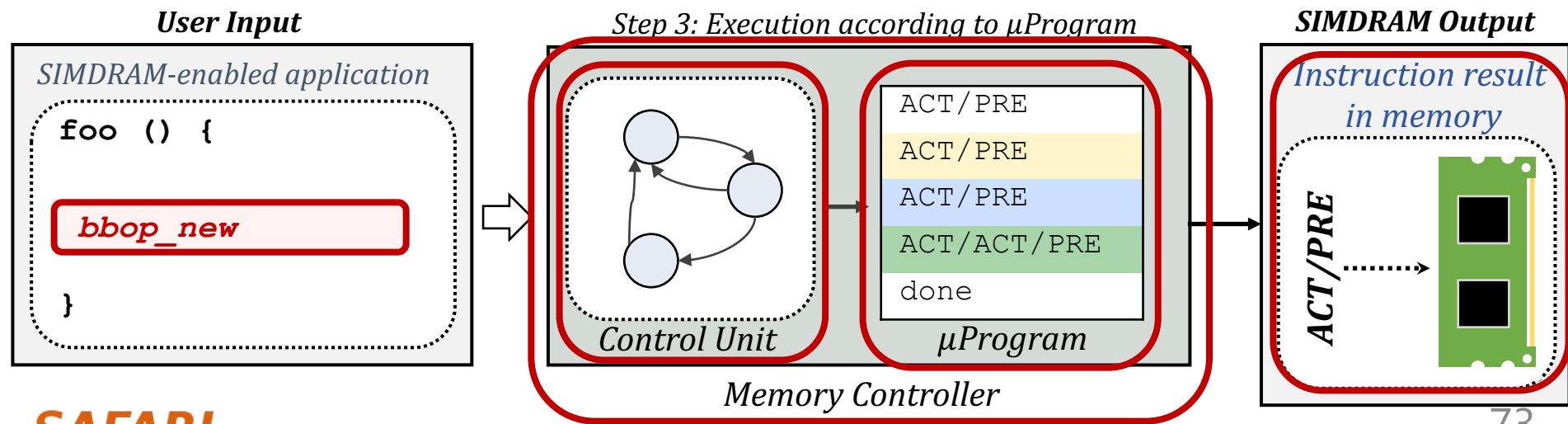


# SIMDRAM Framework: Step 3



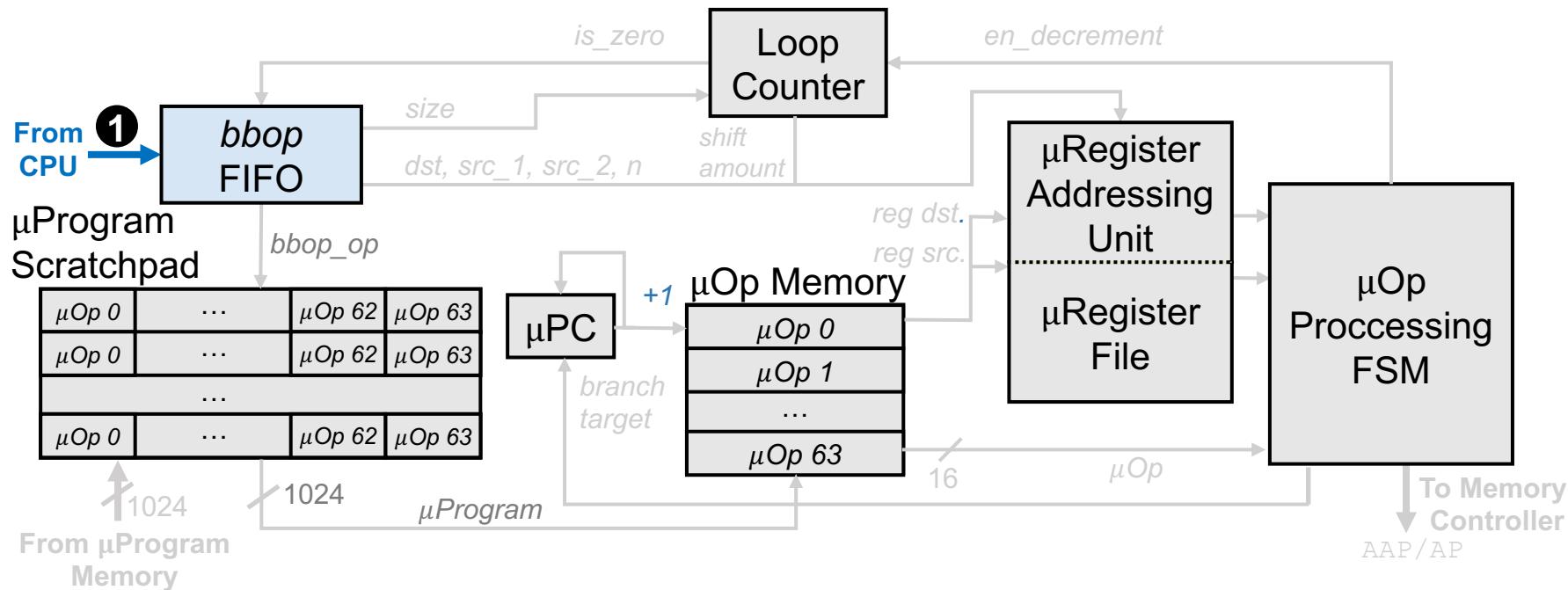
# Step 3: μProgram Execution

- **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



# Step 3: Operation Execution

- The SIMDRAm control unit works as follows:

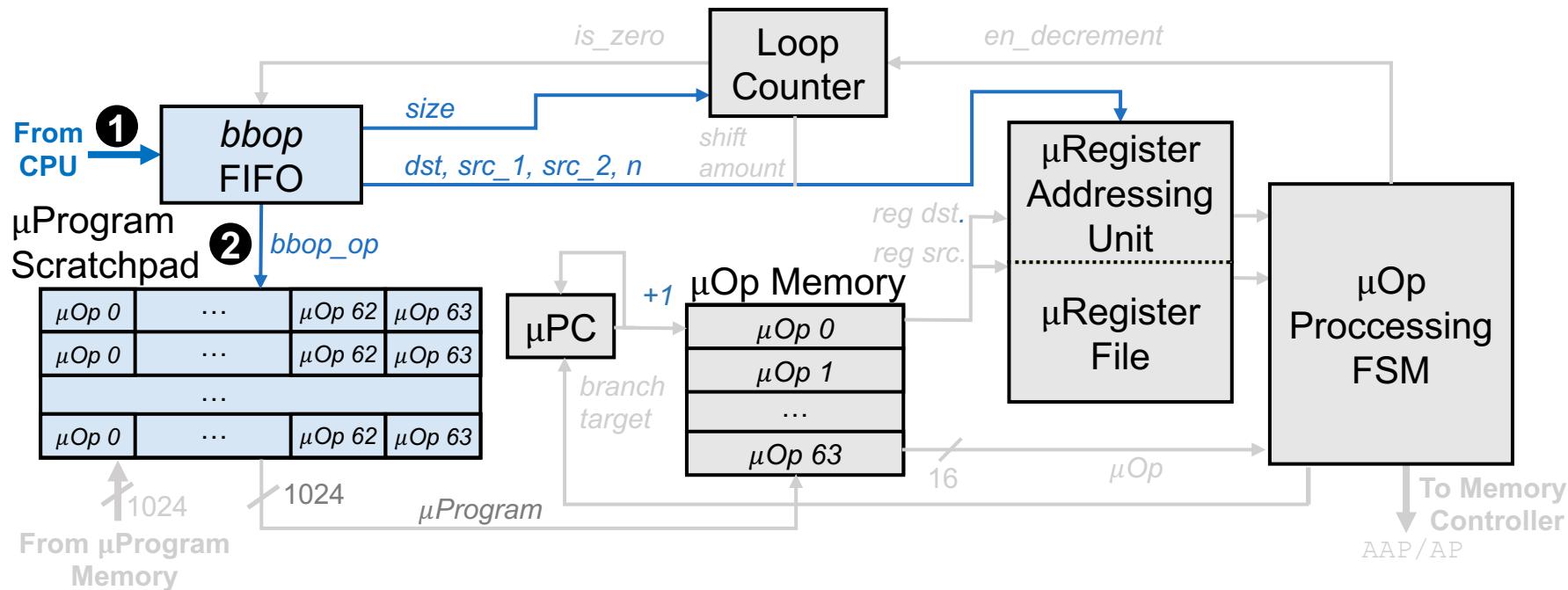


1

The CPU sends a bbop to the SIMDRAm control unit

# Step 3: Operation Execution

- The SIMDRAAM control unit works as follows:



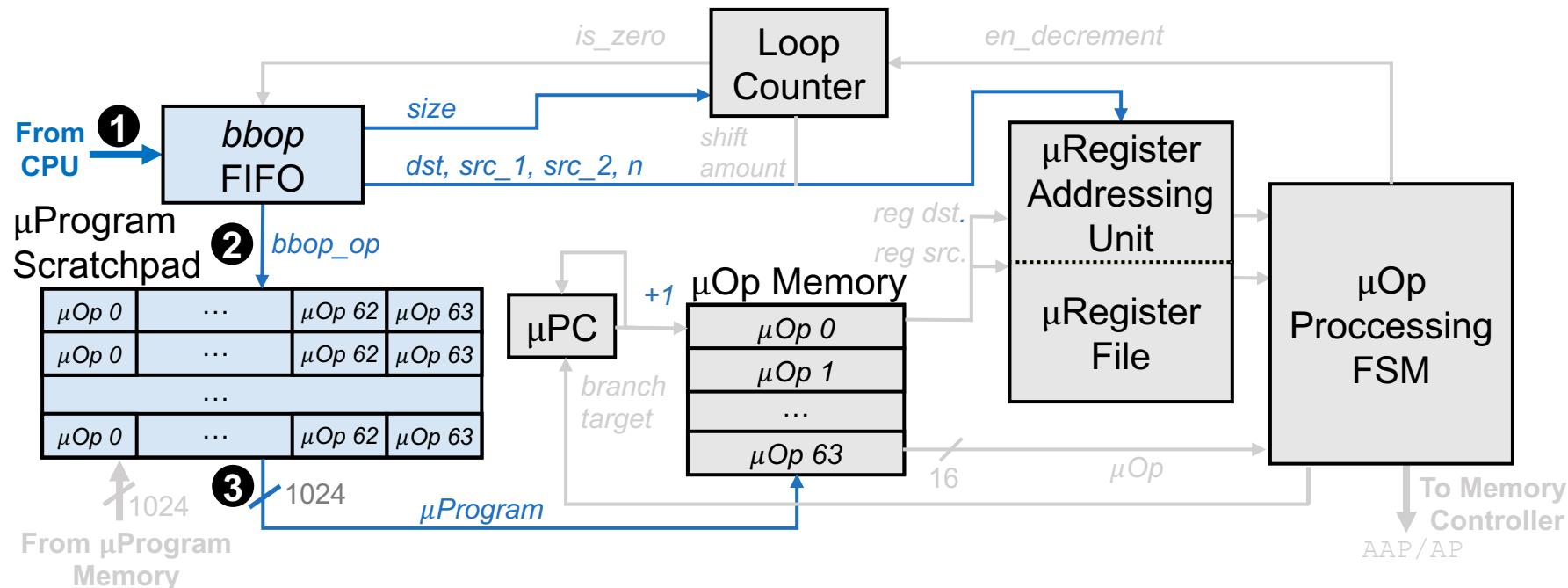
2

The bbop is decoded:

- μProgram in the μProgram scratchpad
- Set loop counter to operate on all elements
- Write sources, destination, and element size into μRegister Addressing Unit

# Step 3: Operation Execution

- The SIMD RAM control unit works as follows:

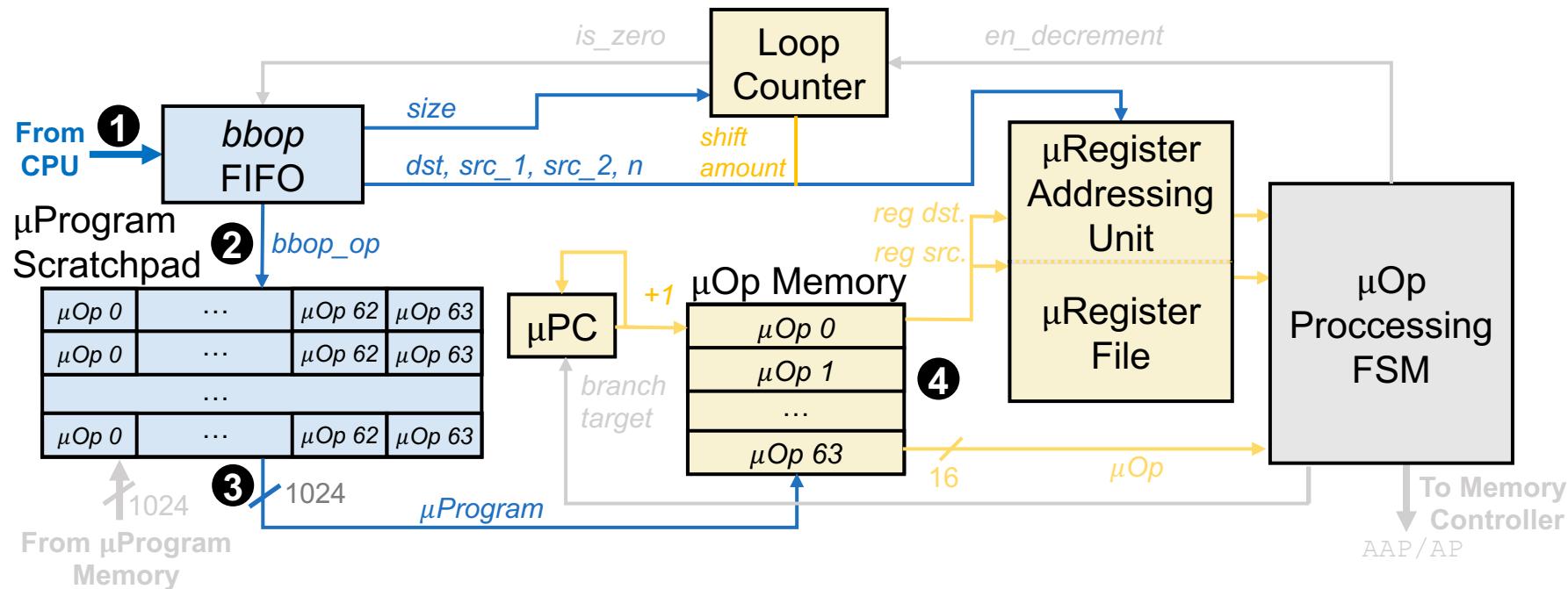


3

Copy the **μProgram** to the **μOp Memory**

# Step 3: Operation Execution

- The SIMD RAM control unit works as follows:

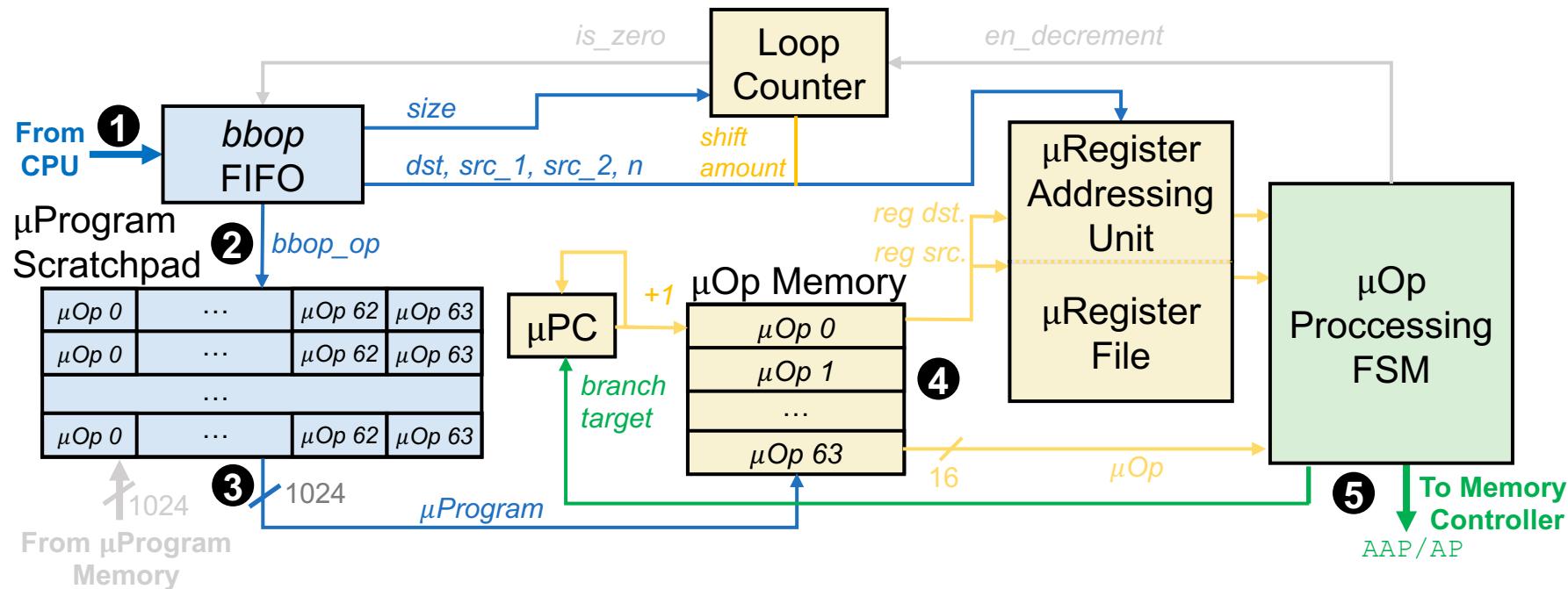


4

The μOp Processing FSM starts decoding μOps

# Step 3: Operation Execution

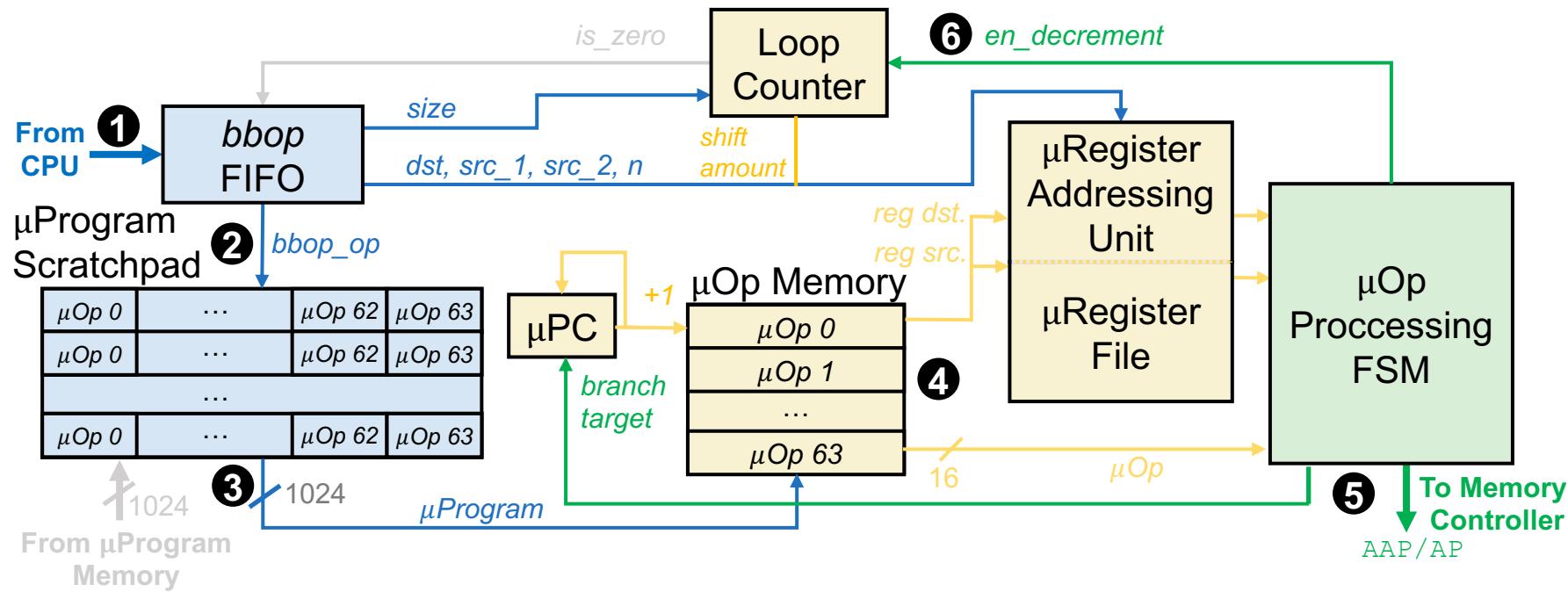
- The SIMDRAAM control unit works as follows:



5 The μOp Processing FSM issues commands to the memory controller (μPC is incremented)

# Step 3: Operation Execution

- The SIMD RAM control unit works as follows:

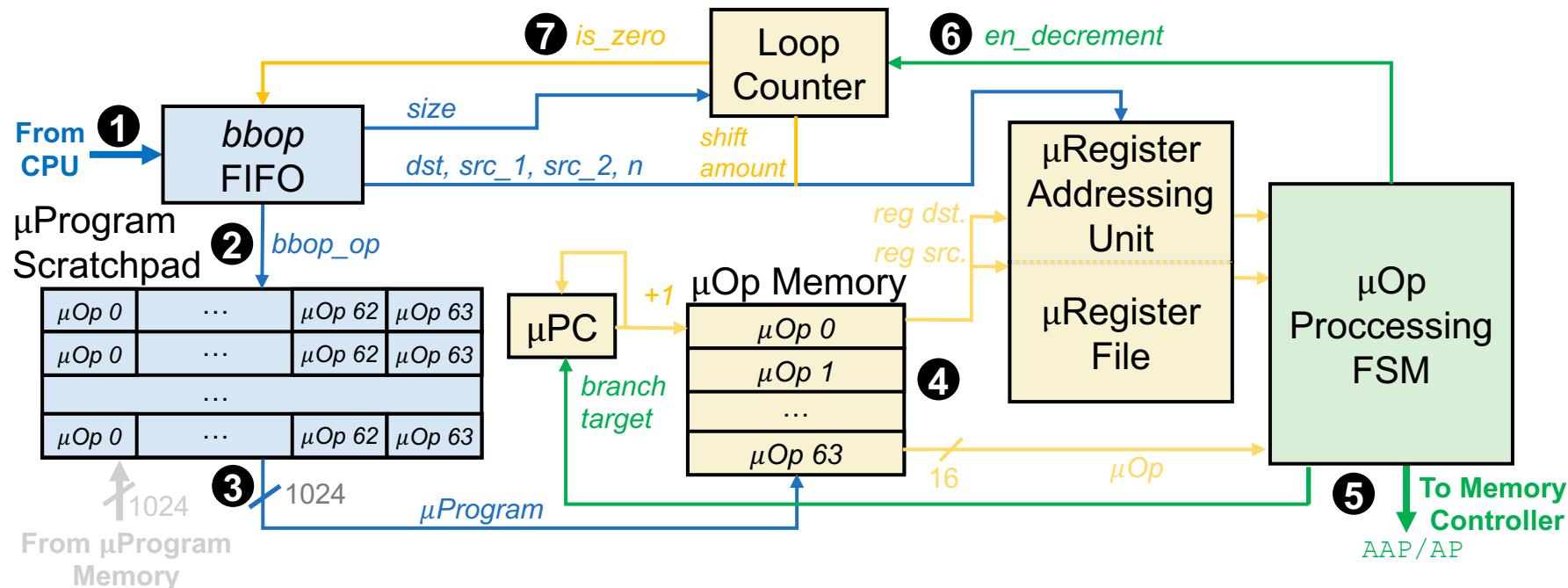


6

When done is found in the **μProgram**,  
the loop counter is decremented,  
and destination and source addresses shifted

# Step 3: Operation Execution

- The SIMD RAM control unit works as follows:



7

When the loop counter is zero,  
the control unit moves to the next bbop in the FIFO

# 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

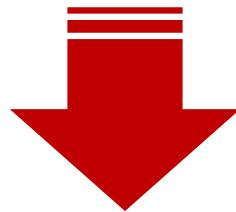
Handling limited subarray size

Security implications

Limitations of our framework

# Transposing Data

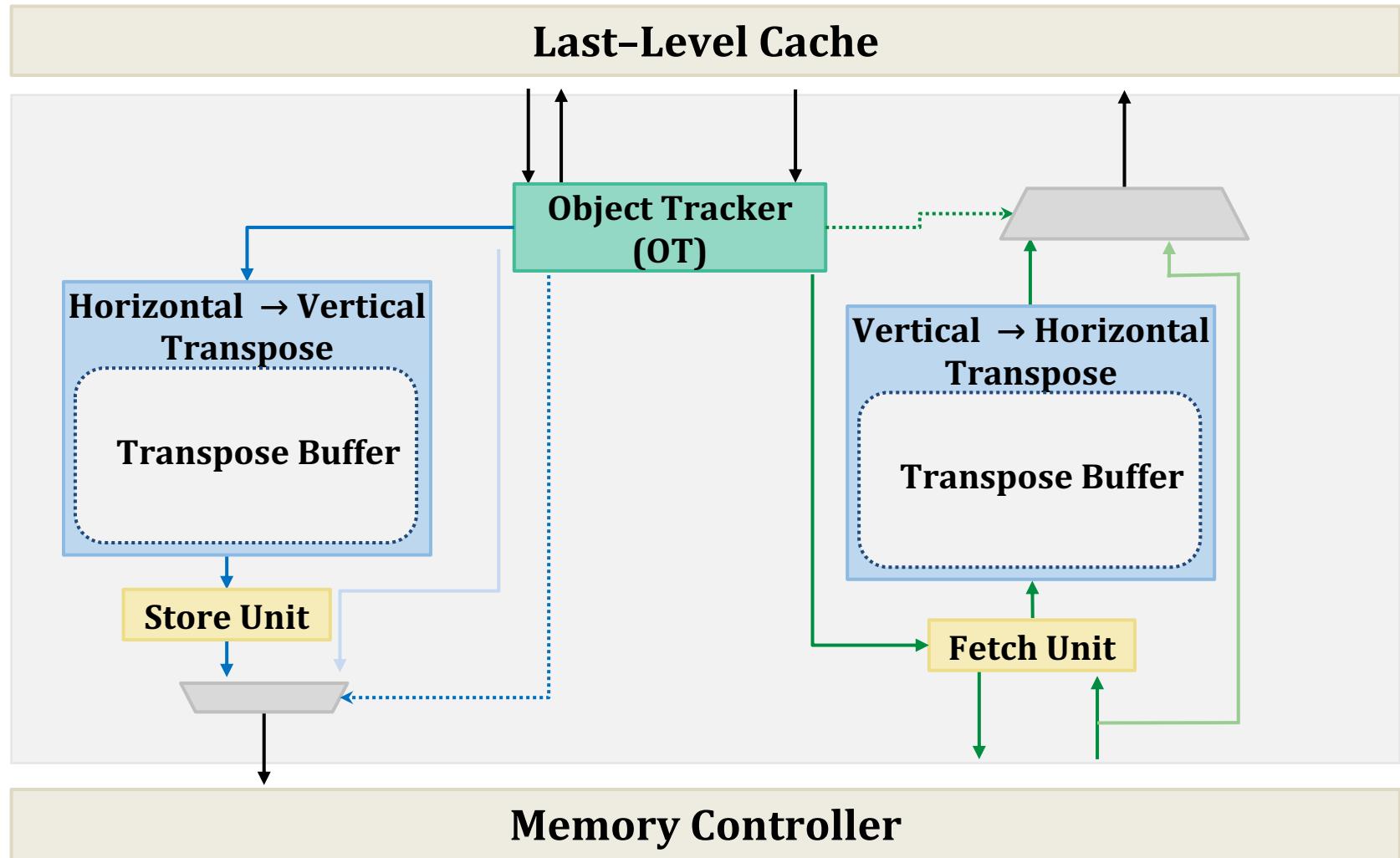
- SIMD RAM operates on vertically-laid-out data
- Other system components expect data to be laid out horizontally



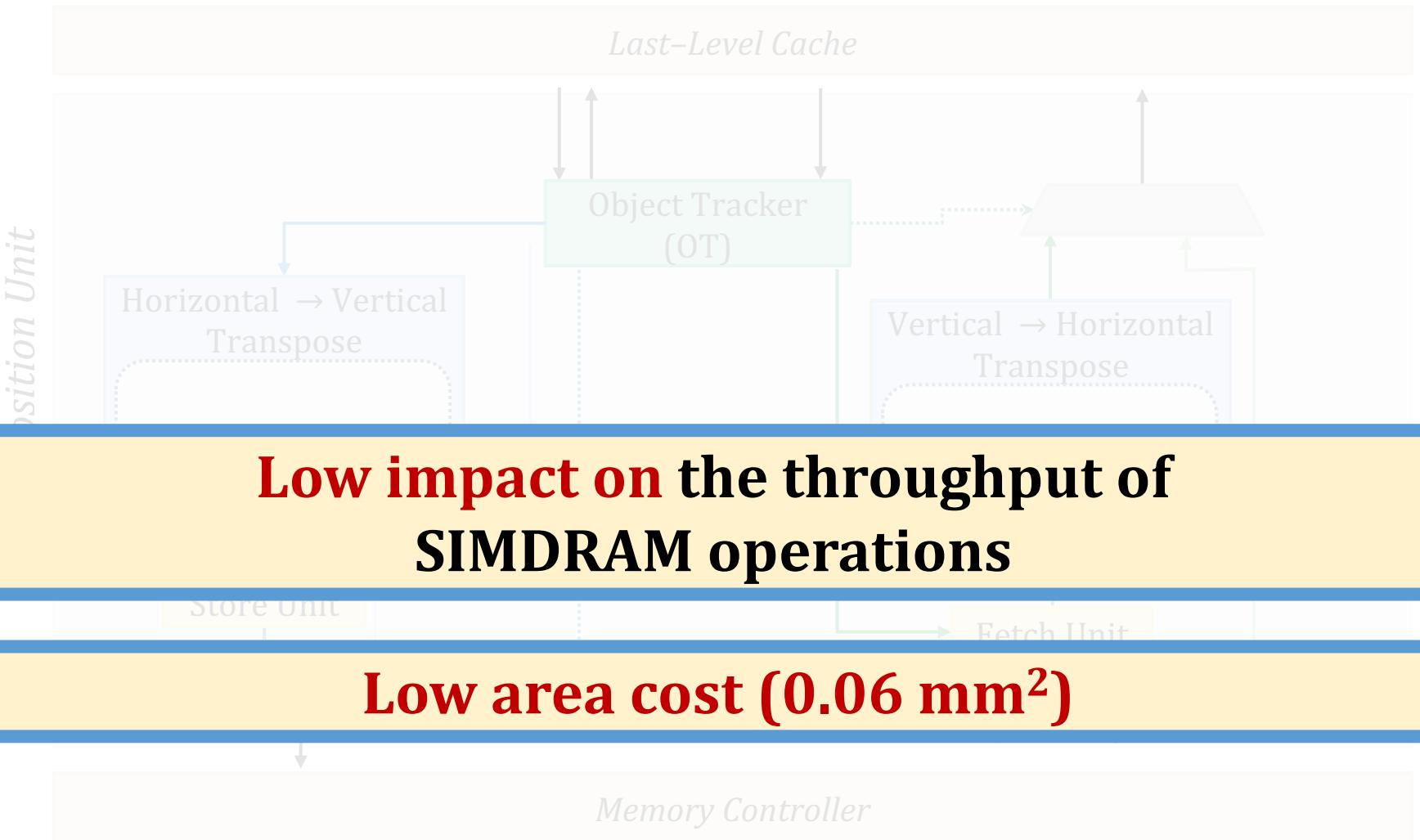
**Challenging to share data between SIMD RAM and CPU**

# 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<sup>1,2</sup>

Nika Mansouri Ghiasi<sup>1</sup>

\*Geraldo F. Oliveira<sup>1</sup>

Minesh Patel<sup>1</sup>

Sven Gregorio<sup>1</sup>

Mohammed Alser<sup>1</sup>

João Dinis Ferreira<sup>1</sup>

Saugata Ghose<sup>3</sup>

Juan Gómez-Luna<sup>1</sup>

Onur Mutlu<sup>1</sup>

<sup>1</sup>ETH Zürich

<sup>2</sup>Simon Fraser University

<sup>3</sup>University 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

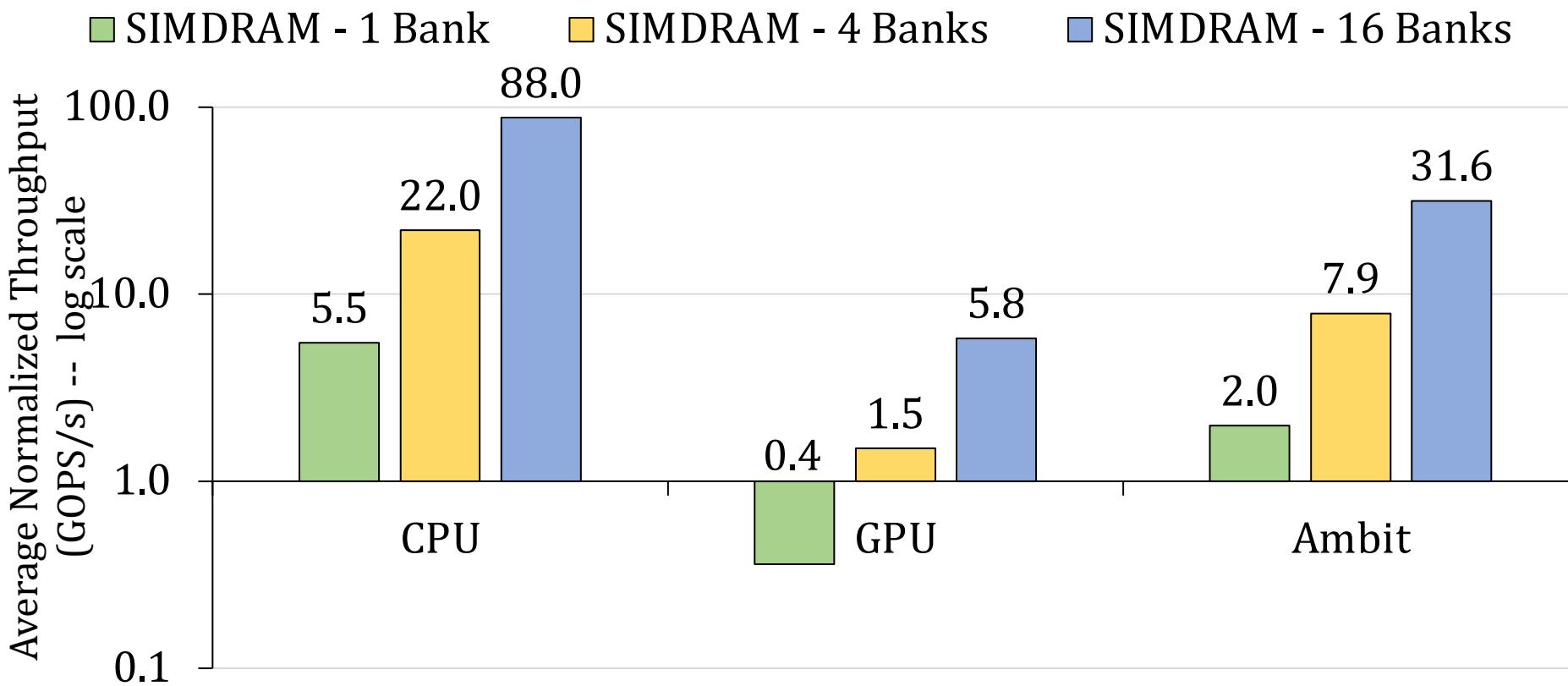
# Methodology: Workloads

## 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)
  - TPC-H (databases)
  - kNN (machine learning)
  - LeNET (Neural Networks)
  - VGG-13/VGG-16 (Neural Networks)
  - brightness (graphics)

# Throughput Analysis

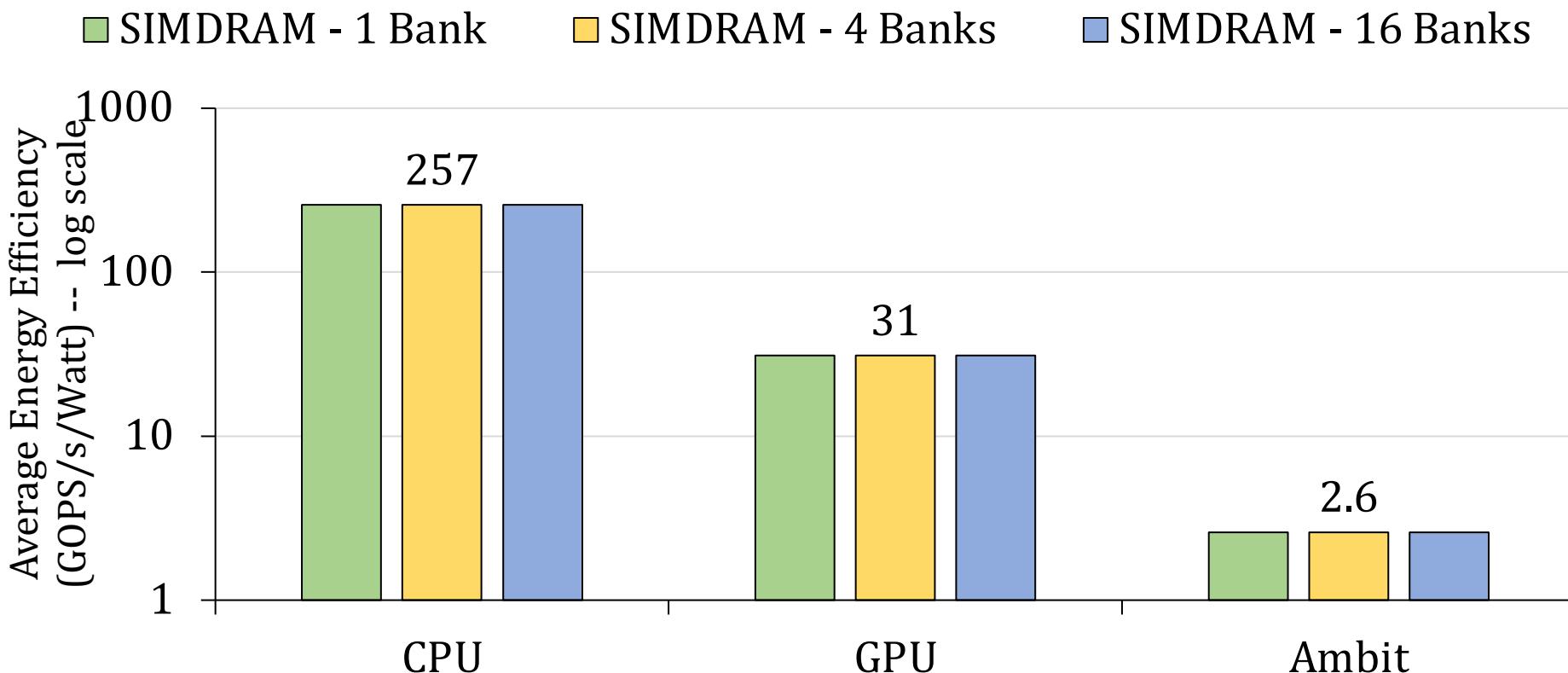
Average normalized throughput across all 16 SIMDRAM operations



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

# Energy Analysis

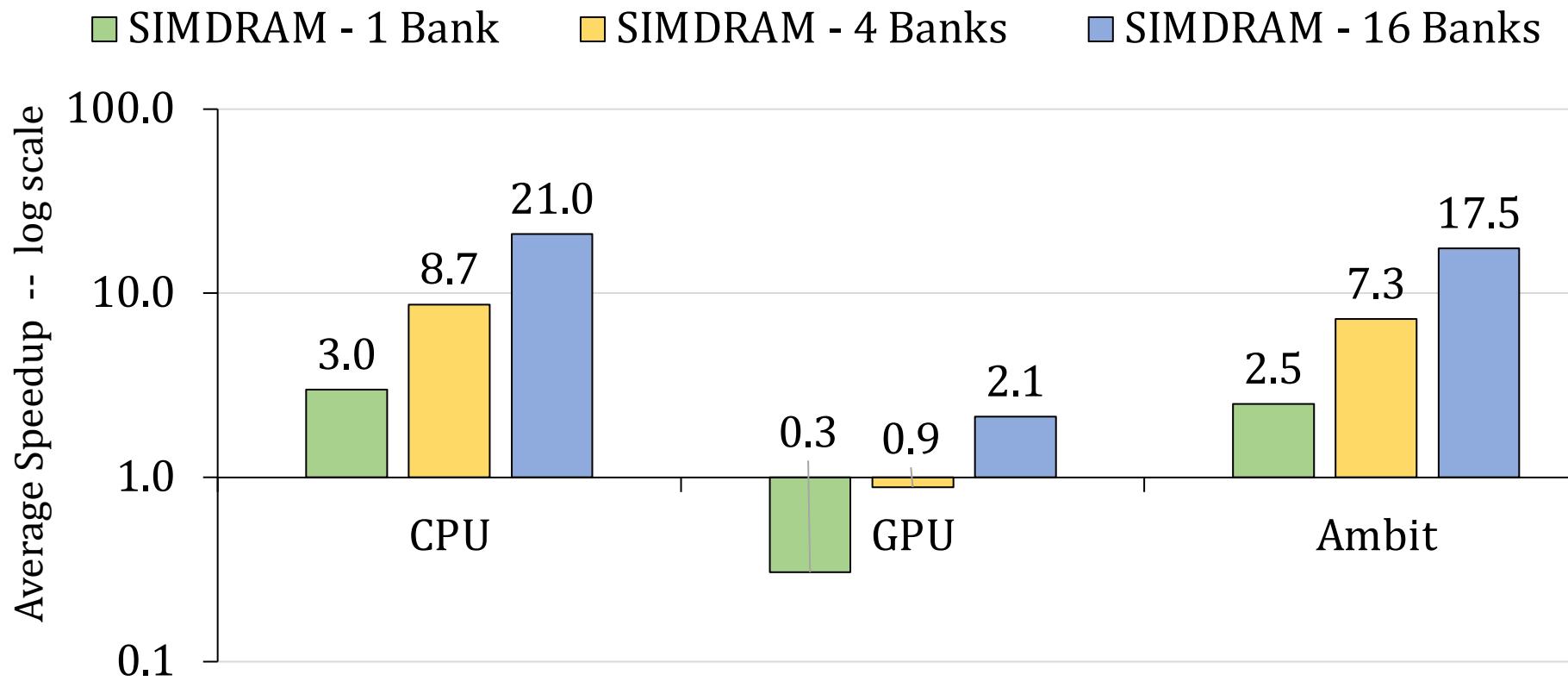
Average normalized energy efficiency across all 16 SIMDRAM operations



**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

# 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*

**Nastaran Hajinazar\***

Sven Gregorio

Joao Ferreira

**Geraldo F. Oliveira\***

Nika Mansouri Ghiasi

Minesh Patel

Mohammed Alser

Saugata Ghose

Juan Gómez-Luna

Onur Mutlu

**SAFARI**



SIMON FRASER  
UNIVERSITY

**ETH** Zürich



UNIVERSITY OF  
**ILLINOIS**  
URBANA-CHAMPAIGN

# P&S Processing-in-Memory

Bit-Serial SIMD Processing  
using DRAM

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Fall 2021

7 December 2021