

# Memory Subsystem of Microprocessors



Chun-Jen Tsai  
NYCU  
11/04/2022

SRAM : On-Chip memory, use transistors, low density.

# Bottleneck of a Computer

DRAM : Use in Main Mem., use capacitors, high density.

DRAM stores data by capacitors, it determines the charge stored in a capacitor. Every R/W is a process of : discharge (read)  $\rightarrow$  charge (write back). DRAM has to write back even when no access due to leakage.

- ❑ Today, the bottleneck of a computer is often in the main memory (DRAM) subsystem

- CPUs do parallel ops (by multi-hart<sup>†</sup>, multi-core, or superscalar)
- DRAM devices usually handle one transaction at a time
- CPU clocks often more than 4× faster than DRAM clocks

- ❑ Most importantly, CPU&DRAM communicate differently

- CPU – word-based read/write instructions
- DRAM – block-based data transactions
- A memory controller sits between the CPU and DRAM as a mediator to alleviate the problem, but it is still a bottleneck

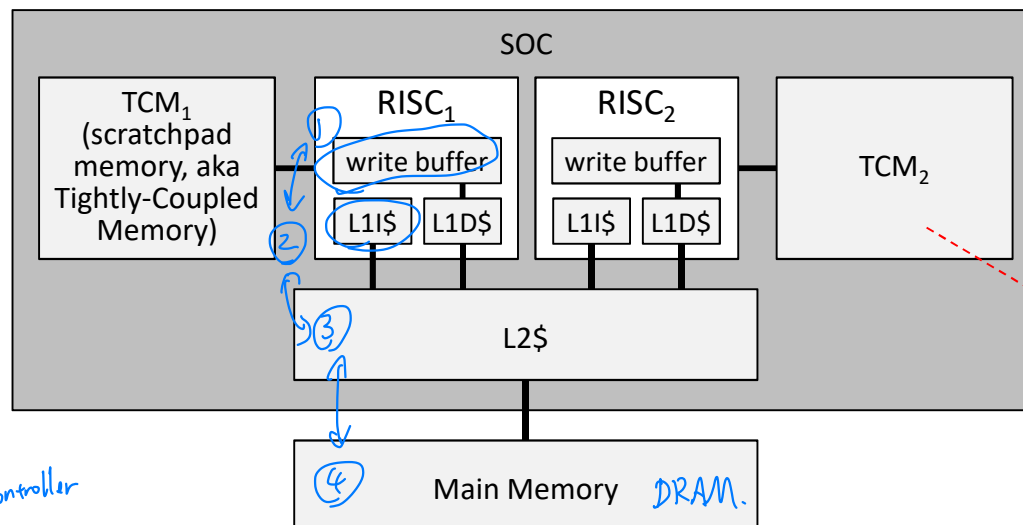
$\rightarrow$  process in memory: Samsung.

<sup>†</sup> Hart stands for “hardware thread.” Also known as Hyper-thread or Simultaneous multi-thread.



# Memory Hierarchy of Computers

- ❑ A computer has several levels of memory hierarchy
  - L1, L2, and L3 caches
  - Scratchpad has lower delays than L1\$, and much cheaper
  - Main memory – usually DRAM
- ❑ **Coherency** among different memory blocks is a critical issue (regardless of the # of cores)



linker script can control whether  
data is placed.

0x00000000 → TCM  
→ \$

Should TCM<sub>1</sub> and  
TCM<sub>2</sub> be in the same  
address space?

TCM v.s Cache  
SRAM.  
Tagged memory  
controlled by cache controller  
controlled by programmer

# Memory-Centric Applications

---

- ❑ Many computers run memory-centric applications:
  - Big-data analysis
  - Deep-learning
  - Network applications
  - Multimedia applications
  
- ❑ The memory behaviors of a RISC, a Digital Signal Processor (DSP), and a hard-wired logic are quite different

# Memory-Centric Processing (1/3)

□ Assuming two functional units (FUs) of computations:

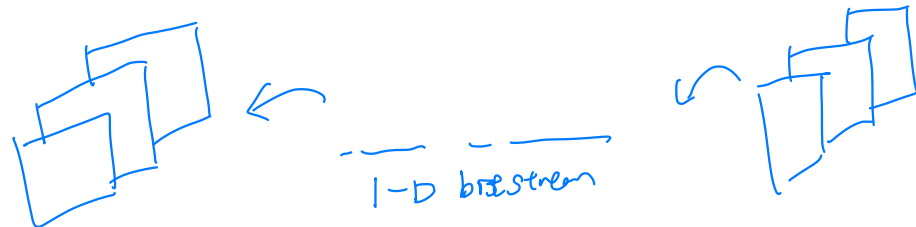
## ■ FU#1

```
for (idx = 0; idx < N; idx++) for (jdx = 0; jdx < N; jdx++)  
{  
    C[idx][jdx] = A[idx][jdx] + B[idx][jdx];  
}
```

## ■ FU#2

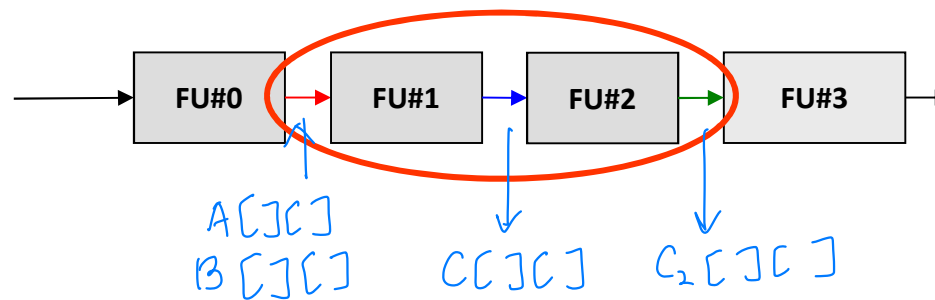
```
for (idx = 0; idx < N; idx++) for (jdx = 0; jdx < N; jdx++)  
{  
    row = scan_row[idx], col = scan_col[jdx];  
    C2[idx][jdx] = C[row][col];  
}
```

FU#1, FU#2 are common in  
video codec

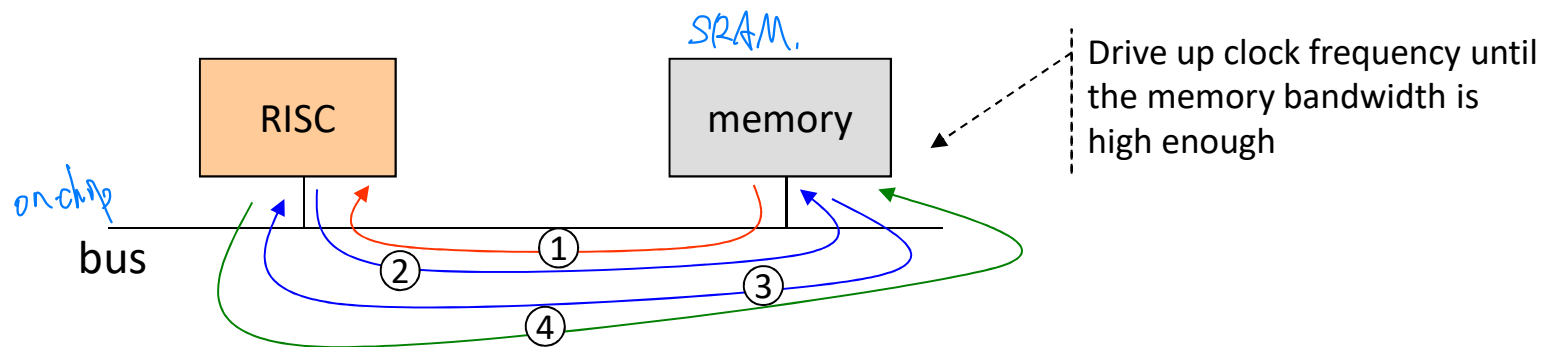


# Memory-Centric Processing (2/3)

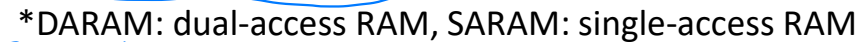
- ❑ System blocks:



- ❑ CPU without using scratchpad memory:



- ❑ **Hardwired accelerator solution:**



7/42

# Memory Model of a Processor

- ❑ The memory model specifies multiple harts (HW threads) data R/W behavior w.r.t. shared memory
  - For a single thread or multiple threads without shared data, nothing needs to be worried about as long as the data dependency order has been followed strictly

```
int A = 0;  
int B = 0;
```

```
Thread_1()  
{  
    register int C1;  
    A = 1;  
    C1 = B;  
}
```

```
Thread_2()  
{  
    register int C2;  
    B = 1;  
    C2 = A;  
}
```

Can we have C1 = 0 and C2 = 0 at the end of both threads?



# Processor/Compiler Tricks on L/S

---

- ❑ A memory model defines the degree of flexibility a processor (or compiler) can have in re-ordering and elimination of the load-store instructions
  - Under a memory model, a programmer must take into account possible re-ordering done by the processor/compiler
- ❑ When compiler optimization is on, some memory operations may be removed
  - Compilers may not see all threads with a shared variable!
  - “`volatile`” keywords in C/C++ is used by a programmer to tell the compiler not to play smart

# Example of Volatile Usage

- ❑ Use `volatile` to avoid optimization errors:

```
int C;  
  
Thread()  
{  
    int A = 0;  
  
    C = 1;  
    while (C) /* busy waiting */;  
  
    A = 1;  
}
```

The thread can never reach here!

```
volatile int C;  
  
Thread()  
{  
    int A = 0;  
  
    C = 1;  
    while (C) /* busy waiting */;  
  
    A = 1;  
}
```

The thread can reach here!

# Is Memory Model Necessary?

---

- ❑ Since the main memory are much slower than the  $\mu P$ , memory accesses are cached, buffered, and pipelined
- ❑ The memory model defines how read/write requests can be serialized to maintain “program correctness” while achieving high bandwidth and low latency<sup>†</sup>

---

<sup>†</sup> K. Gharachorloo, et al., “Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors,” ACM SIGARCH Computer Architecture News, Vol. 18, Issue 2, June 1990, pp.15-26.

# Sequential Consistency Model

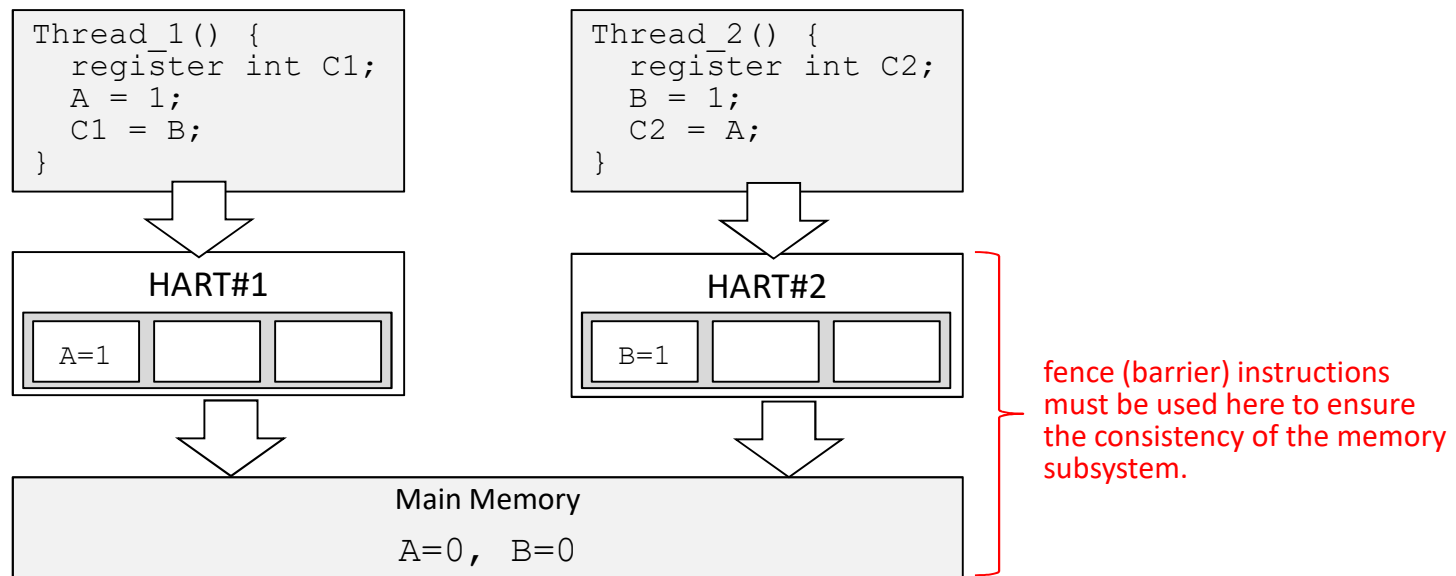
- ❑ The Sequential Consistency (SC) model assumes that each thread executes only in program order, and all threads share a single-port main memory
  - When interleaving instructions from all threads, the instruction order of each thread are preserved. From page 8 example:

A = 1; C1 = B; B = 1; C2 = A;	A = 1; B = 1; C1 = B; C2 = A;	A = 1; B = 1; C2 = A; C1 = B;	B = 1; C2 = A; A = 1; C1 = B;	B = 1; A = 1; C2 = A; C1 = B;	B = 1; A = 1; C1 = B; C2 = A;
----------------------------------------	----------------------------------------	----------------------------------------	----------------------------------------	----------------------------------------	----------------------------------------

- SC model means that each read operation (C1 or C2 in the example) from all threads on a variable sees the last write operation on that variable
  - Implies full coherency across all layers of the memory subsystem

# Total Store Order (TSO) Model

- ❑ TSO model only requires preserving the store order of each thread (without violation of data dependency)
- ❑ If write buffers are used in the processors, different harts may see different values of the same variable
  - Under TSO model, previous example can have  $C1 = C2 = 0$ :



# Data Race Issue

---

- ❑ A data race condition happens when
  - Concurrent accesses to a shared variable includes a write plus one or more read/write operations
  - Instructions from all harts are not just interleaved, but also executed during the same cycles → a multi-cycle operation can be preempted
- ❑ For race-free programs, compilers often offer a sequential consistency guarantee
  - Fence instructions will be inserted to preserve the appearance of sequential consistency

# Weak Memory Model

---

- ❑ Modern superscalars usually adopt a weak memory model that allows aggressive reordering of the instructions for ultimate performance
- ❑ The consistency among threads must be explicitly enforced by the programmers/compiler using fence, barrier, or atomic instructions
- ❑ A  $\mu\text{P}$  with a weak memory model may have better multi-thread performance than a  $\mu\text{P}$  with a TSO model

# Non-Idempotent Memory

- ❑ For memory-mapped I/O addresses, memory cells may not behave in conventional ways
  - Writing to an address may store a different value in the cell
  - Writing to address A may change the content of address B
- ❑ Example: in the I/O addresses of Aquila, the content of `*uart_status` changes due to writing of `*uart_txfifo`:

```
#define uart_rxfifo ((unsigned int volatile *) 0xC0000000)
#define uart_txfifo ((unsigned int volatile *) 0xC0000004)
#define uart_status ((unsigned int volatile *) 0xC0000008)
```

```
while (*uart_status & TX_FIFO_FULL) /* wait */;
    *uart_txfifo = (unsigned char) '\r';
```



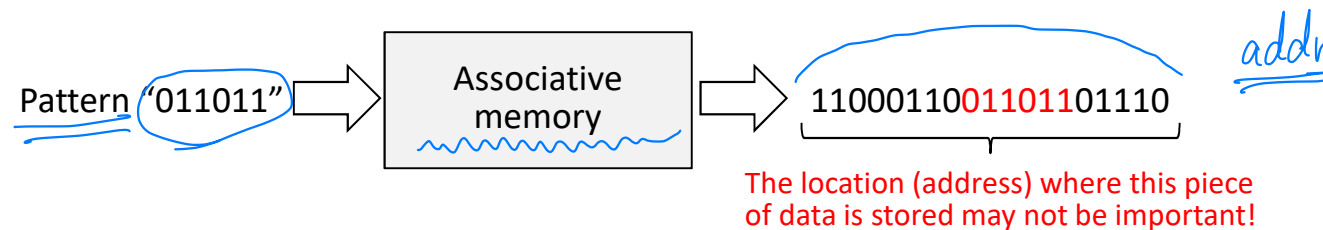
# Cache Memory

---

- ❑ A cache is a high-speed memory that sits between the processor core and the slow main memory
  - Stores most frequently accessed data
- ❑ Three types of caches in a CPU core:
  - Instruction cache – stores read-only instructions
  - Data cache – stores read/write data and possibly instructions
  - TLB cache – stores virtual memory table entries
- ❑ A cache memory is essentially an extremely simple type of associative memory

# Associative Memory

- ❑ Associative memory can be used to retrieve a stored data by a “bit pattern” instead of an address
  - Tagged memory is a special type of associative memory



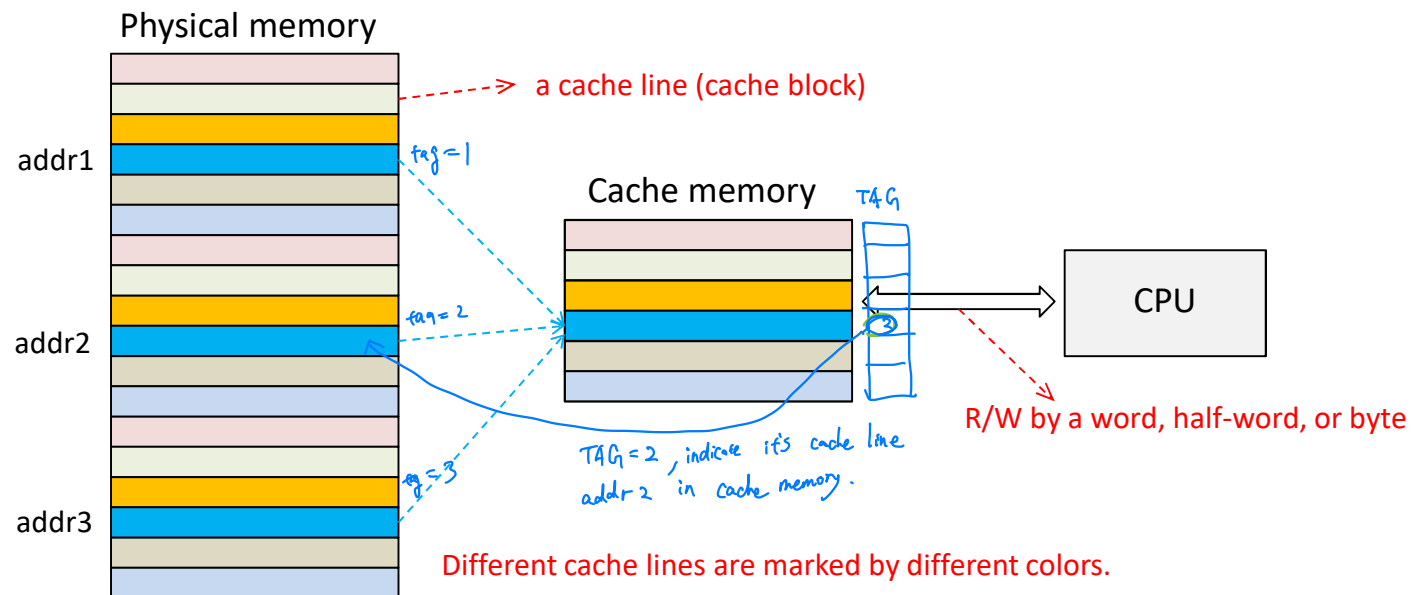
- ❑ Examples of associative memory usage:
  - Human brains
  - Artificial neural networks
  - Network filters/routers

TAG-memory.

data	tag

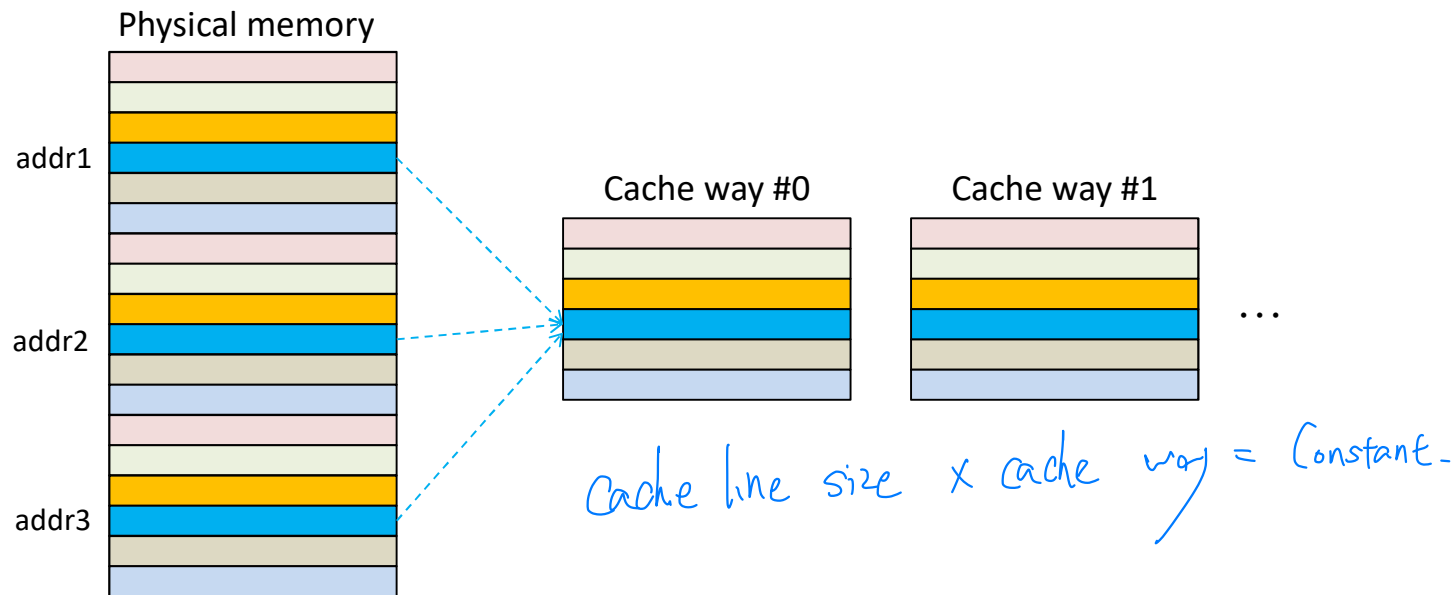
# Caches Design Principle

- ❑ Data exchange between physical memory and cache operates at large blocks called a cache line
  - Typical cache line size: 8 to 64 bytes
- ❑ Each cache line contains a “tag” that indicate which main memory cache line the data come from



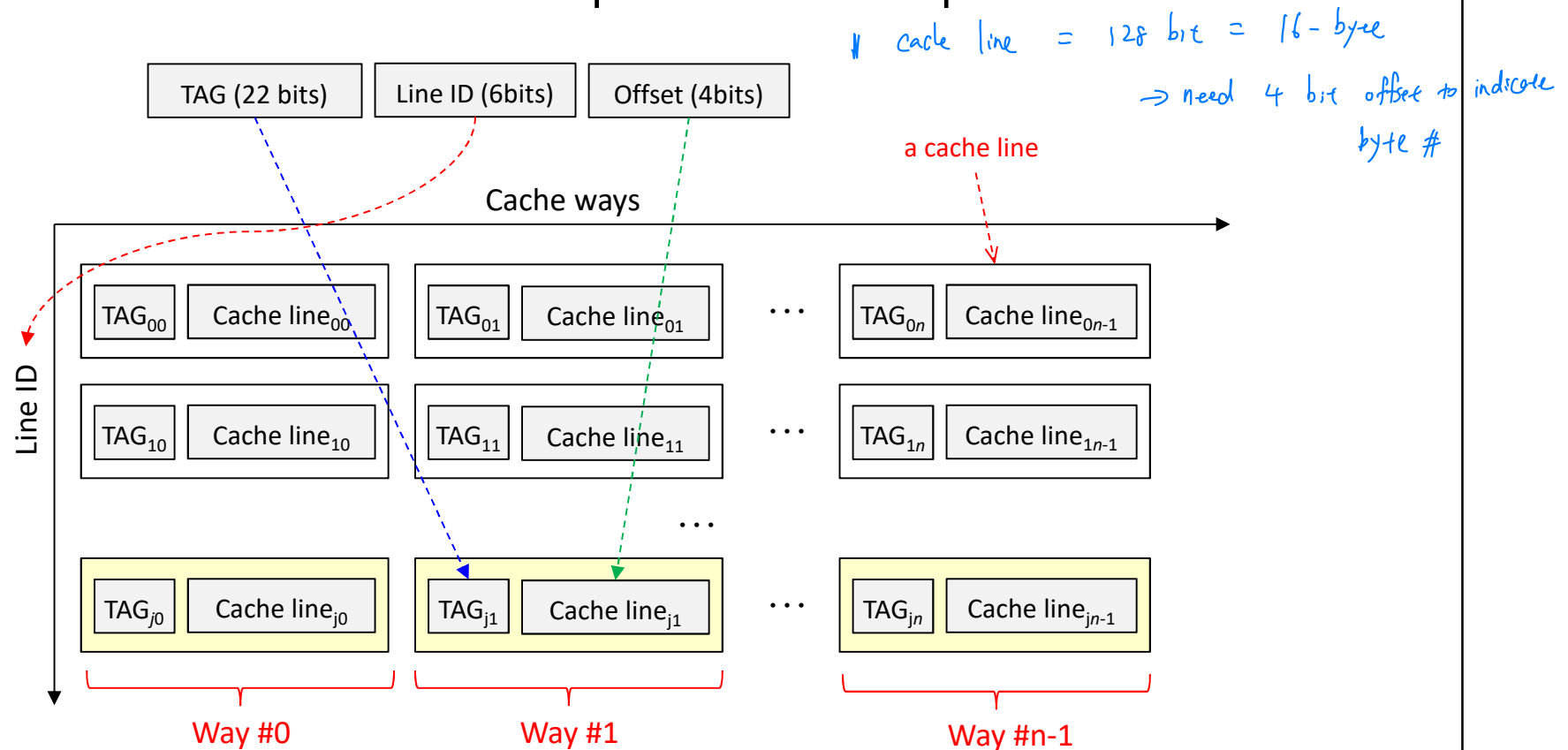
# N-Way Cache Memory

- ❑ For n-way cache, we need to determine which “way” contains the target data using associative memory
  - A 1-way cache is also called a direct-mapping cache
  - Cache memory is also called tagged memory



# Tagged Memory Structure

- Tagged memory has a two-dimensional structure:
  - A 32-bit address is decomposed into three parts:



# Cache Structure of Aquila

---

- ❑ For Aquila on Arty, the cache parameters are:
  - 4-way set associative
  - 128-bit cache line
  - Adjustable cache size
- ❑ If the cache size is 4KB, a 32-bit address is factored into 22-bit tag, 6-bit line index, and 4-bit byte offset
  - When cache size increases, the tag bits will drop and the line bits will increase
  - If the cache way increases (and cache size fixed), the line index bits drop, and the tag bits and offset bits do not change

# Cache Design Parameters (1/2)

---

- ❑ Cache line size (aka. cache block size)
  - Cache exchange data with DRAM one line at a time
  - Cache line size does not have to match the DRAM block size, but integer ratios between these two are preferred  
*usually limited to the memory controller.*
- ❑ Cache write policy: when to write cache lines to DRAM
  - Write through *(flush every time)*
    - Write a block back upon any write request. Simplify data coherency issues. Less performance.
  - Writeback
    - Write back only when necessary. Better performance.
  - Write allocate
    - Write upon cache-miss, read the cache line and write to cache

# Cache Design Parameters (2/2)

---

## ❑ Cache replacement policy:

- When all the cache ways are full, the cache line in which cache way should be replaced by the new incoming data?
- Popular policies are first-in-first-out (FIFO) and least-recently used (LRU)

## ❑ Cache associativity

- Direct-mapping (no associative memory) *1-way cache*
- $n$ -way set associative

## ❑ Cache prefetching

- Do we prefetch data ahead of time?

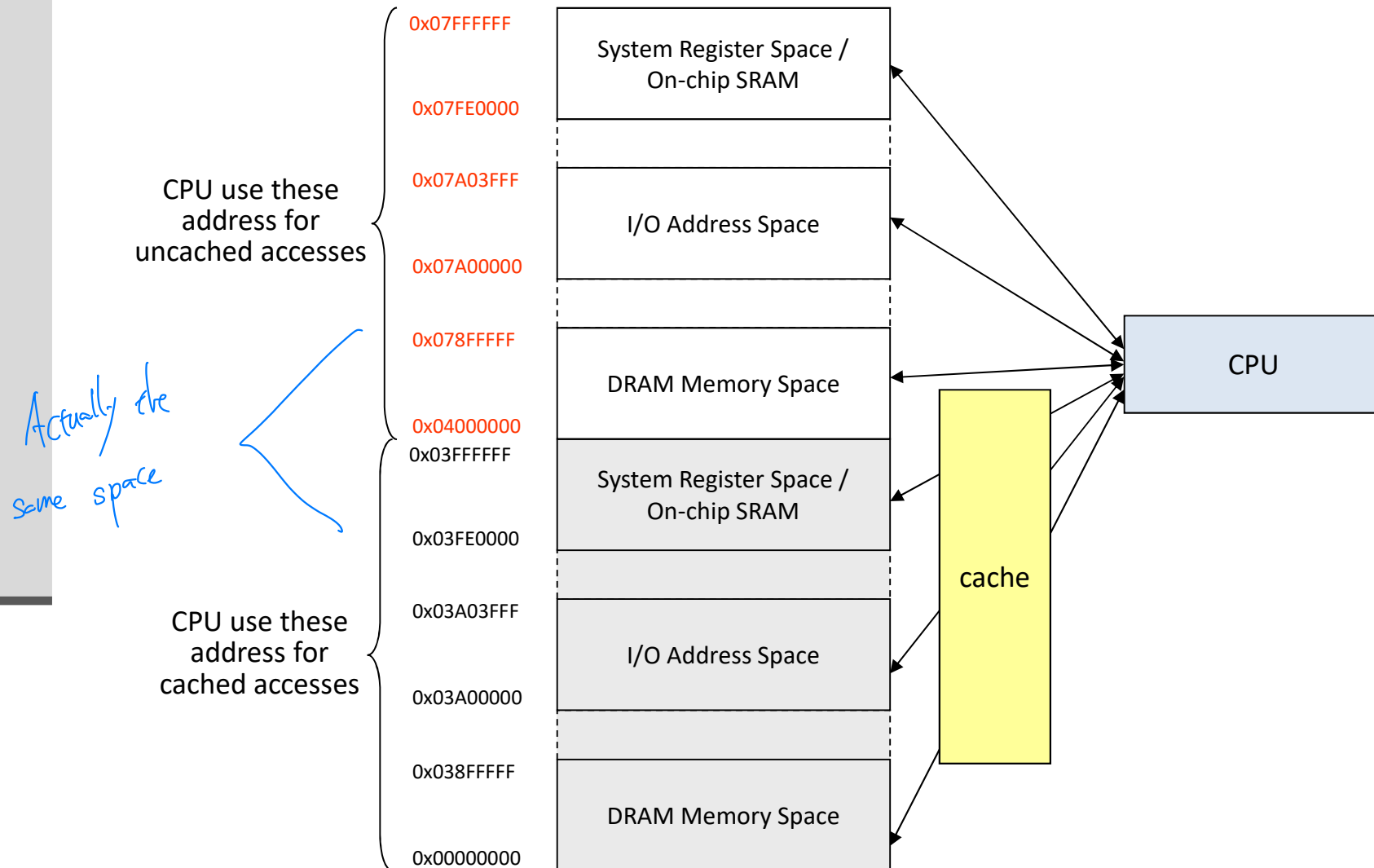


# Uncached Memory Accesses

---

- ❑  $\mu$ P also needs a way to bypass cache controller to access memory directly
  - I/O addresses space should not be cached since they are non-idempotent memory cells
- ❑ There are different schemes to do this
  - Some  $\mu$ Ps use system registers to define non-cacheable memory blocks
  - Some  $\mu$ Ps use MMU to define non-cacheable memory pages
  - Some  $\mu$ Ps divide the memory address space into cacheable and uncachable addresses (see next slide)

# Illustration of Uncached Accesses



# Snooping-Based Coherent Cache

## ❑ Goals of memory coherency

- Strong: any read of an address must return the most recent write to that address → performance may suffer
- Weaker: any write to an address will be seen by a read after some delay (i.e. writes to the same address must be serialized)

## ❑ Snooping-based cache coherence protocols

- Each cache line records its usage states (shared/valid)
- Write operation from a  $\mu P$  will broadcast either the shared write data or an invalidate message to all caches
- Each cache will snoop its cache lines and update their content or states accordingly

cache : brainless , power-consuming  
TCM : need programmer allocate data in TCM.

# MSI Protocol (1/2)

Another is called MESI

- ❑ Each cache line can be in one of three states
  - Modified (M): data is dirty, no other cache has a copy
  - Shared (S): data is clean, other cache may have a copy
  - Invalid (I): data not in cache or invalidated by other controllers
- ❑ Cache line tag augmented with state information
- ❑ Given a pair of caches, the possible states of the same cache line:

Cache 2

	M	S	I
M	x	x	✓
S	x	✓	✓
I	✓	✓	✓

Cache 1

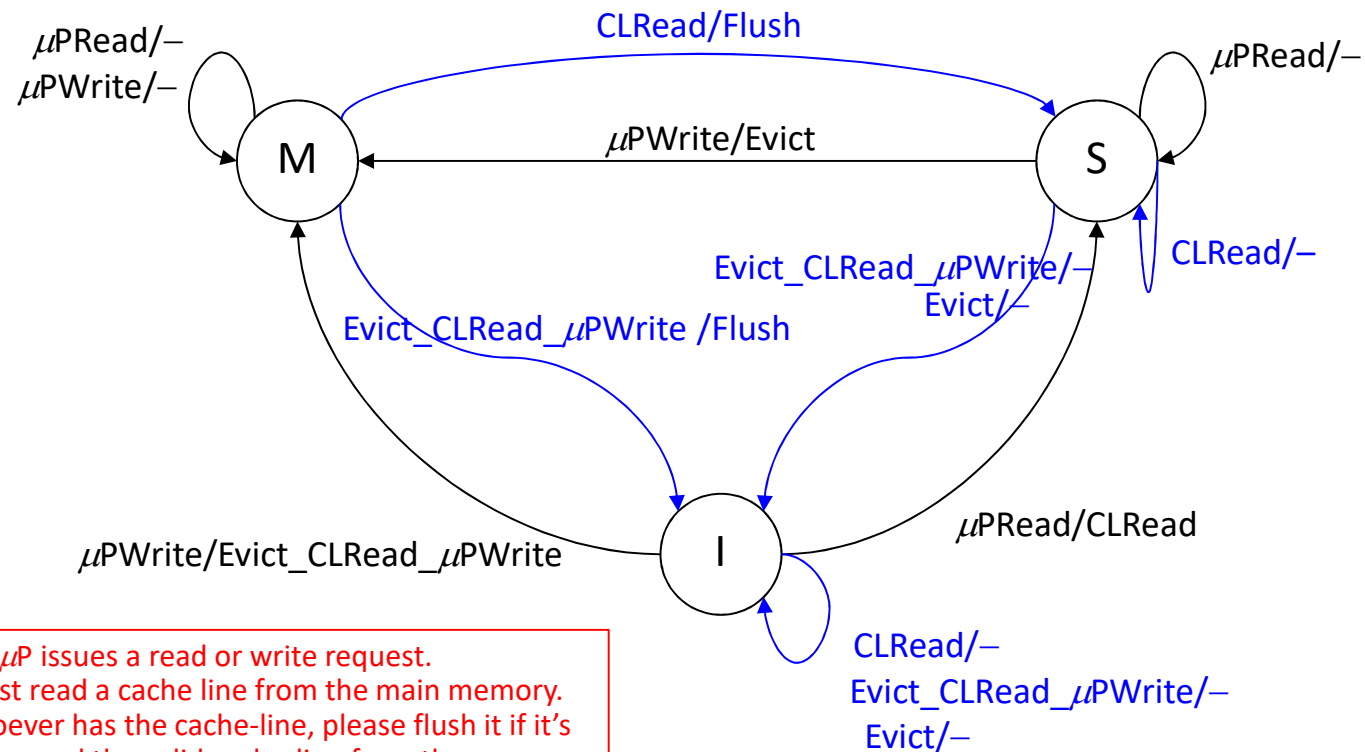
# MSI Protocol (2/2)

---

- ❑ MSI read/write behavior depends on the state of the target cache line
- ❑ For read requests:
  - M- or S-states: the cache supplies the data.
  - I-state: verify that the line is not in the M-state in other cache
- ❑ For write requests:
  - M-state: the cache modifies the data locally
  - S-state: the cache must notify other caches that contain the line in the S-state to evict the line
  - I-state, the cache must notify other caches that contain the line in the S- or M-states to evict the line, before fetching it

# FSM of MSI Protocols

- ❑ The FSM of cache block states:
  - Black requests are sent from the owner  $\mu P$
  - Blue requests are sent from other cache controllers



- $\mu P$ Read/ $\mu P$ Write:  $\mu P$  issues a read or write request.
- CLRead: cache must read a cache line from the main memory.
- Evict\_CLRead: whoever has the cache-line, please flush it if it's dirty so that we can read the valid cache-line from the memory.

# MSEI Protocol

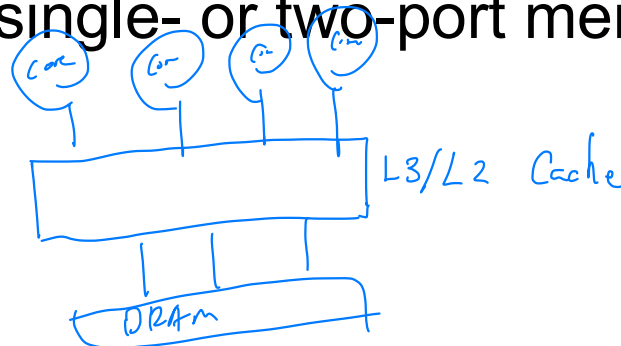
---

- ❑ The MESI is an invalidate-based protocol, and is one of the most common protocols for write-back caches
  - Cache controller invalidates its own copy when it snoops a modification to a cached line
  - Less memory traffic than the MSI protocol
- ❑ Each cache line is in one of four states:
  - Modified (M): data is dirty, no other cache has a copy
  - Exclusive (E): data is clean , no other cache has a copy
  - Shared (S): data is clean, other cache may have a copy
  - Invalid (I): data not in cache

originally  
from E

# Data Sharing thru Multi-Port Memory

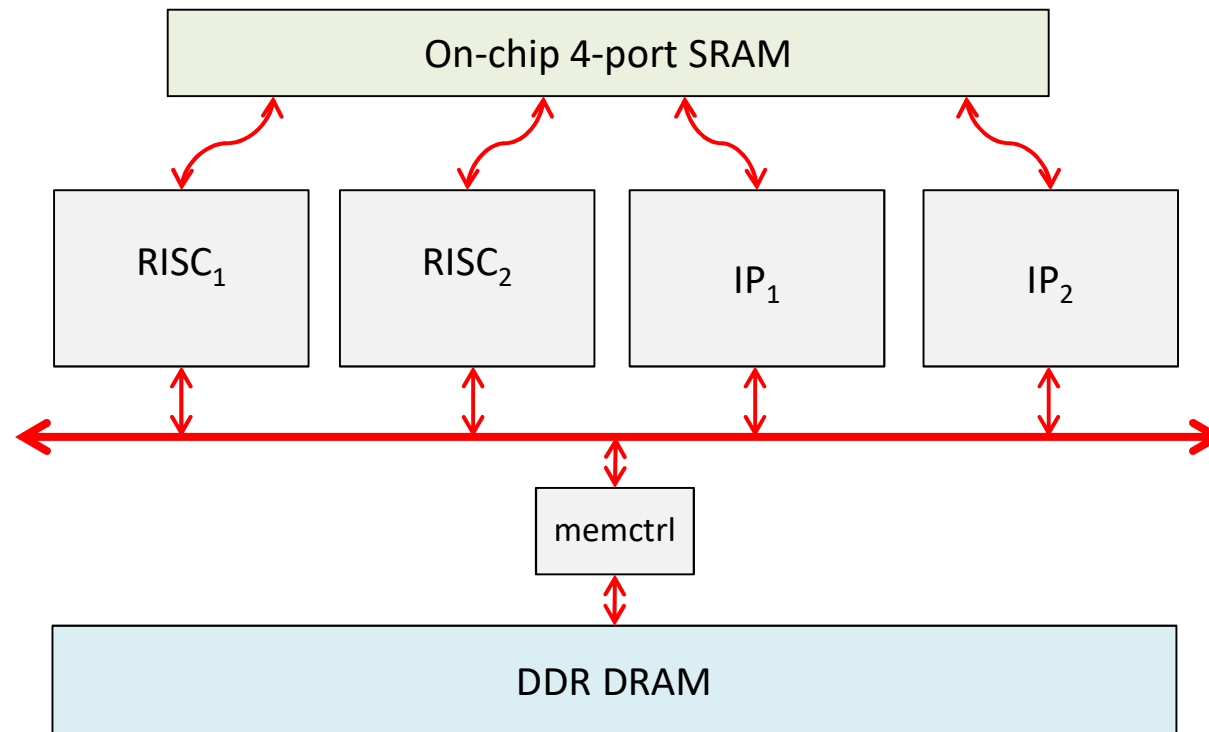
- ❑ A multi-port on-chip memory block for data sharing is more efficient, but a lot more expensive
  - The complexity of the address decoding logic grows as the memory size increases:
    - $n$ -bit decoder requires  $2^n$   $n$ -input AND gates
    - Coincident decoding reduces the complexity, but still expensive
  - Each port needs its own address decoder
  - Data racing results are implementation dependent
- ❑ To reduce cost, a multi-port memory can be synthesized using single- or two-port memory blocks





# Data Sharing with 4-Port OCM

- ❑ A multi-port scratchpad allows concurrent memory operations and data-sharing for multi-thread systems



# Multi-Port Memory for FPGA

- ❑ Synthesis of a multi-port memory requires design trade-off between access delay and logic usage
  - If LUTs are used to directly synthesize a 4-port memory, the memory size cannot be too large
- ❑ BRAMs shall be used for multi-port memory synthesis:
  - You can use some decoding logic to arbitrate four concurrent accesses through the port(s) to a two-port BRAM
  - Alternatively, you can use more advanced techniques to design a multi-port memory that allows true simultaneous memory accesses†

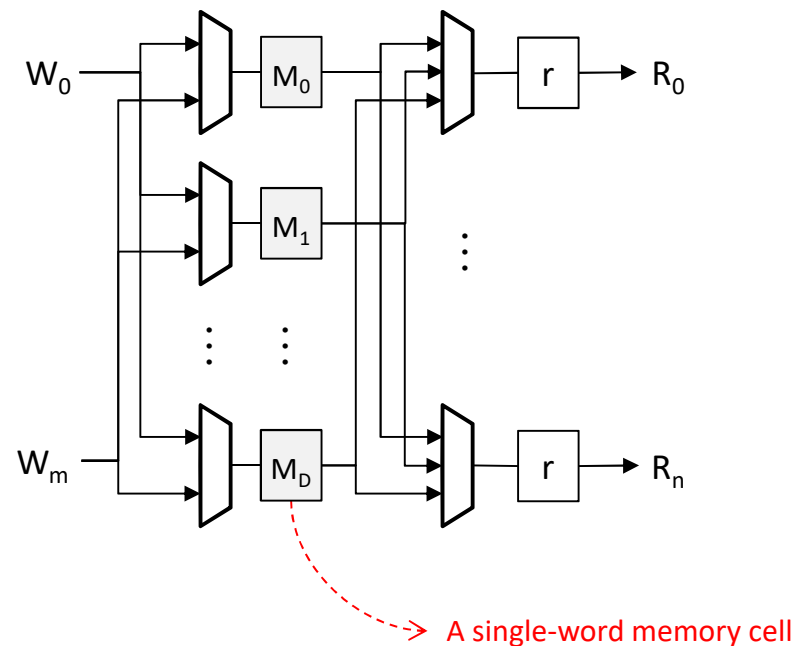
URAM: 2-port, larger in size

---

† C. E. Laforest et al., "Composing Multi-Ported Memories on FPGAs," *ACM Trans. on Reconfigurable Technology and Systems*, Vol. 7, No. 3, Article 16, August 2014.

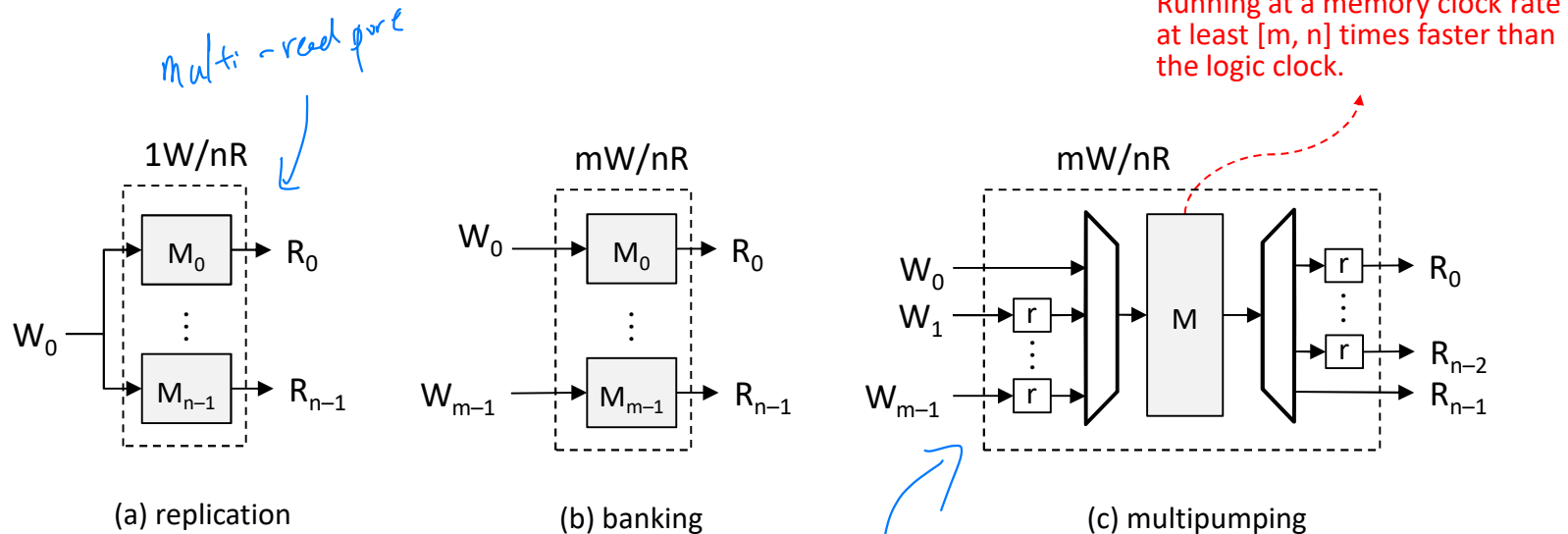
# Direct Multi-Port Memory Synthesis

- A multi-ported memory implemented using logic cells, having  $D$  single-word storage ( $S$ ),  $m$  write ( $W$ ) ports,  $n$  read ( $R$ ) ports, and  $n$  output registers  $r$  is as follows:



# More Ports, Less Logic Cells

- ❑ Three conventional techniques for providing more R/W ports given 1W/1R memory blocks:
  - Note that each memory block in the following diagrams has the same size as the desired multi-port memory



register file structure alike.

memory runs faster, e.g. 100 MHz  
if logic runs at 10 MHz,  
memory can serve 10 req at 1 cycle.

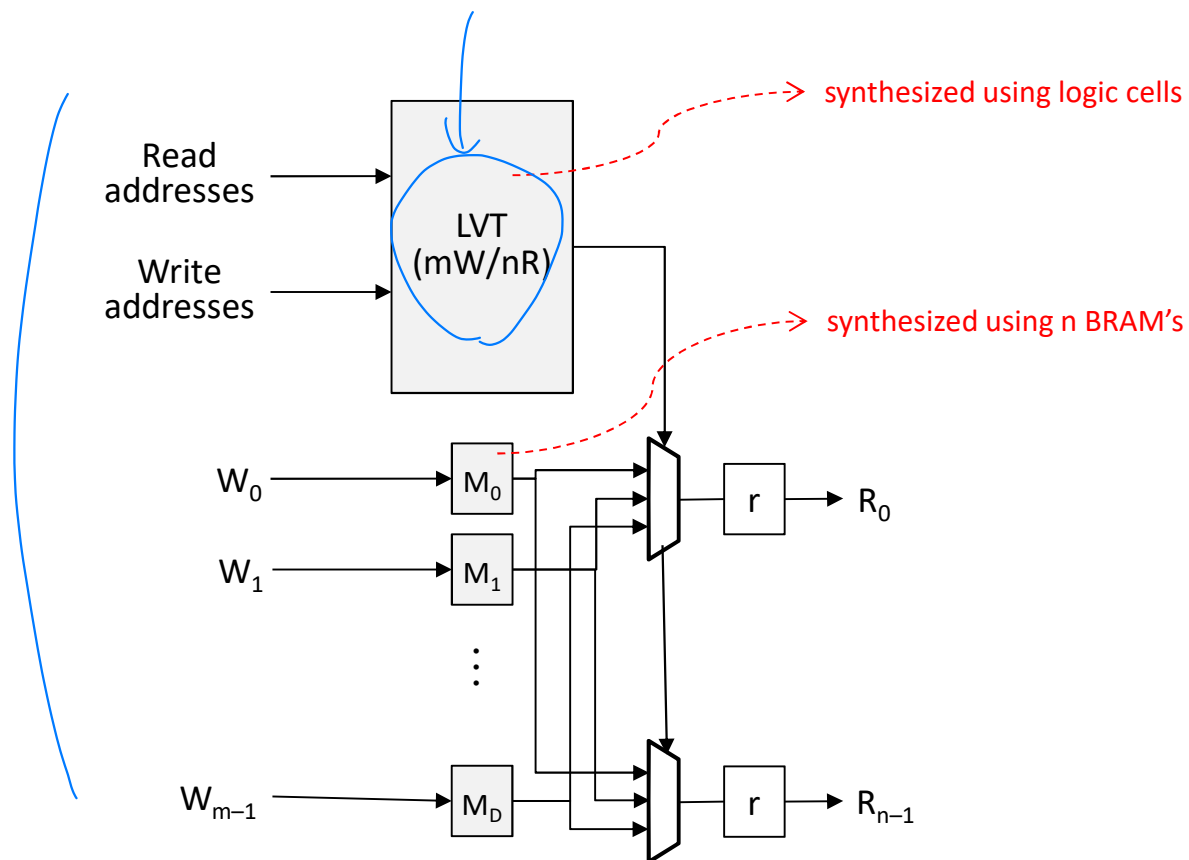
# The Live Value Table (LVT) Approach

---

- ❑ The LVT approach is based on memory banking
- ❑ LVT uses multiplexers and a table to steer reads to the most recently updated bank for each memory address
  - Improves significantly on the area and speed of comparable designs built using only logic cells
  - Can implement multi-ported memories with bidirectional ports

# LVT-based mW/nR Memory Design

- LVT records the most-recent write bank for each write addresses:



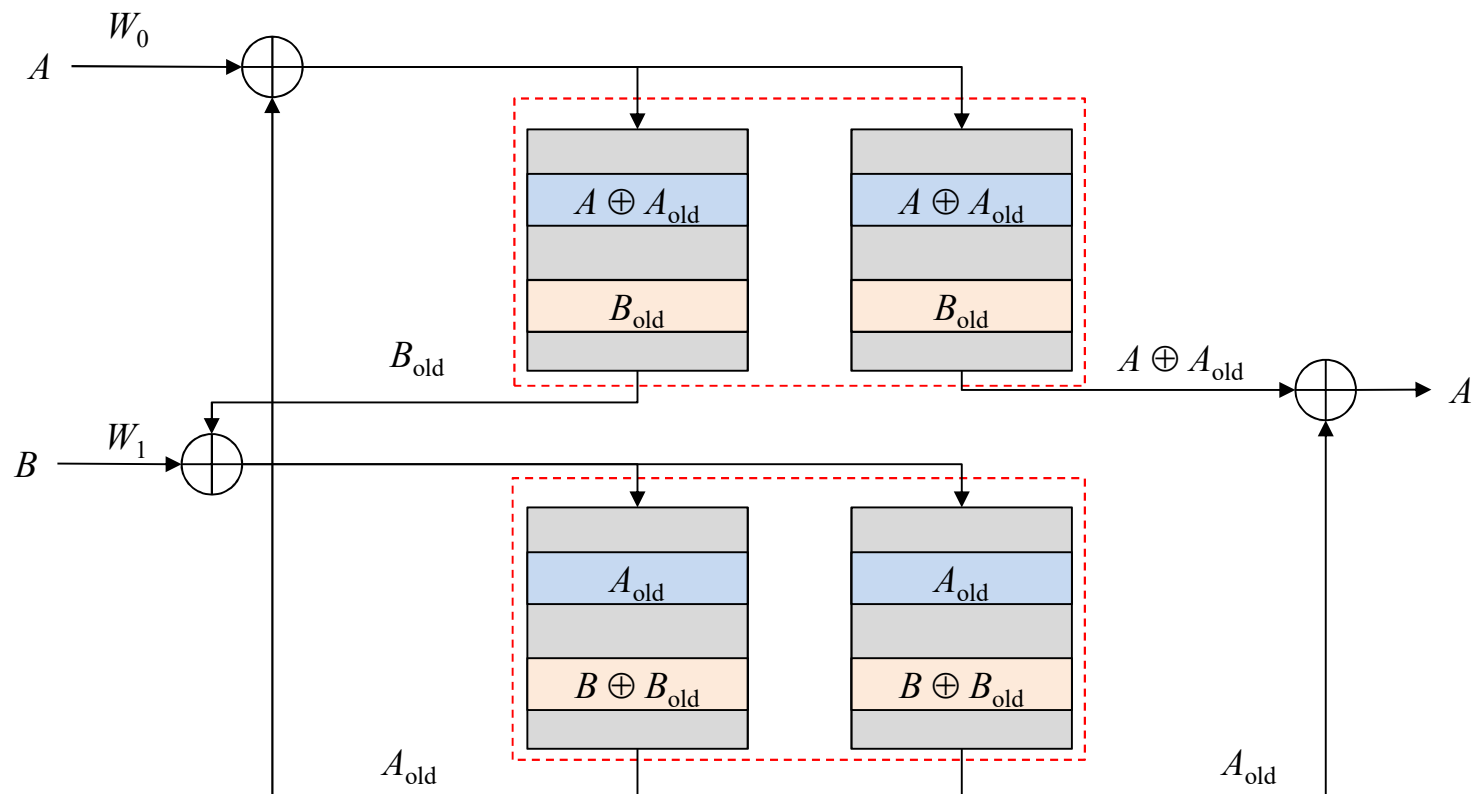
# The XOR-Based Approach

---

- ❑ The XOR approach is also based on memory banking
  - Using the property of  $\oplus$  operation that  $(A \oplus B) \oplus B = A$
- ❑ The design removes the need for a Live Value Table and thus avoids output multiplexing → usually uses less logic but require more BRAMs
- ❑ Under some configurations, the XOR design is faster and consumes less total area than the LVT designs

# XOR-based 2W/1R Design

- Constructing a 2W/1R memory using 1W/1R memory:
  - The row # matches the # of write ports
  - The column # matches  $\#W-1$  plus  $\#R$





# XOR-based mW/nR Design

---

- ❑ For mW/nR XOR-based multiport memory, we need:
  - A 2D BRAM array of  $m \times (m-1)$ , each row of BRAMs feeds old values to the other  $m-1$  write ports
  - For each read port, we need an extra 1-D BRAMs array of  $m \times 1$ , which total to a 2D read array of  $m \times n$
- ❑ There is no need for multiplexors, only BRAM blocks and m-input XOR logic gates

# Discussions

---

- ❑ Today, the bottleneck of high performance processors are often in the memory subsystem
  - Fast local memory can improve performance at low device cost, but programming effort can be high
  - Cache memory are easy for programmers, but suffers high overhead as the # of threads increases
  
- ❑ We pretty much ran out of ideas for general-purpose design tricks of memory subsystem
  - Application-specific optimization is the key to faster systems