

# Embedded Processor



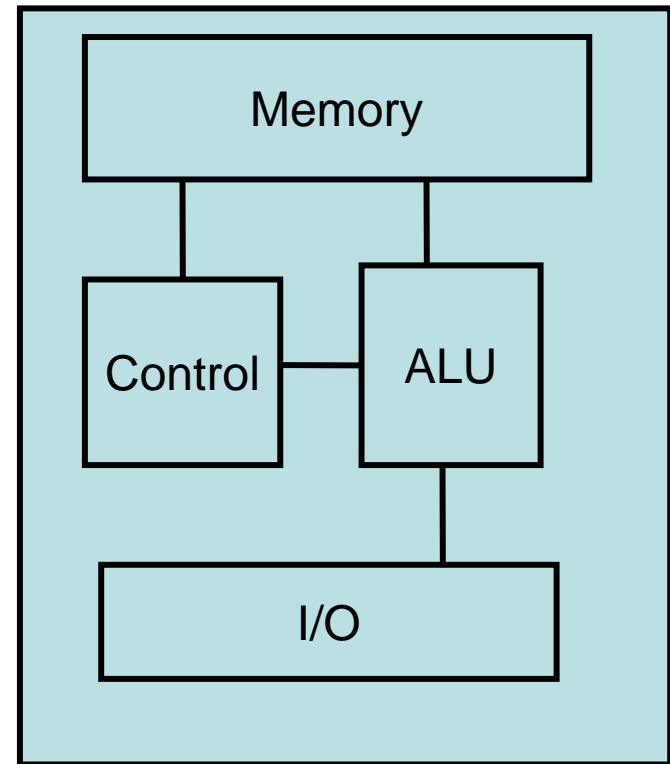
# Agenda

- ❖ CPU architecture
- ❖ Pipeline
- ❖ Cache
- ❖ Memory Management Units (MMU)
- ❖ Direct Memory Access (DMA)
- ❖ Bus Interface Unit (BIU)
- ❖ Examples

# CPU Architecture (1/8)

## ❖ von Neumann architecture

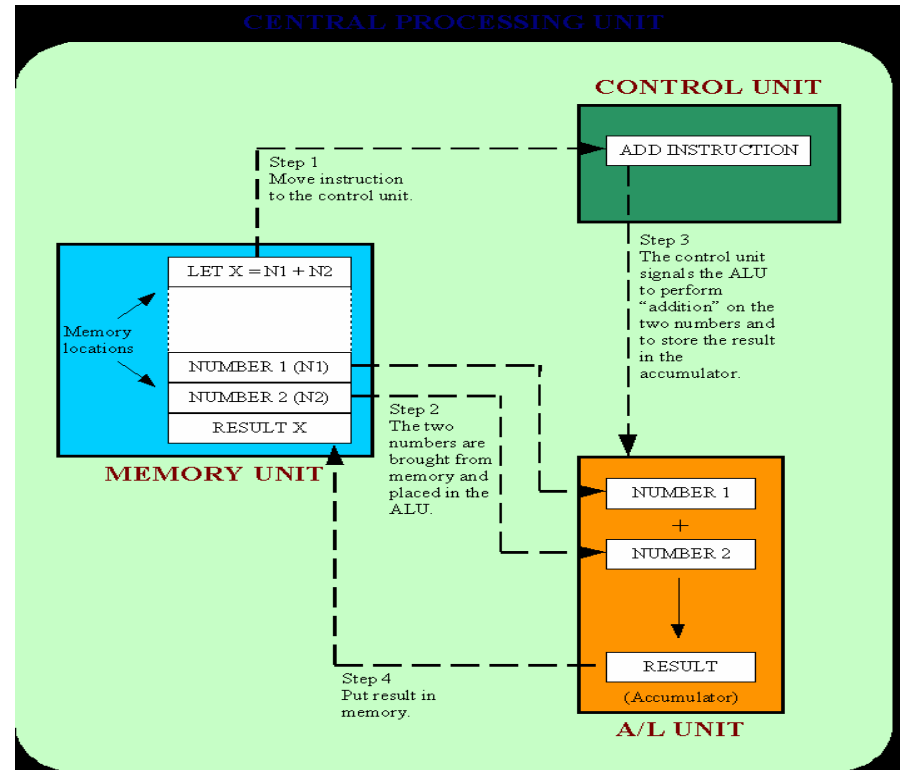
- Data and instructions are stored in memory, the Control Unit takes instructions and controls data manipulation in the Arithmetic Logic Unit.
- Input/Output is needed to make the machine a practicality
- Execution in multiple cycles
- Serial fetch instructions & data
- Single memory structure
  - Can get data/program mixed
  - Data/instructions same size



Examples of von Neumann  
ORDVAC (U-Illinois) at  
Aberdeen Proving Ground,  
Maryland (completed Nov  
1951)  
IAS machine at Princeton  
University (Jan 1952)

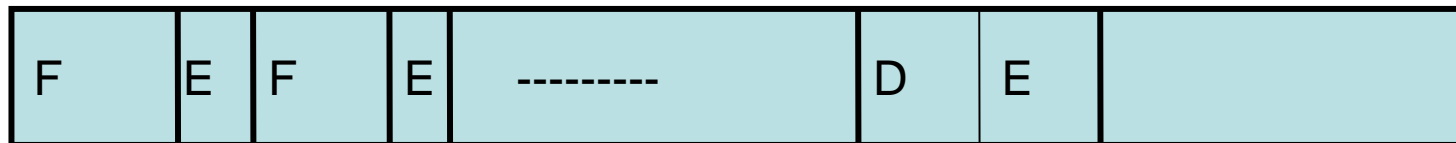
# CPU Architecture (2/8)

- ❖ The ALU manipulates two binary words according to the instruction decoded in the Control unit. The result is in the Accumulator and may be moved into Memory.



# CPU Architecture (3/8)

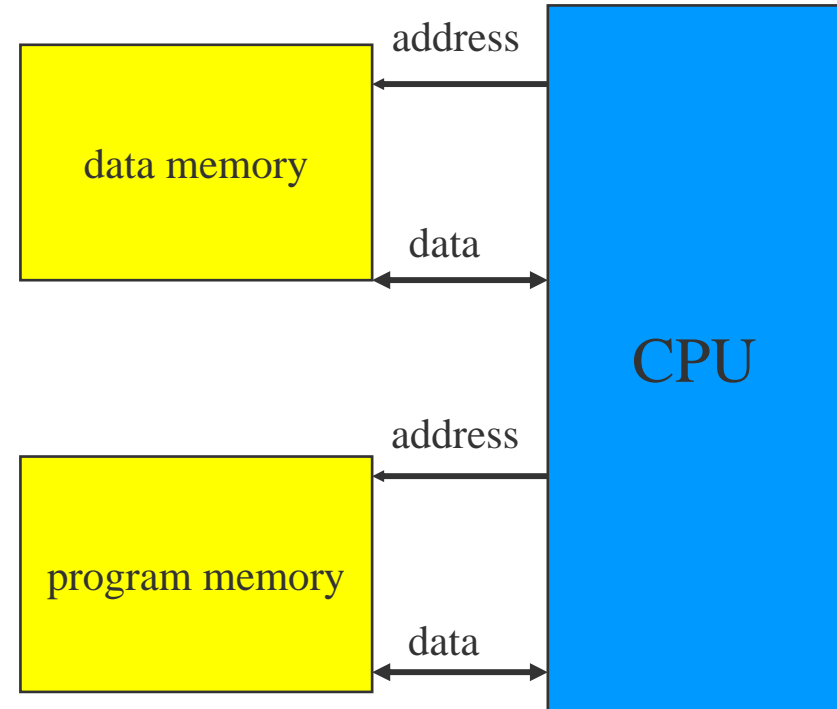
- ❖ von Neumann architecture introduces a problem, which is not solved until much later.
- ❖ The CPU work on only one instruction at a time, each must be fetched from memory then executed. During fetch the CPU is idle, this is a waste of time. Made worse by slow memory technology compared with CPU.
- ❖ Time is also lost in the CPU during instruction decode.



# CPU Architecture (4/8)

## ❖ Harvard architecture

- In a computer using the Harvard architecture, the CPU can both read an instruction and perform a data memory access at the same time.
- A Harvard architecture computer can thus be faster for a given circuit complexity because instruction fetches and data access do not contend for a single memory pathway.
- Execution in 1 cycle
- Parallel fetch instructions & data
- More Complex hardware
  - Instructions and data always separate
  - Different code/data path widths (E.G. 14 bit instructions, 8 bit data)



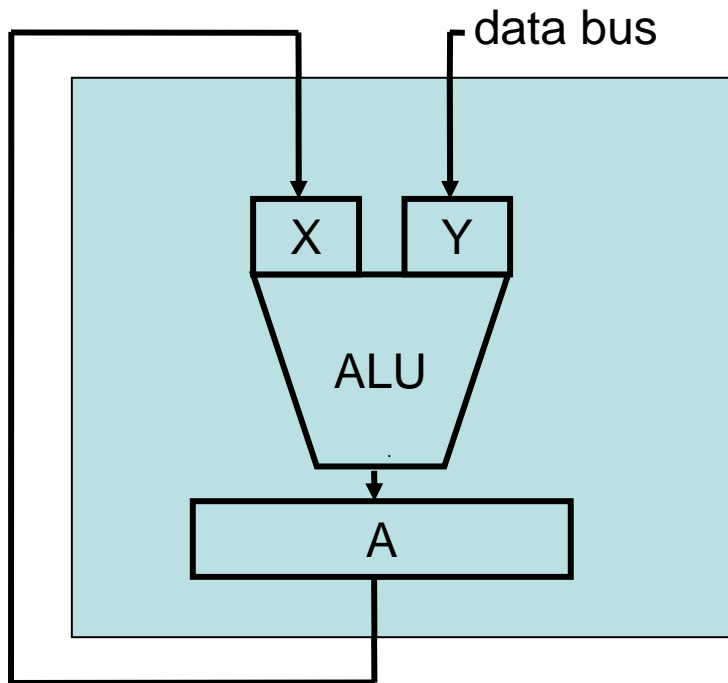
Examples of Harvard:

Microchip PIC families  
Atmel AVR  
AndeScore  
Atom

# CPU Architecture (5/8)

- ❖ Take a closer look at the action part of the CPU, the ALU, and how data circulates around it.
- ❖ An ALU is combinational logic only, no data is stored, i.e. no registers. This is the original reason for having CPU registers.

# CPU Architecture (6/8)



In the simplest, minimum hardware, solution one of them, say X, is the accumulator A, the other, Y, is straight off the memory bus (this requires a temporary register not visible to the programmer).

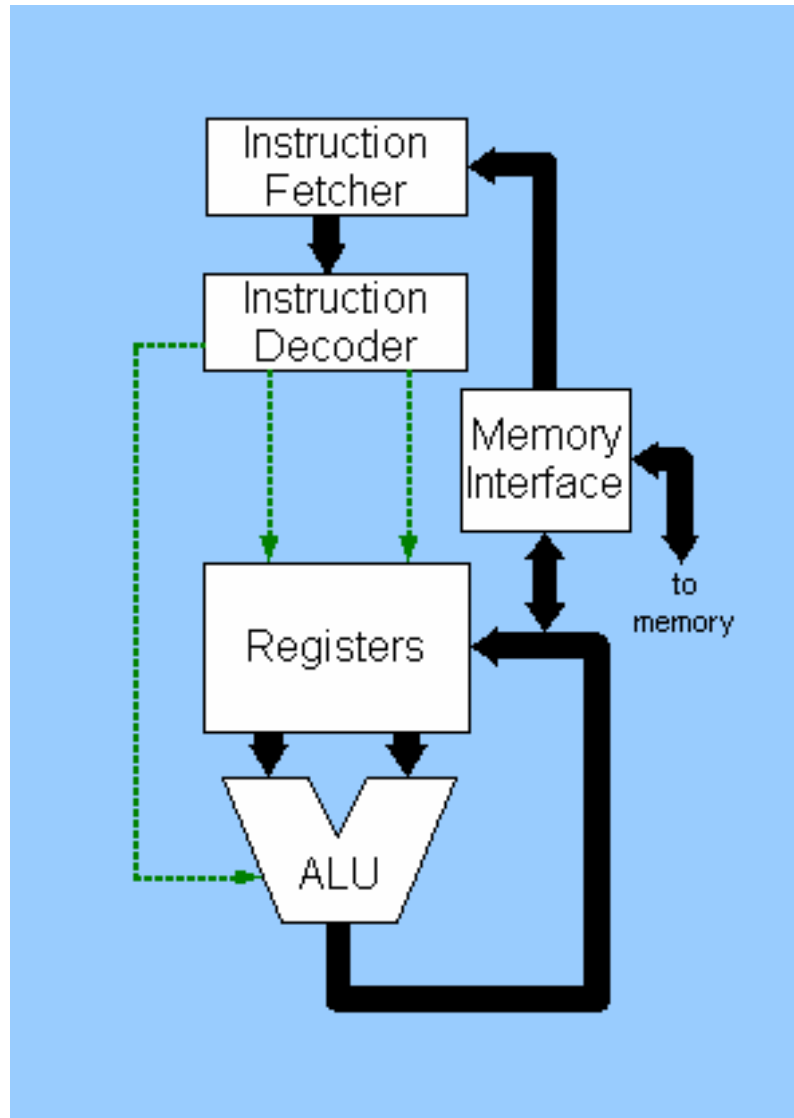
The instruction may be ADDA, which means: add to the contents of A the number (Y) and put the answer in A.



# CPU Architecture (7/8)

- It's a simple step to add more CPU data registers and extend the instructions to include B, C,..... as well as A.
- An internal CPU bus structure then becomes a necessity.

# CPU Architecture (8/8)



# Architectures: CISC vs. RISC (1/2)

## ❖ CISC - Complex Instruction Set Computers:

- von Neumann architecture
- Emphasis on hardware
- Includes multi-clock complex instructions
- Memory-to-memory
- Sophisticated arithmetic (multiply, divide, trigonometry etc.)
- Special instructions are added to optimize performance with particular compilers

# Architectures: CISC vs. RISC (2/2)

## ❖ RISC - Reduced Instruction Set Computers:

- Harvard architecture
- A very small set of primitive instructions
- Fixed instruction format
- Emphasis on software
- All instructions execute in one cycle (Fast!)
- Register to register (except Load/Store instructions)
- Pipeline architecture

# Single-, Dual-, Multi-, Many- Cores

## ❖ Single-core:

- Most popular today.

## ❖ Dual-core, multi-core, many-core:

- Forms of multiprocessors in a single chip

## ❖ Small-scale multiprocessors (2-4 cores):

- Utilize task-level parallelism.
- Task example: audio decode, video decode, display control, network packet handling.

## ❖ Large-scale multiprocessors (>32 cores):

- nVidia's graphics chip: >128 core
- Sun's server chips: 64 threads

# Pipeline (1/2)

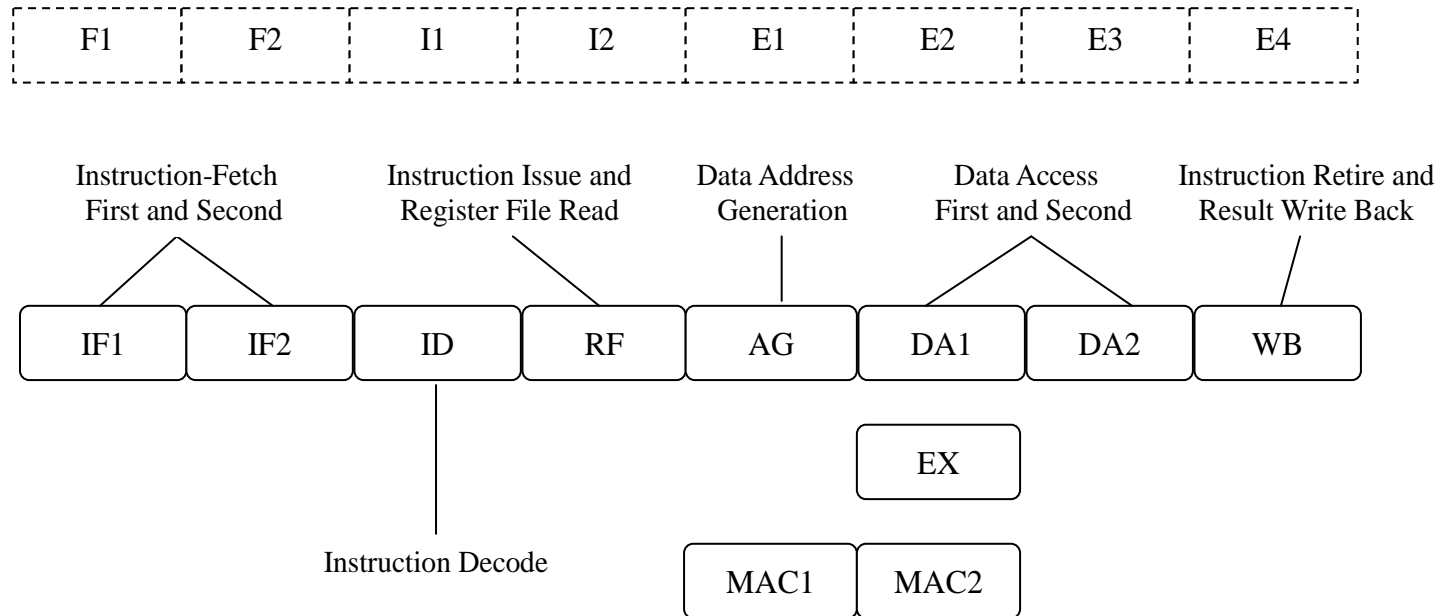
- ❖ An **instruction pipeline** is a technique used in the design of computers and other digital electronic devices to increase their instruction throughput (the number of instructions that can be executed in a unit of time).
- ❖ The fundamental idea is to split the processing of a computer instruction into a series of independent steps, with storage at the end of each step.
- ❖ This allows the computer's control circuitry to issue instructions at the processing rate of the slowest step, which is much faster than the time needed to perform all steps at once.
- ❖ The term pipeline refers to the fact that each step is carrying data at once (like water), and each step is connected to the next (like the links of a pipe.)

# Pipeline (2/2)

Instr. No.	Pipeline Stage						
	IF	ID	EX	MEM	WB		
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

- ❖ Basic five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back). In the fourth clock cycle (the green column), the earliest instruction is in MEM stage, and the latest instruction has not yet entered the pipeline.

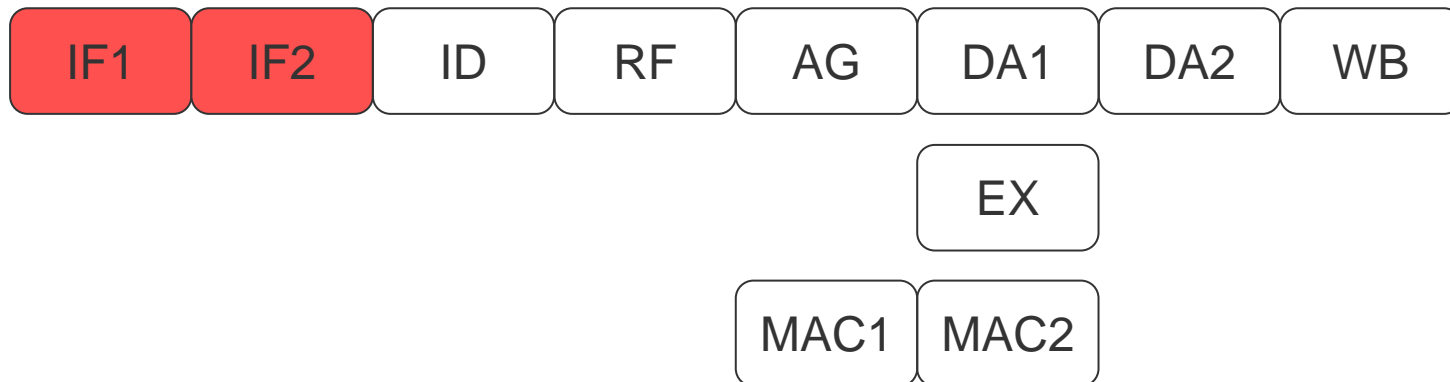
# 8-stage pipeline





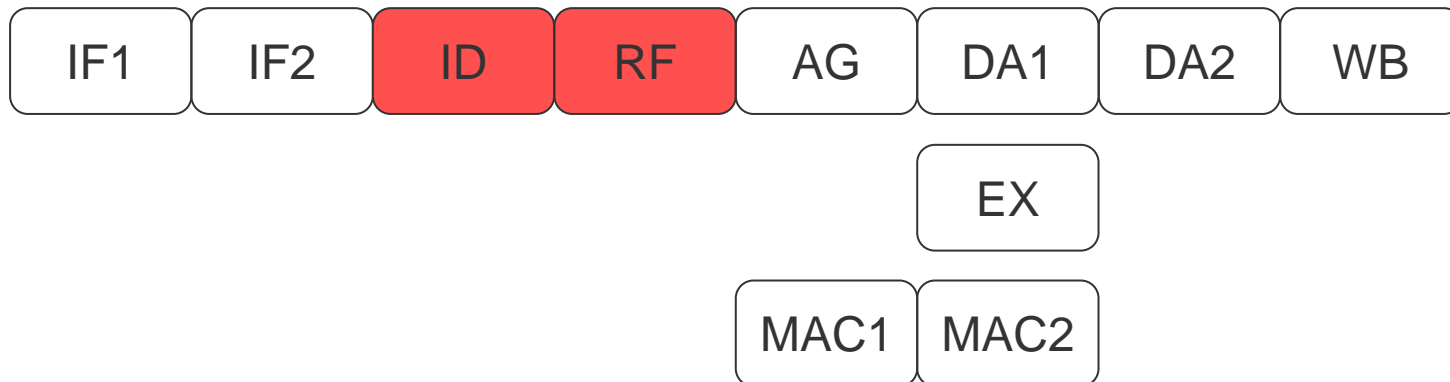
# Instruction Fetch Stage

- ❖ F1 – Instruction Fetch First
  - Instruction Tag/Data Arrays
  - ITLB Address Translation
  - Branch Target Buffer Prediction
- ❖ F2 – Instruction Fetch Second
  - Instruction Cache Hit Detection
  - Cache Way Selection
  - Instruction Alignment



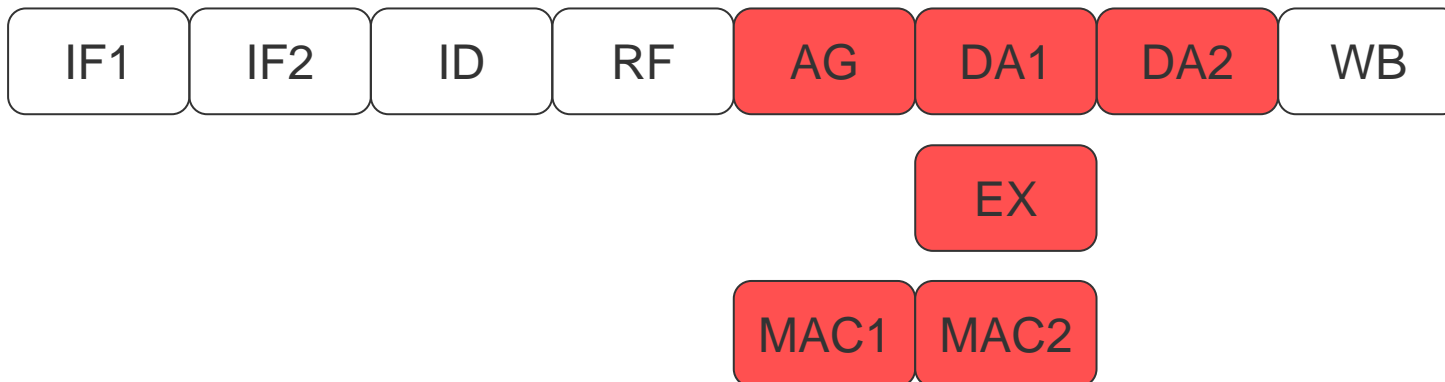
# Instruction Issue Stage

- ❖ I1 – Instruction Issue First / Instruction Decode
  - 32/16-Bit Instruction Decode
  - Return Address Stack prediction
- ❖ I2 – Instruction Issue Second / Register File Access
  - Instruction Issue Logic
  - Register File Access



# Execution Stage

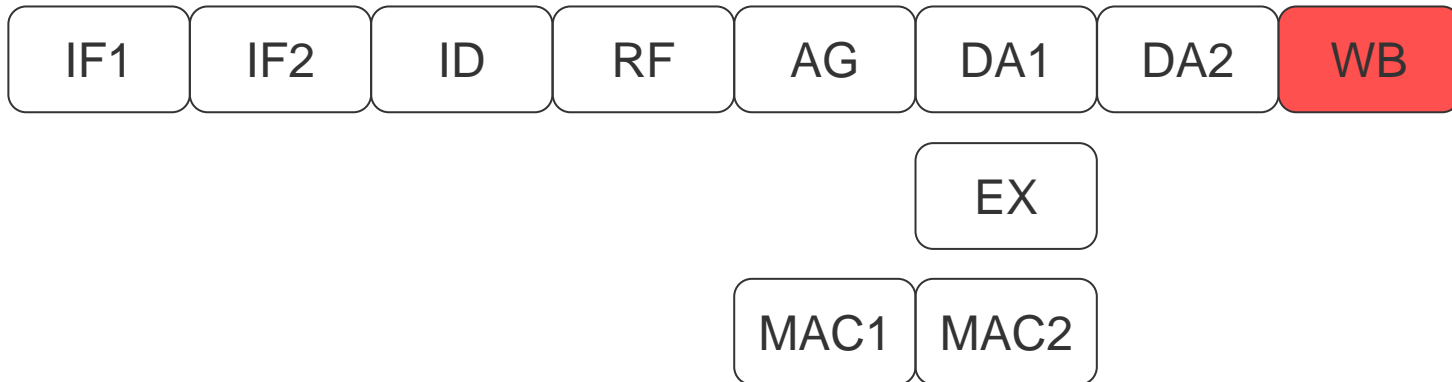
- ❖ **E1 – Instruction Execute First / Address Generation / MAC First**
  - Data Access Address Generation
  - Multiply Operation (if MAC presents)
- ❖ **E2 –Instruction Execute Second / Data Access First / MAC Second / ALU Execute**
  - ALU
  - Branch/Jump/Return Resolution
  - Data Tag/Data arrays
  - DTLB address translation
  - Accumulation Operation (if MAC presents)
- ❖ **E3 –Instruction Execute Third / Data Access Second**
  - Data Cache Hit Detection
  - Cache Way Selection
  - Data Alignment



# Write Back Stage

## ❖ E4 –Instruction Execute Fourth / Write Back

- Interruption Resolution
- Instruction Retire
- Register File Write Back



# Branch Prediction Overview

- ❖ Why is branch prediction required?
  - A deep pipeline is required for high speed
  - Branch cost is a combination of penalty, frequency of branches, and frequency that they are taken
  - Moving detection earlier reduces penalty
  - Conditional operations reduce number of branches
  - Pipelines offer speedup due to parallel execution of instructions
  - Full speedup is rarely obtained due to hazards (data, control, structural)
  - Control hazards handled by avoidance and prediction
  - Prediction can be static, dynamic, local, global, and hybrid
  - Prediction reduces number "taken"
- ❖ Why *dynamic* branch prediction?
  - Static branch prediction
    - Static uses fixed prediction algorithm, or compile-time information
  - Dynamic branch prediction
    - Dynamic gathers information as the program executes, using historical behavior of branch to predict its next execution

# Branch Prediction Overview

## ❖ Simple Static Prediction:

- if 60% of branches are taken, assume all are taken, and the percent "taken" drops to 40%
- Once a choice is made, the compiler can order branches appropriately
- Very simple, cheap to implement, moderately effective

## ❖ Complex Static Prediction:

- Assume backwards branches are taken (likely loop returns) and forward branches aren't (BTFN)
- Compiler provides hints by selecting different branches or including flags to indicate likely to be taken -- requires a predecoding of the branch

# Branch Prediction Overview

## ❖ Simple Dynamic Prediction

- Records portion of jump address, and most recent action (taken/not)
- Partial address can result in aliasing (behavior of multiple branches is combined)
- Single bit of state can lead to worse behavior in degenerate cases

## ❖ Extended Dynamic Prediction

- 2-bit state -- none, taken, not, both
- 2-bit state -- taken, multi-taken, not-taken, multi-not-taken
- Not fooled by branch occasionally behaving in the opposite manner
- May be associated with I-cache, with BTB, or as a separate table

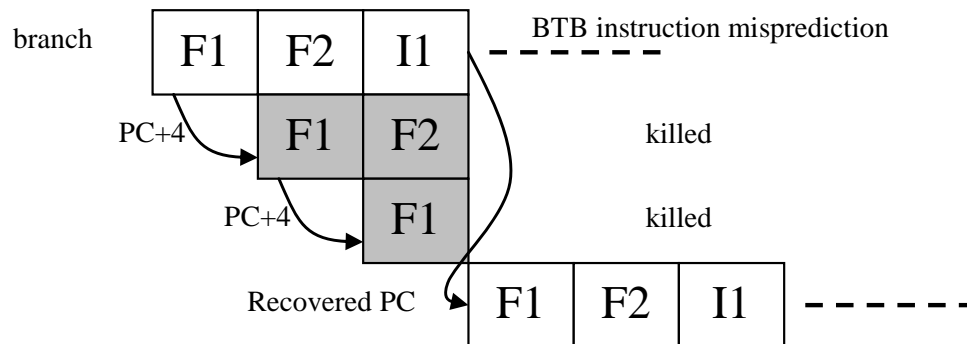
# Branch Prediction Unit

- ❖ Branch Target Buffer (BTB)
  - Branch target buffer (BTB) stores address of the last target location to avoid recomputing
  - Indexed by low-order bits of jump address, tag holds high-order bits
  - Enables earlier prefetching of target based on prediction
  - 128 entries of 2-bit saturating counters
  - 128 entries, 32-bit predicted PC and 26-bit address tag
- ❖ Return Address Stack (RAS)
  - Four entries
- ❖ BTB and RAS updated by committing branches/jumps



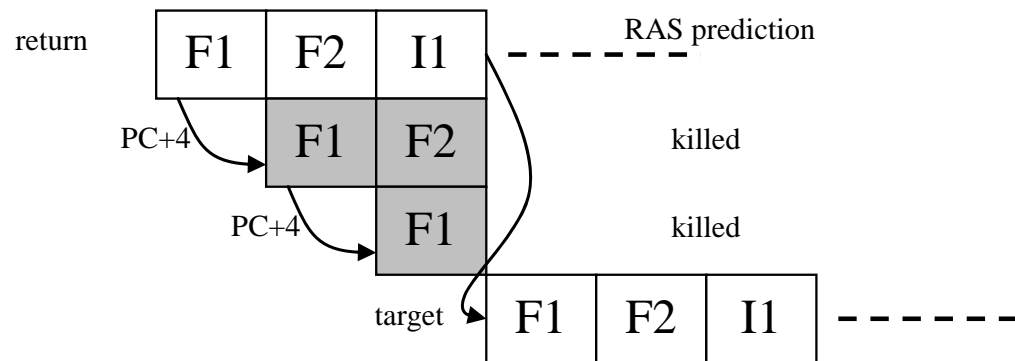
# BTB Instruction Prediction

- ❖ BTB predictions are performed based on the previous PC instead of the actual instruction decoding information, BTB may make the following two mistakes
  - Wrongly predicts the non-branch/jump instructions as branch/jump instructions
  - Wrongly predicts the instruction boundary (32-bit -> 16-bit)
- ❖ If these cases are detected, IFU will trigger a BTB instruction misprediction in the I1 stage and re-start the program sequence from the recovered PC. There will be a 2-cycle penalty introduced here



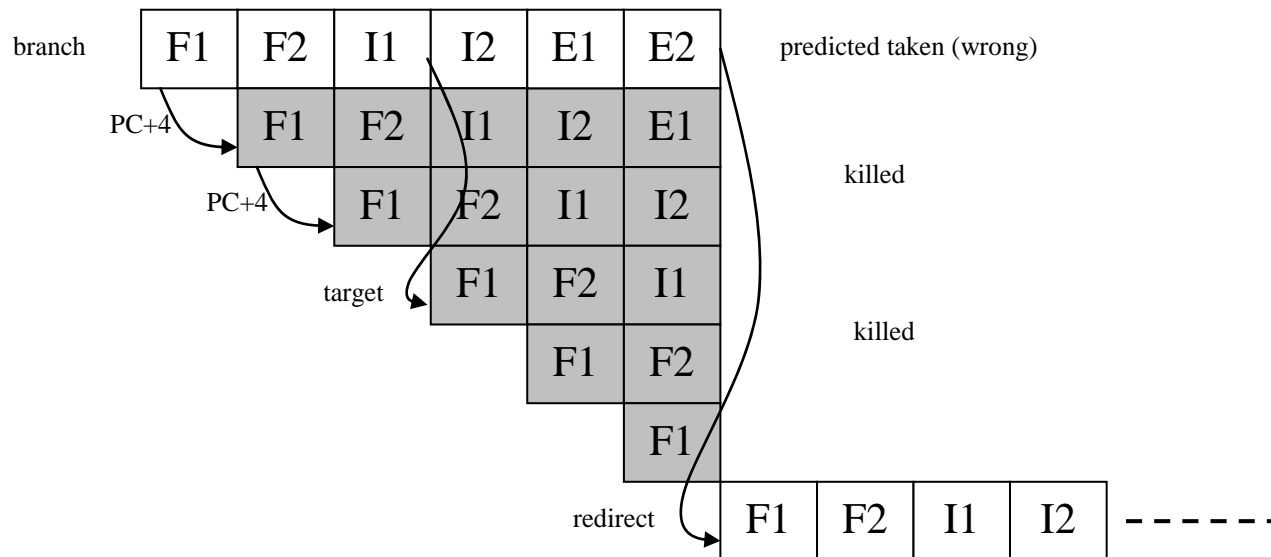
# RAS Prediction

- ❖ When return instructions present in the instruction sequence, RAS predictions are performed and the fetch sequence is changed to the predicted PC.
- ❖ Since the RAS prediction is performed in the I1 stage. There will be a 2-cycle penalty in the case of return instructions since the sequential fetches in between will not be used.



# Branch Miss-Prediction

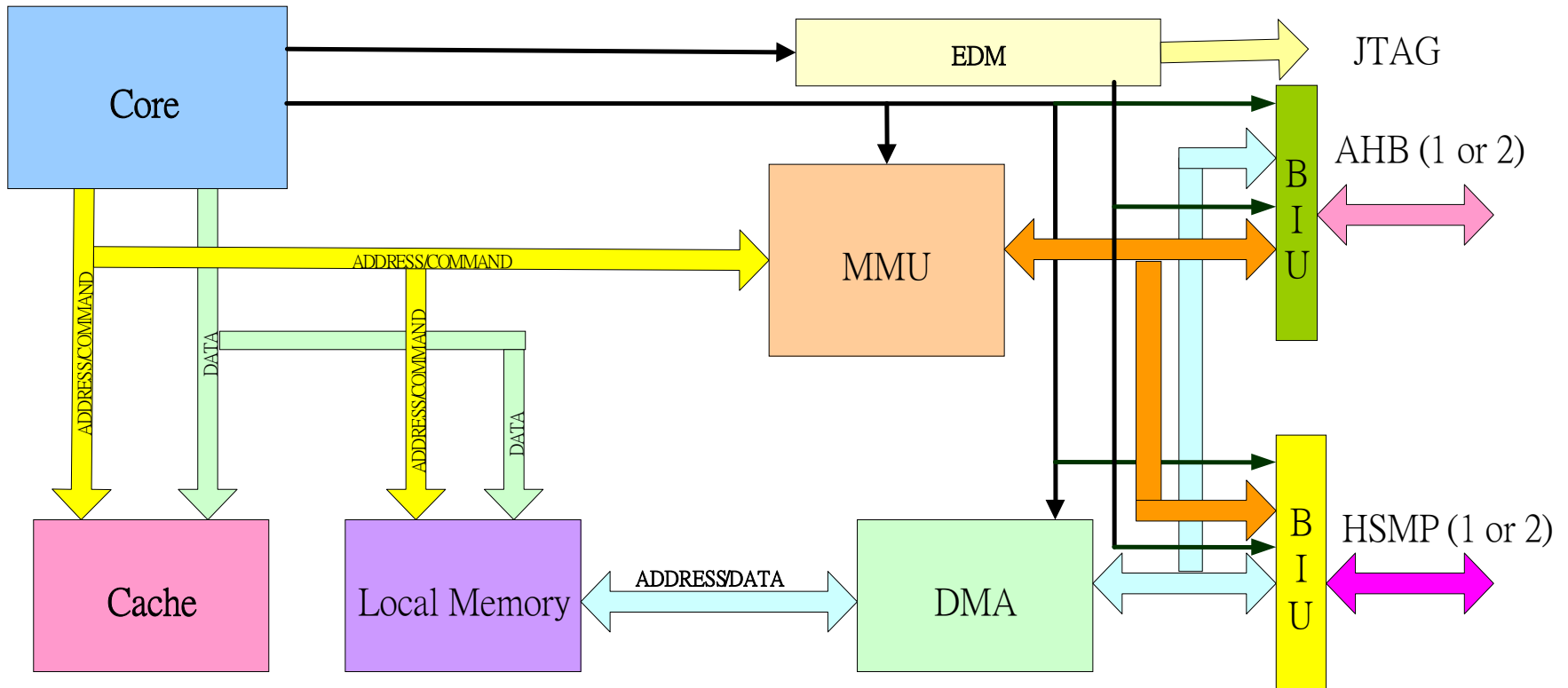
- ❖ In processor core, the resolution of the branch/return instructions is performed by the ALU in the E2 stage and will be used by the IFU in the next (F1) stage. In this case, the misprediction penalty will be 5 cycles.



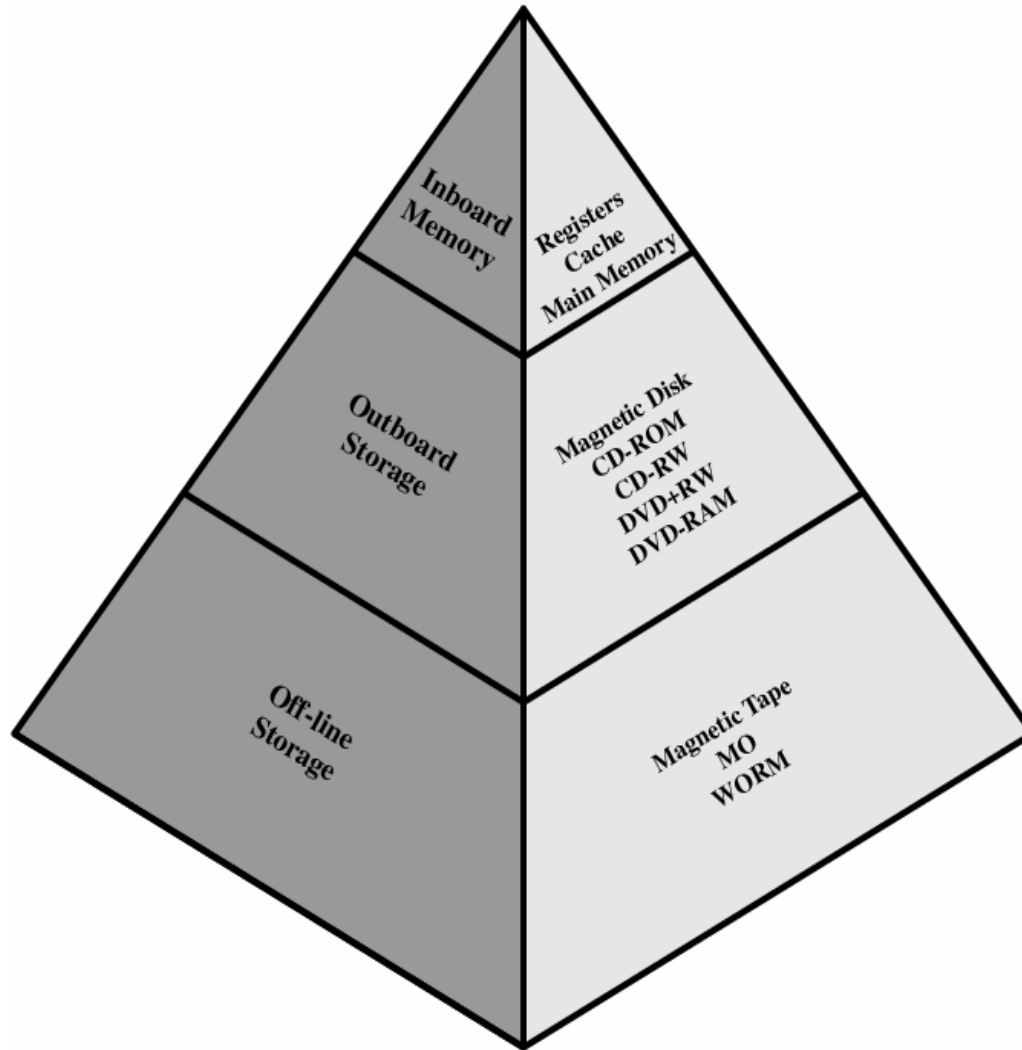
# Cache

- ❖ Store copies of data at places that can be accessed more quickly than accessing the original
  - Speed up access to frequently used data
  - At a cost: Slows down the infrequently used data

# Block diagram

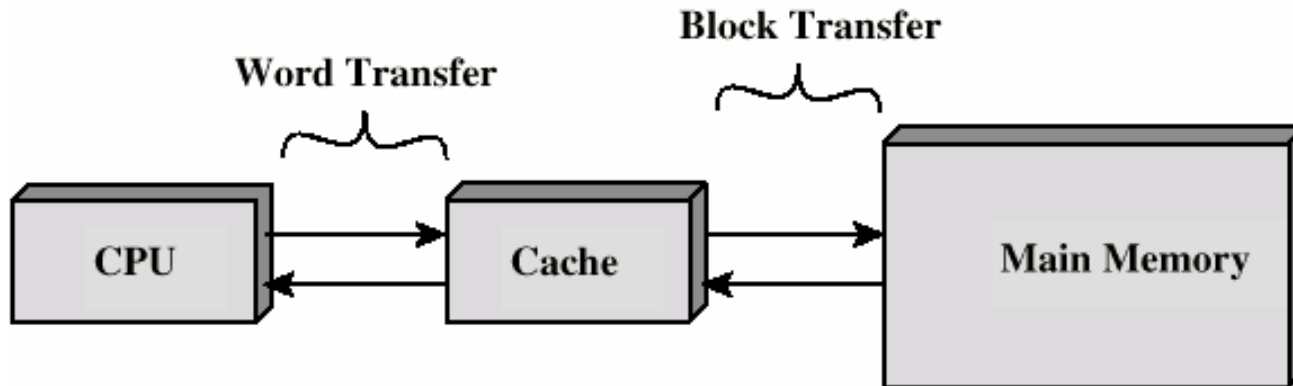


# Memory Hierarchy - Diagram



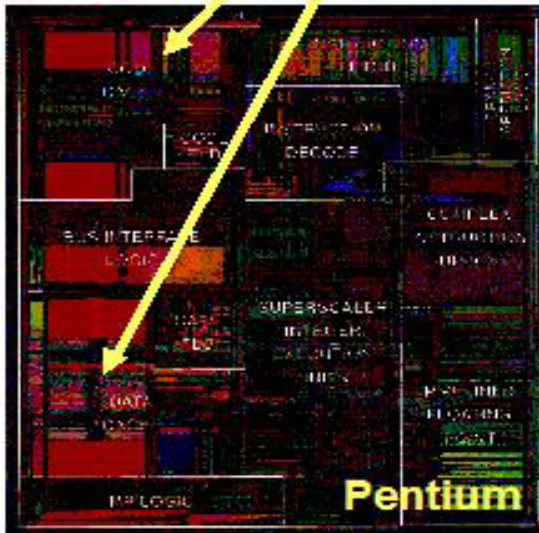
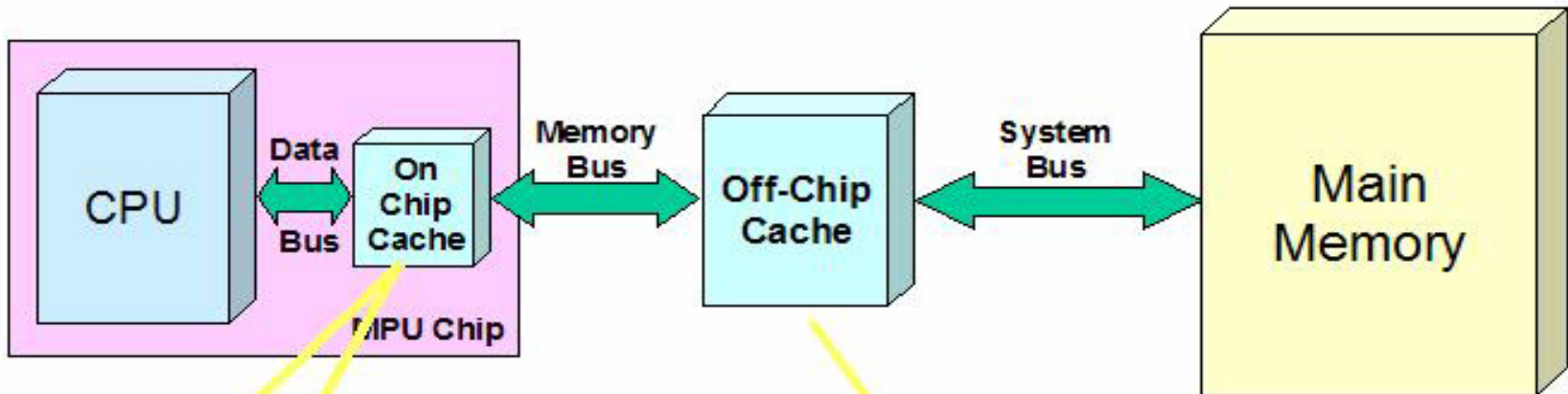
# Cache (1/3)

- ❖ Small amount of fast memory
- ❖ Sits between normal main memory and CPU
- ❖ May be located on CPU chip or module
- ❖ Size does matter
  - Cost
    - More cache is expensive
  - Speed
    - More cache is faster (up to a point)
    - Checking cache for data takes time

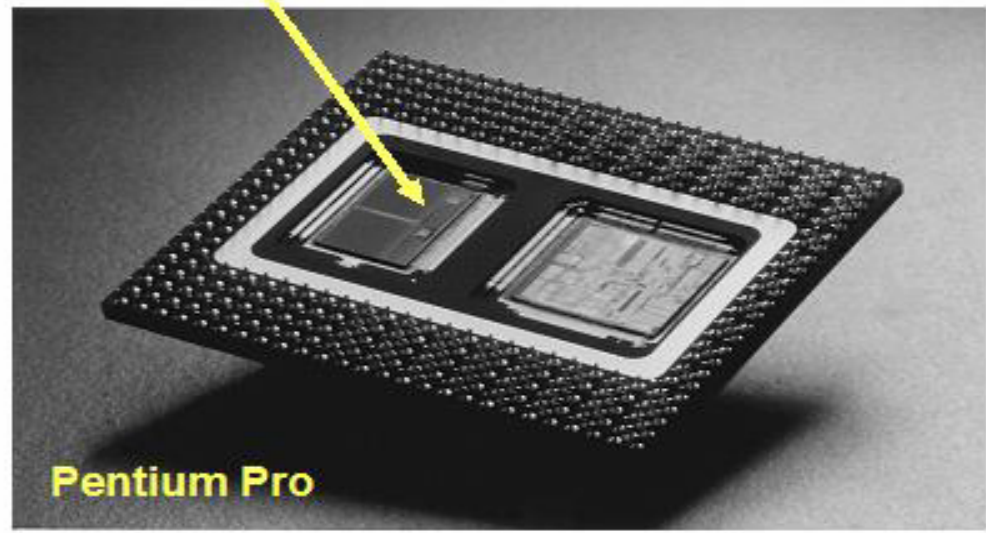


# Cache (2/3)

## Cache Levels



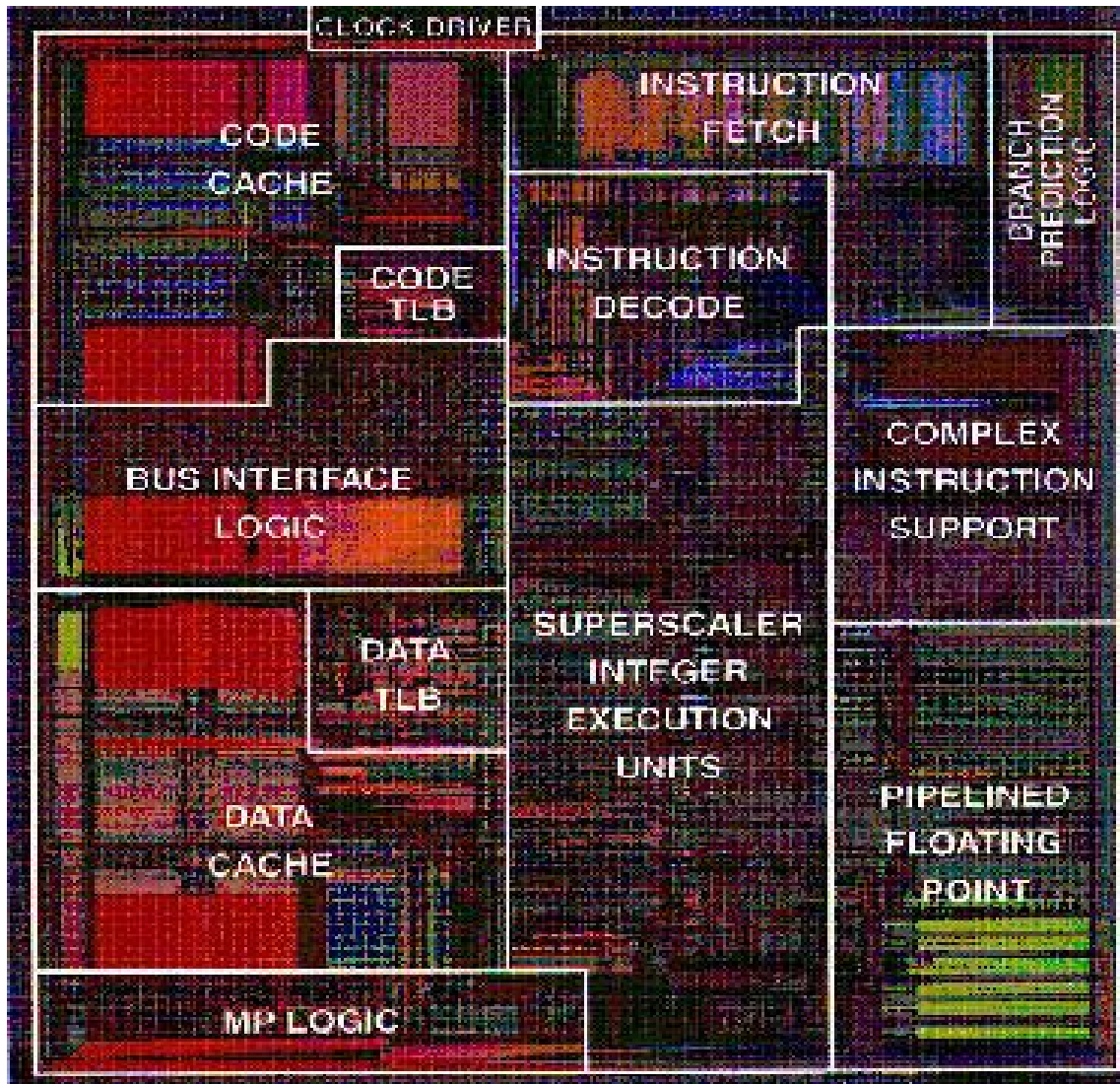
**Pentium**



**Pentium Pro**



# Cache (3/3)



# Hierarchy List

- ❖ Registers
- ❖ L1 Cache
- ❖ L2 Cache
- ❖ Main memory
- ❖ Disk cache
- ❖ Disk
- ❖ Optical
- ❖ Tape

# Caching in Memory Hierarchy

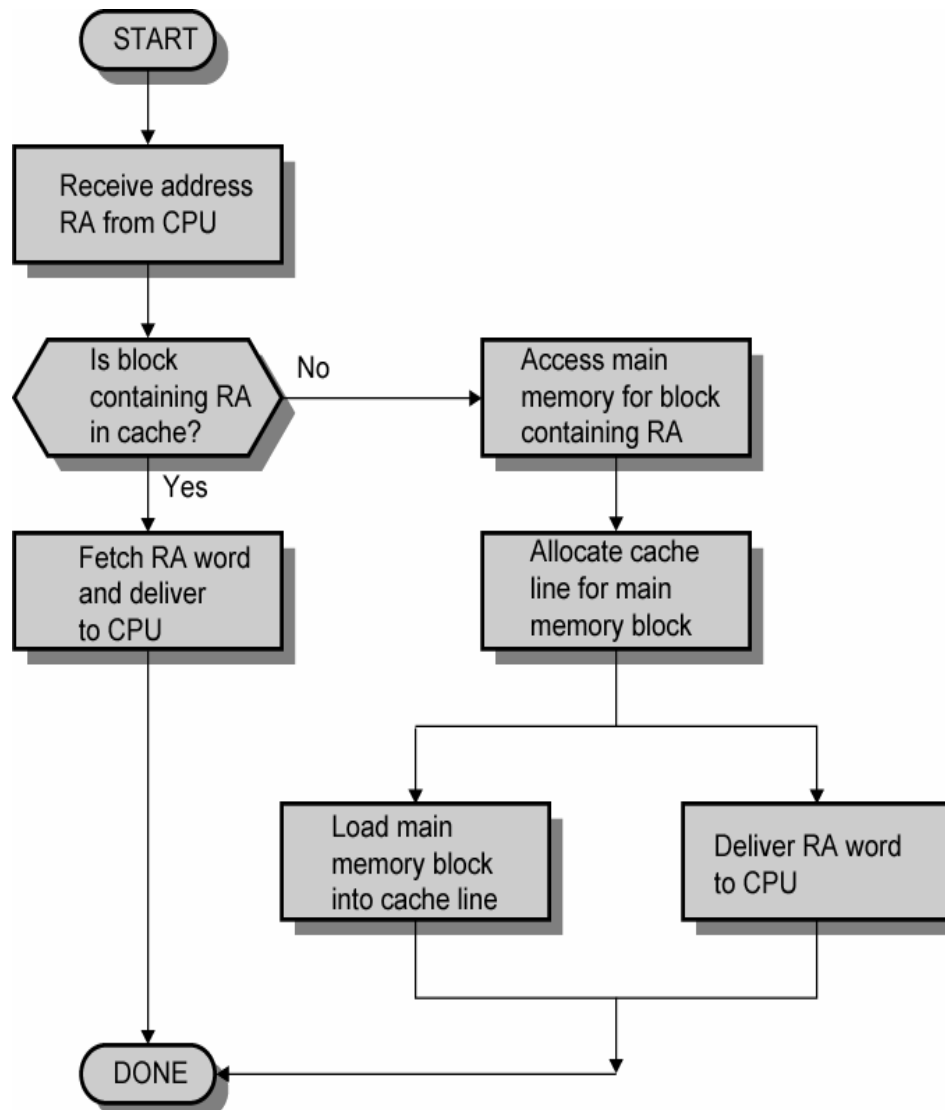
- ❖ Provides the illusion of GB storage
  - With register access time

		Access Time	Size	Cost
Primary memory	Registers	1 clock cycle	~500 bytes	On chip
	Cache	1-2 clock cycles	<10 MB	
	Main memory	1-4 clock cycles	< 4GB	\$0.1/MB
Secondary memory	Disk	5-50 msec	< 200 GB	\$0.001/MB

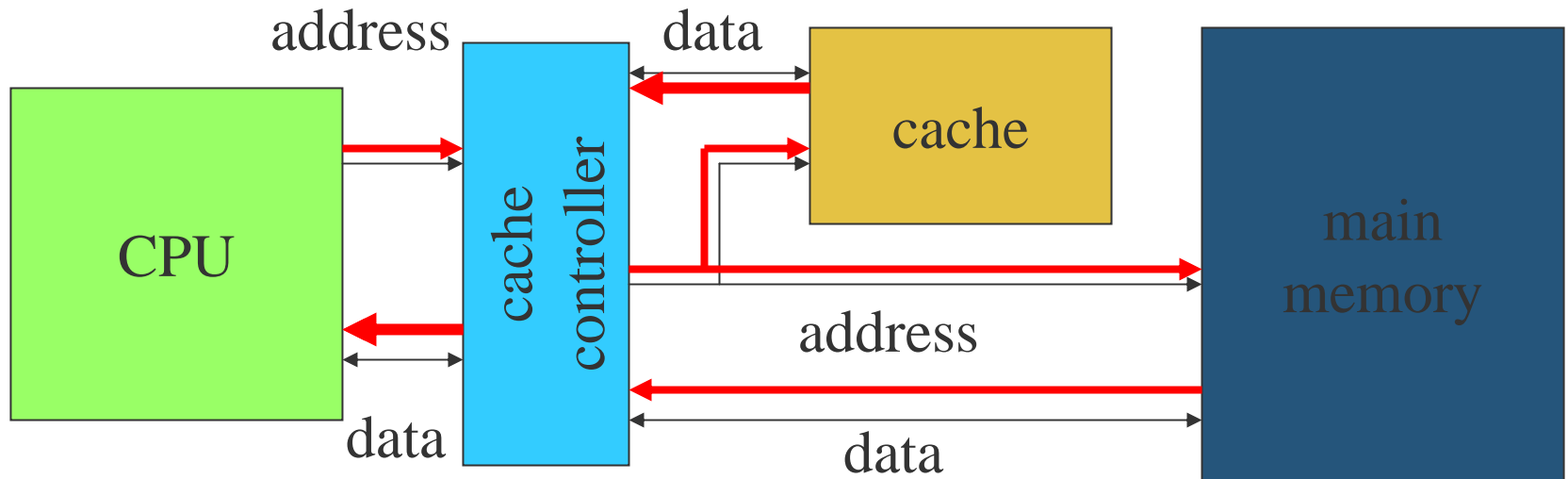
# Caching in Memory Hierarchy

- ❖ Exploits two hardware characteristics
  - Smaller memory provides faster access times
  - Large memory provides cheaper storage per byte
- ❖ Puts frequently accessed data in small, fast, and expensive memory

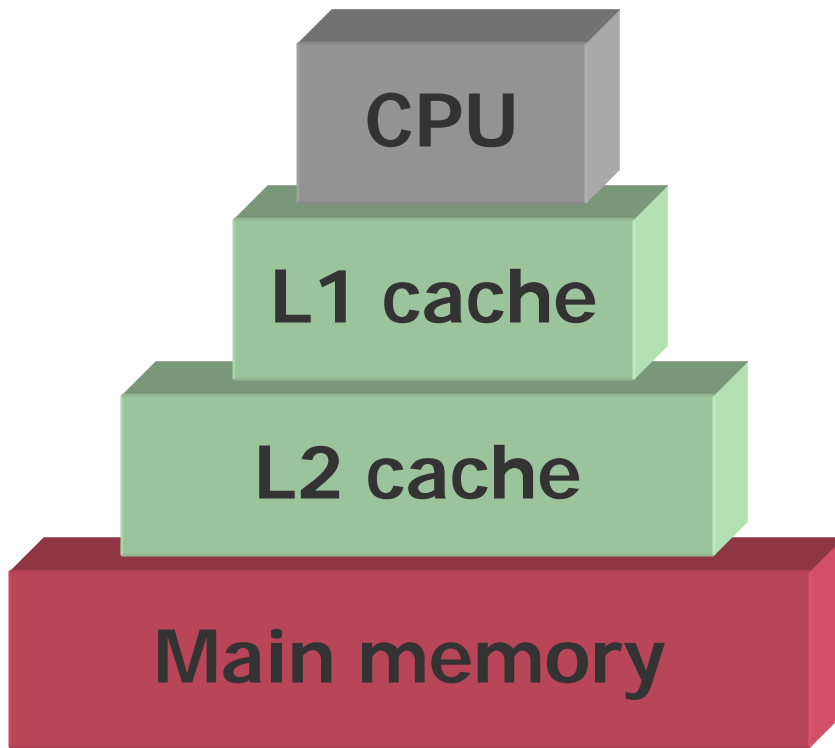
# Cache Read Operation - Flowchart



# Cache and CPU (1/2)



# Cache and CPU (2/2)



Roughly

1 cycle

~1-5 cycles

~5-20 cycles

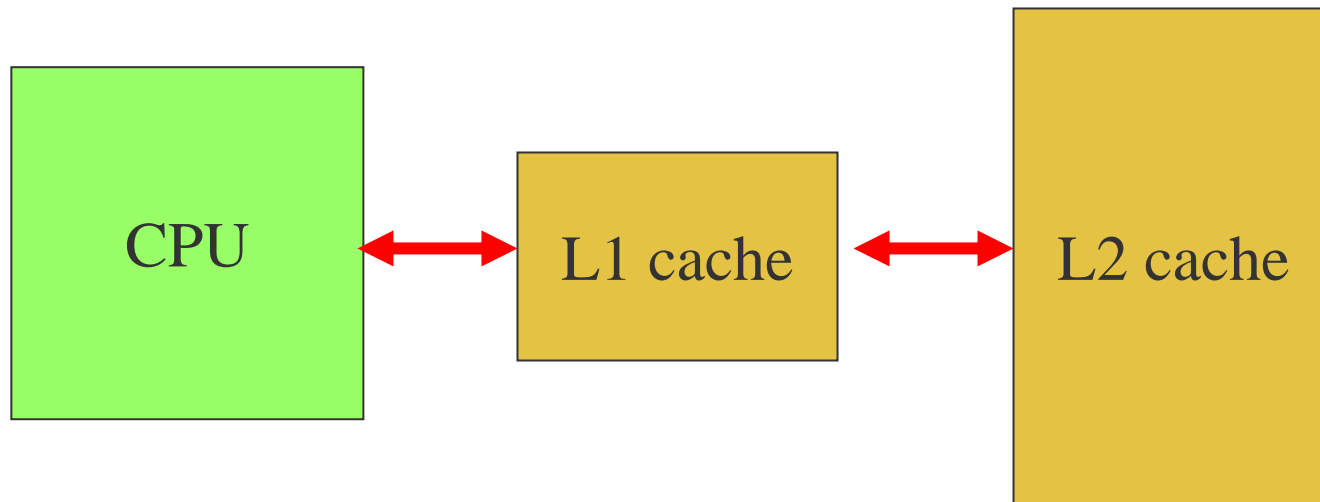
~40-100 cycles

# Some Cache Spec.

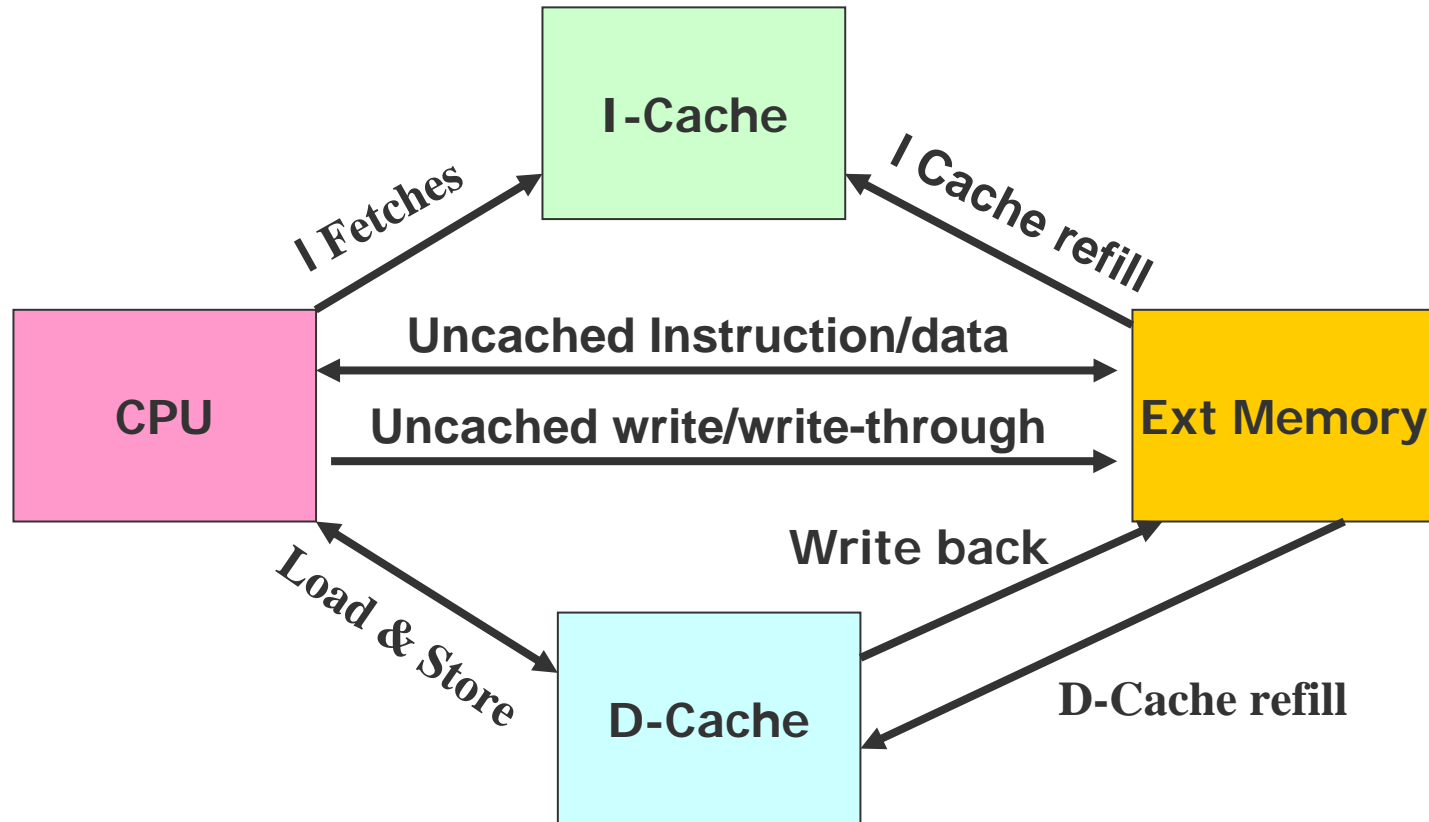
	L1 cache (I/D)	L2 cache
PS2	16K/8K <sup>†</sup> 2-way	N/A
GameCube	32K/32K <sup>‡</sup> 8-way	256K 2-way unified
XBOX	16K/16K 4-way	128K 8-way unified
PC	~32-64K	~128-512K



# Multiple levels of cache



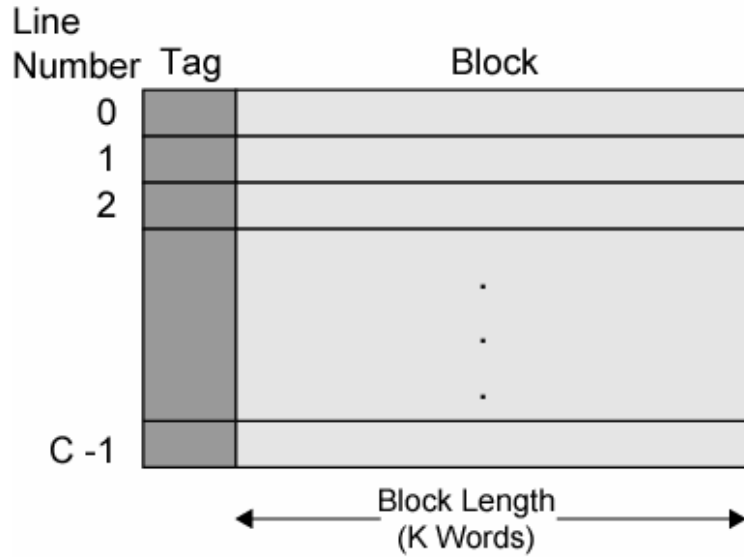
# Cache data flow



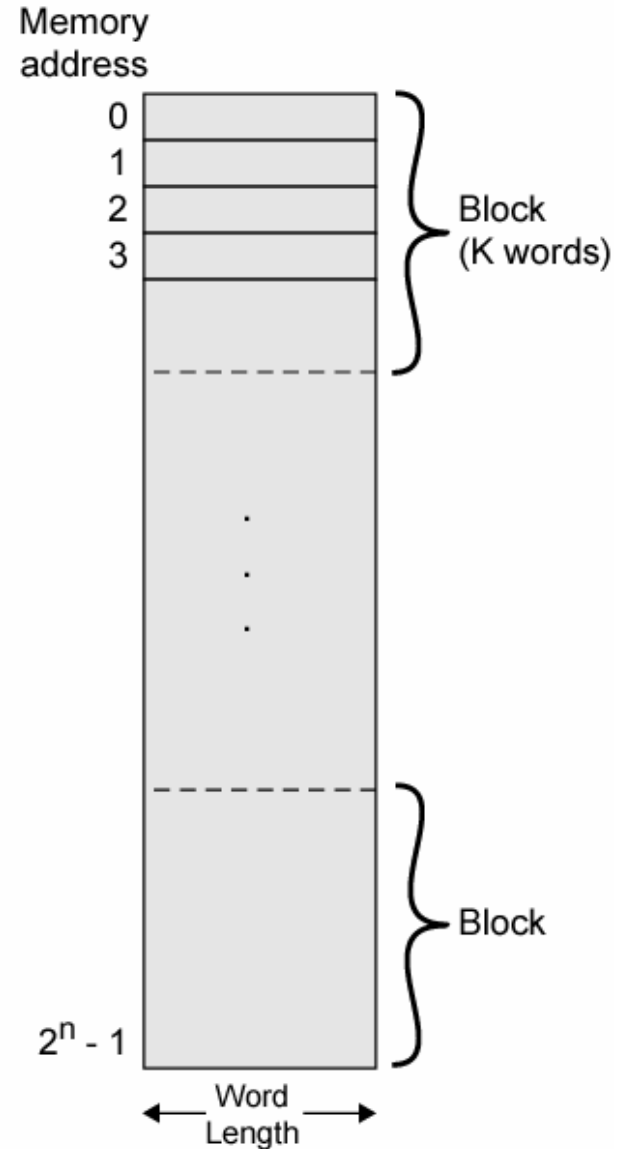
# Cache operation

- ❖ CPU requests contents of memory location
- ❖ Check cache for this data
- ❖ If present, get from cache (fast)
- ❖ If not present, read required block from main memory to cache
- ❖ Then deliver from cache to CPU
- ❖ Cache includes tags to identify which block of main memory is in each cache slot
- ❖ Many main memory locations are mapped onto one cache entry.
- ❖ May have caches for:
  - instructions;
  - data;
  - data + instructions (unified).

# Cache/Main Memory Structure

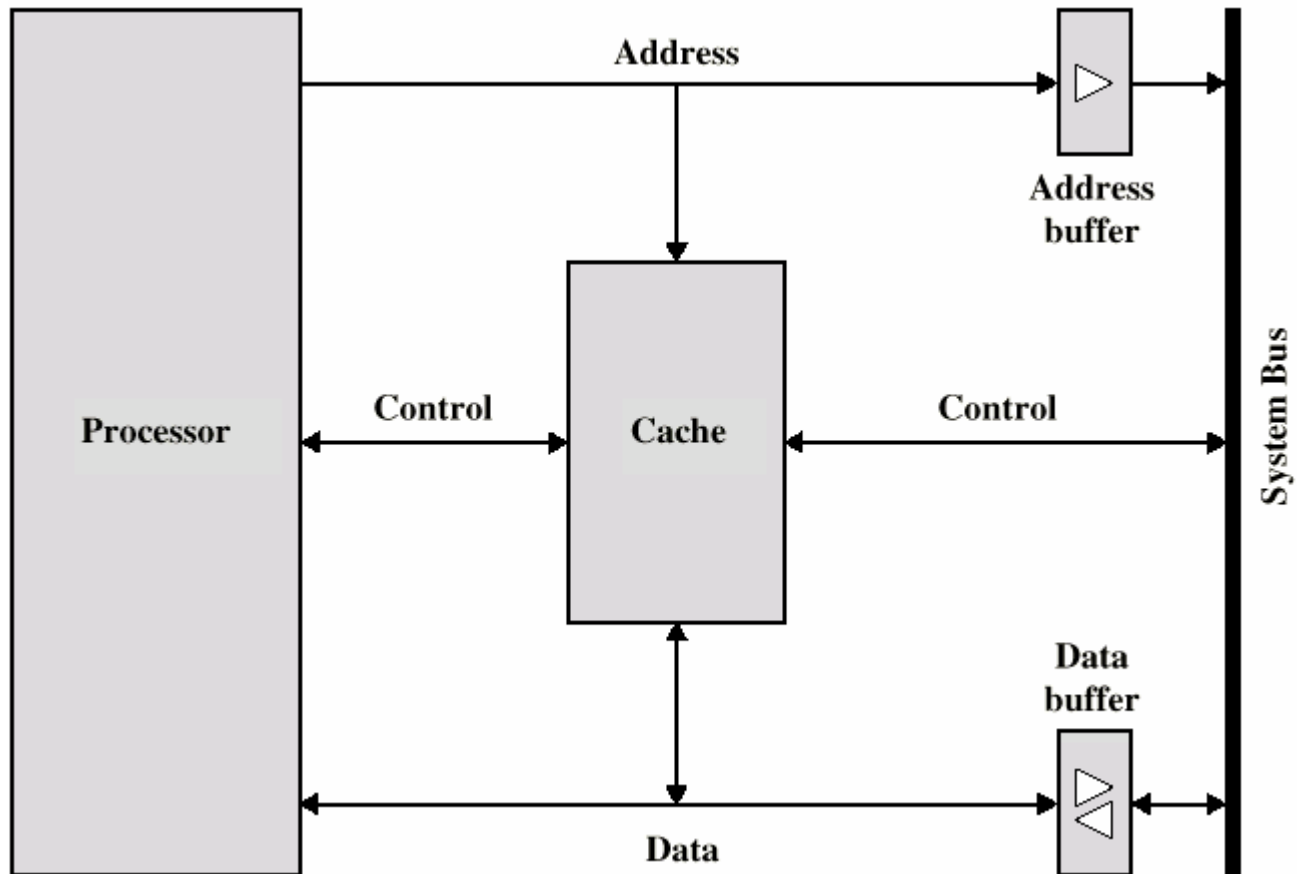


(a) Cache



(b) Main memory

# Typical Cache Organization

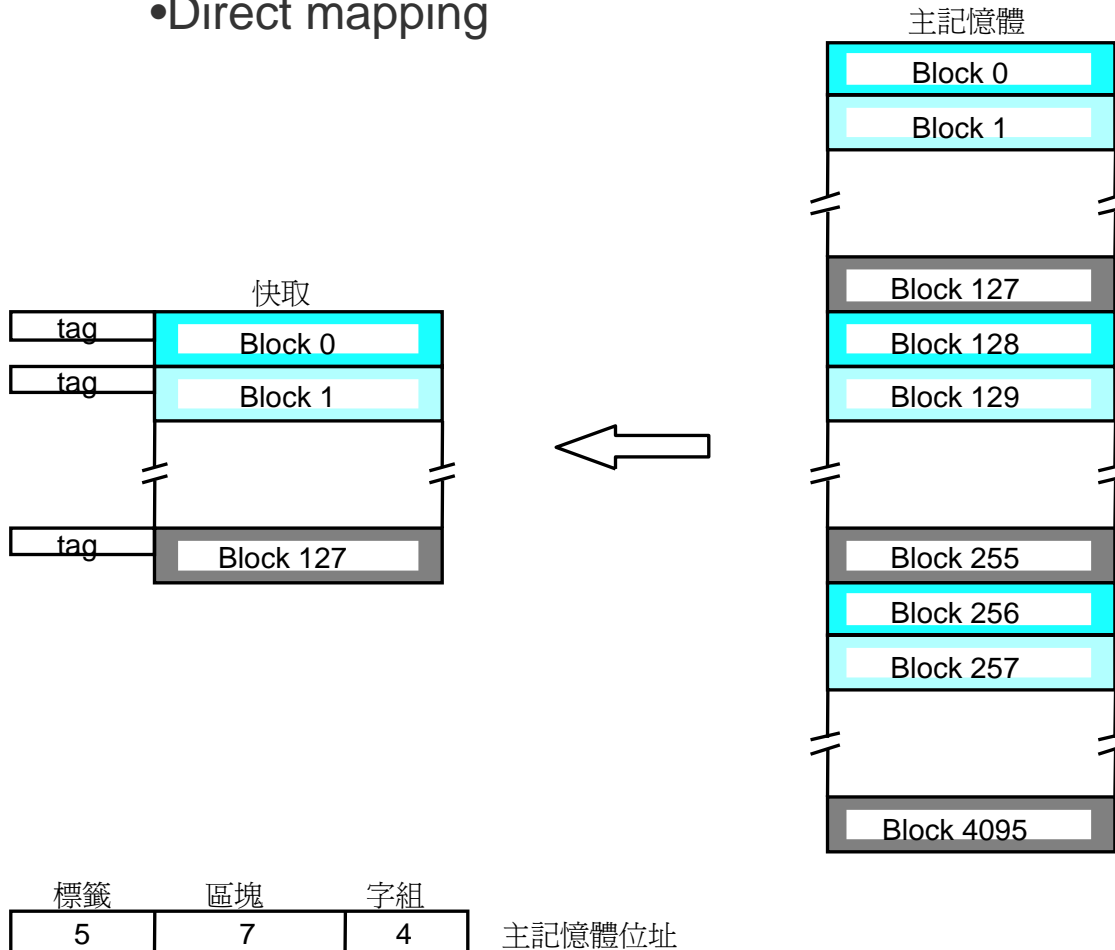


# Direct Mapping (1/2)

- ❖ Each block of main memory maps to only one cache line
  - i.e. if a block is in cache, it must be in one specific place
- ❖ Address is in two parts
- ❖ Least Significant  $w$  bits identify unique word
- ❖ Most Significant  $s$  bits specify one memory block
- ❖ The MSBs are split into a cache line field  $r$  and a tag of  $s-r$  (most significant)

# Direct Mapping (2/2)

## •Direct mapping



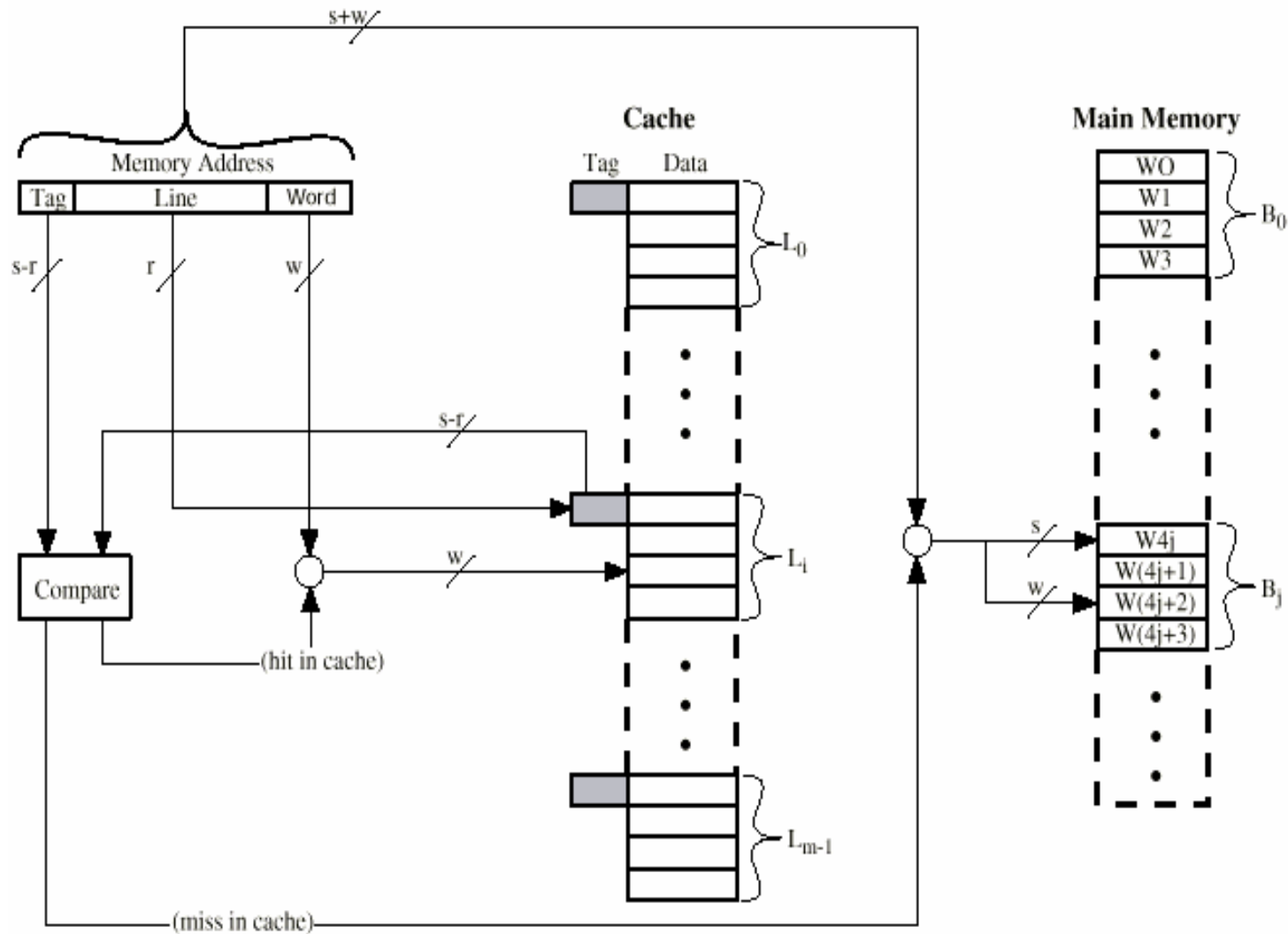
# Direct Mapping Address Structure

Tag $s-r$	Line or Slot $r$	Word $w$
8	14	2

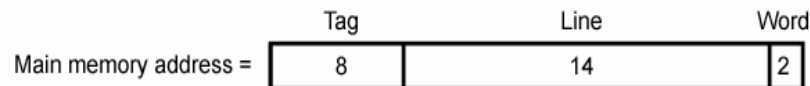
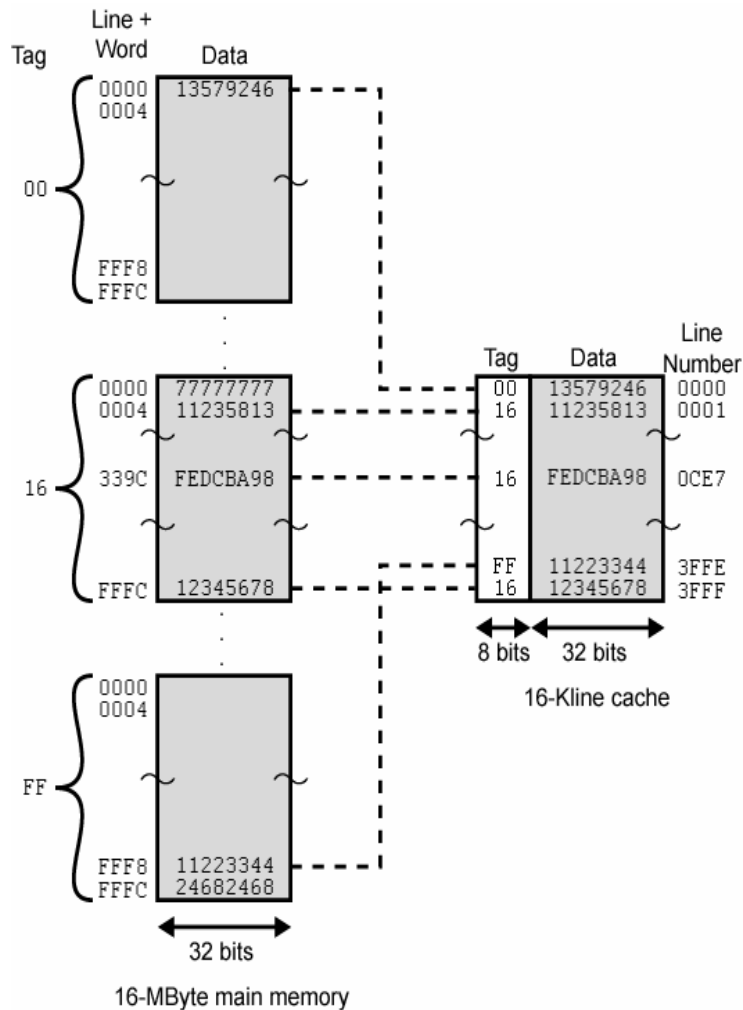
- ❖ 24 bit address
- ❖ 2 bit word identifier (4 byte block)
- ❖ 22 bit block identifier
  - 8 bit tag (=22-14)
  - 14 bit slot or line
- ❖ No two blocks in the same line have the same Tag field
- ❖ Check contents of cache by finding line and checking Tag



# Direct Mapping Cache Organization



# Direct Mapping Example



# Direct Mapping Summary

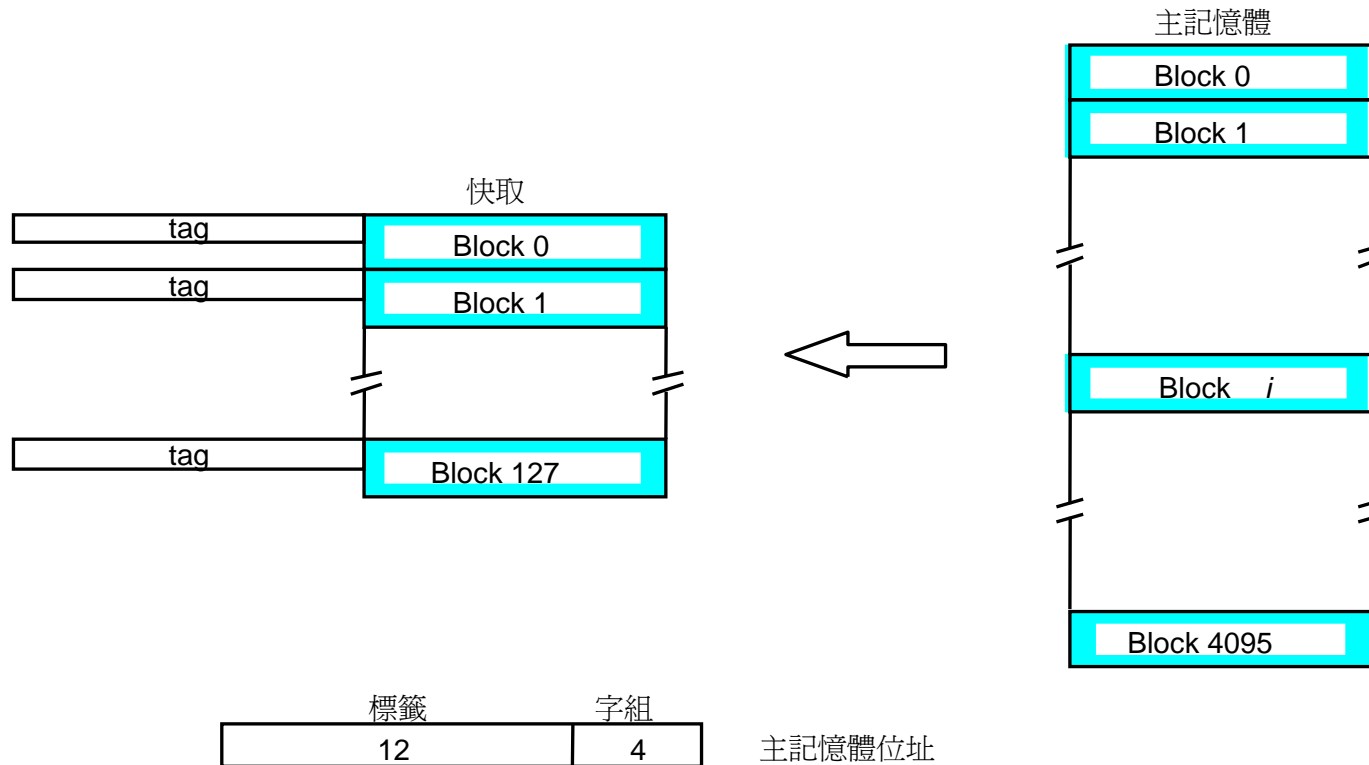
- ❖ Address length =  $(s + w)$  bits
- ❖ Number of addressable units =  $2^{s+w}$  words or bytes
- ❖ Block size = line size =  $2^w$  words or bytes
- ❖ Number of blocks in main memory =  $2^{s+w}/2^w = 2^s$
- ❖ Number of lines in cache =  $m = 2^r$
- ❖ Size of tag =  $(s - r)$  bits

# Associative Mapping (1/2)

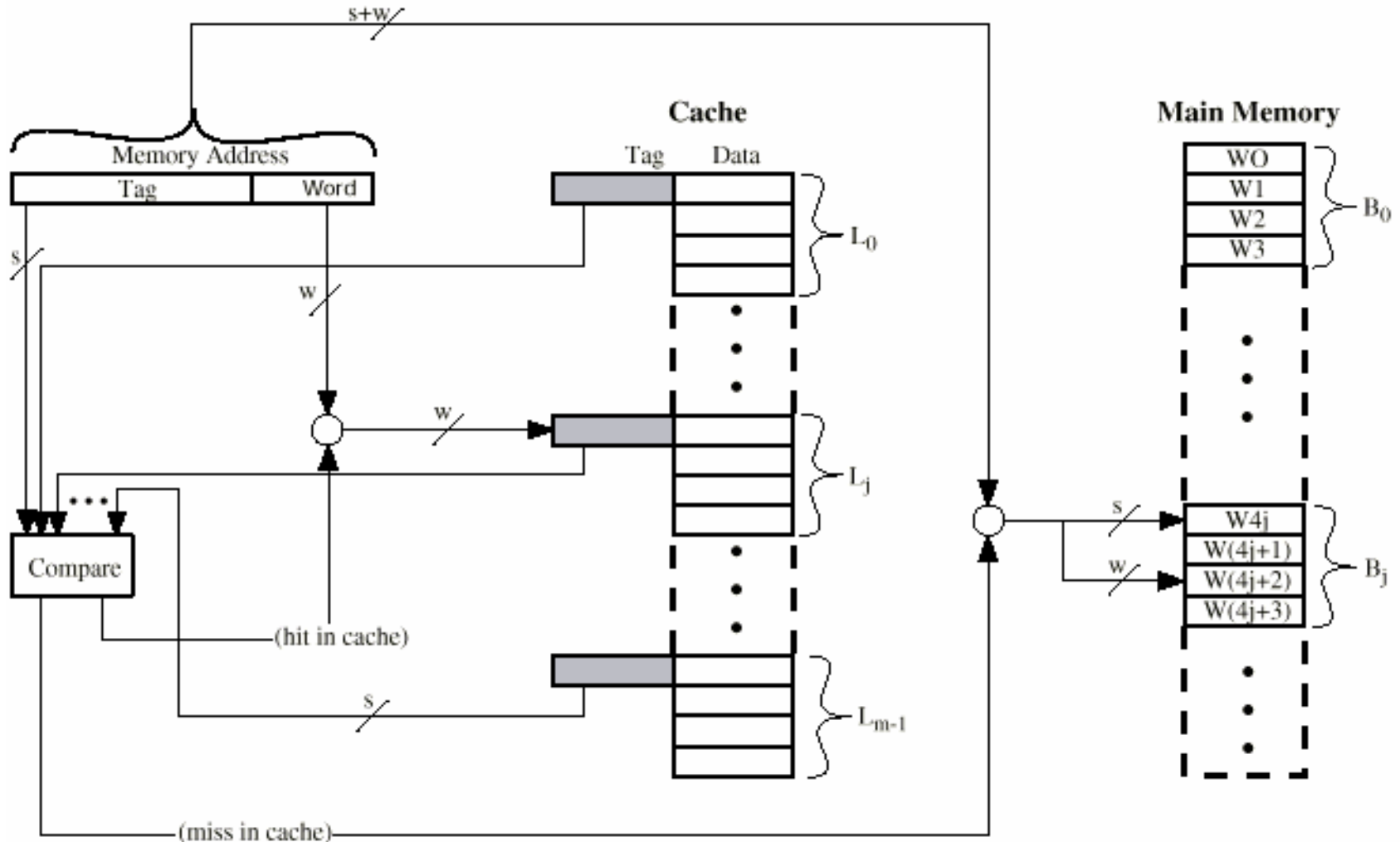
- ❖ A main memory block can load into any line of cache
- ❖ Memory address is interpreted as tag and word
- ❖ Tag uniquely identifies block of memory
- ❖ Every line's tag is examined for a match
- ❖ Cache searching gets expensive

# Associative Mapping (2/2)

Fully associative mapping(完全關聯映射)



# Fully Associative Cache Organization



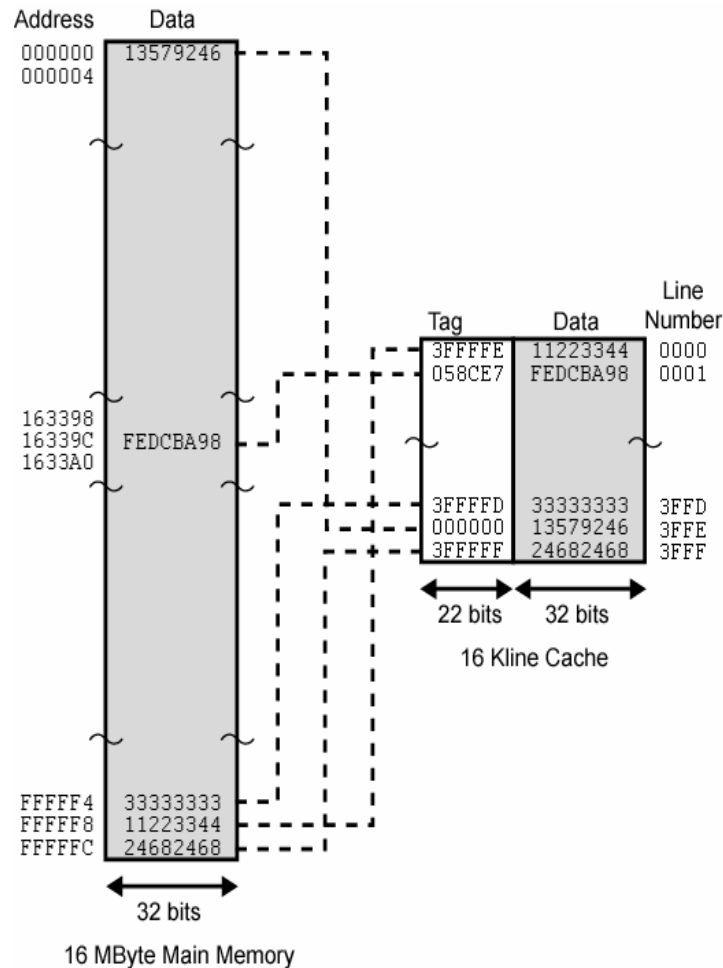
# Associative Mapping Address Structure



- ❖ 22 bit tag stored with each 32 bit block of data
- ❖ Compare tag field with tag entry in cache to check for hit
- ❖ e.g.

Address line	Tag	Data	Cache
▪ FFFFFC	FFFFFC	24682468	3FFF

# Associative Mapping Example





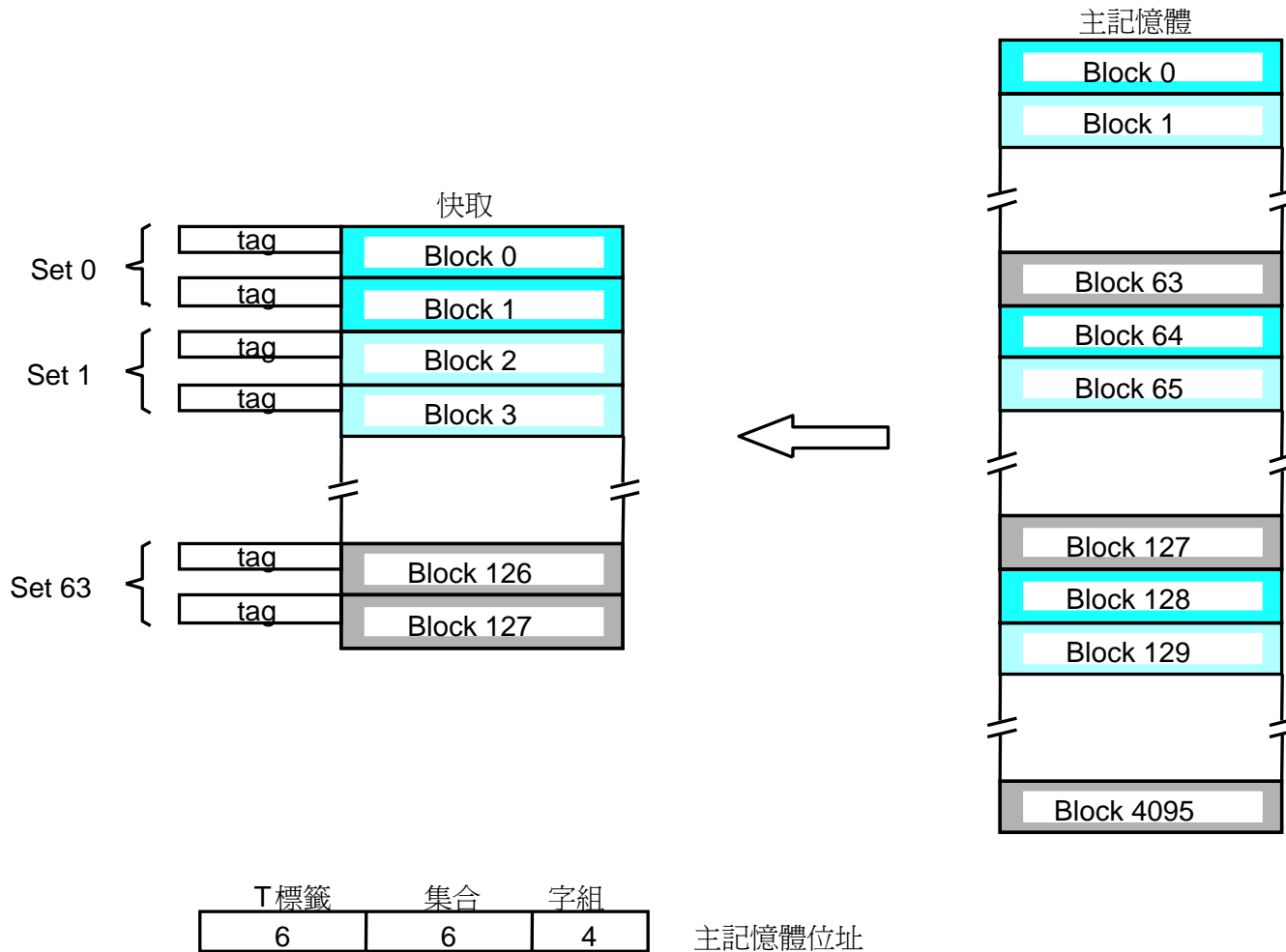
# Associative Mapping Summary

- ❖ Address length =  $(s + w)$  bits
- ❖ Number of addressable units =  $2^{s+w}$  words or bytes
- ❖ Block size = line size =  $2w$  words or bytes
- ❖ Number of blocks in main memory =  $2^{s+w}/2^w = 2^s$
- ❖ Number of lines in cache = undetermined
- ❖ Size of tag =  $s$  bits

# Set Associative Mapping (1/2)

- ❖ Cache is divided into a number of sets
- ❖ Each set contains a number of lines
- ❖ A given block maps to any line in a given set
  - e.g. Block B can be in any line of set i
- ❖ e.g. 2 lines per set
  - 2 way associative mapping
  - A given block can be in one of 2 lines in only one set

# Set Associative Mapping (2/2)



# Set Associative Mapping Address Structure

Tag 9 bit	Set 13 bit	Word 2 bit
-----------	------------	---------------

- ❖ Use set field to determine cache set to look in
- ❖ Compare tag field to see if we have a hit
- ❖ e.g

Address	Tag	Data	Set number
1FF 7FFC	1FF	12345678	1FFF
001 7FFC	001	11223344	1FFF

# Set Associative Mapping Example

- ❖ 13 bit set number
- ❖ Block number in main memory is modulo  $2^{13}$
- ❖ 000000, 00A000, 00B000, 00C000 ... map to same set

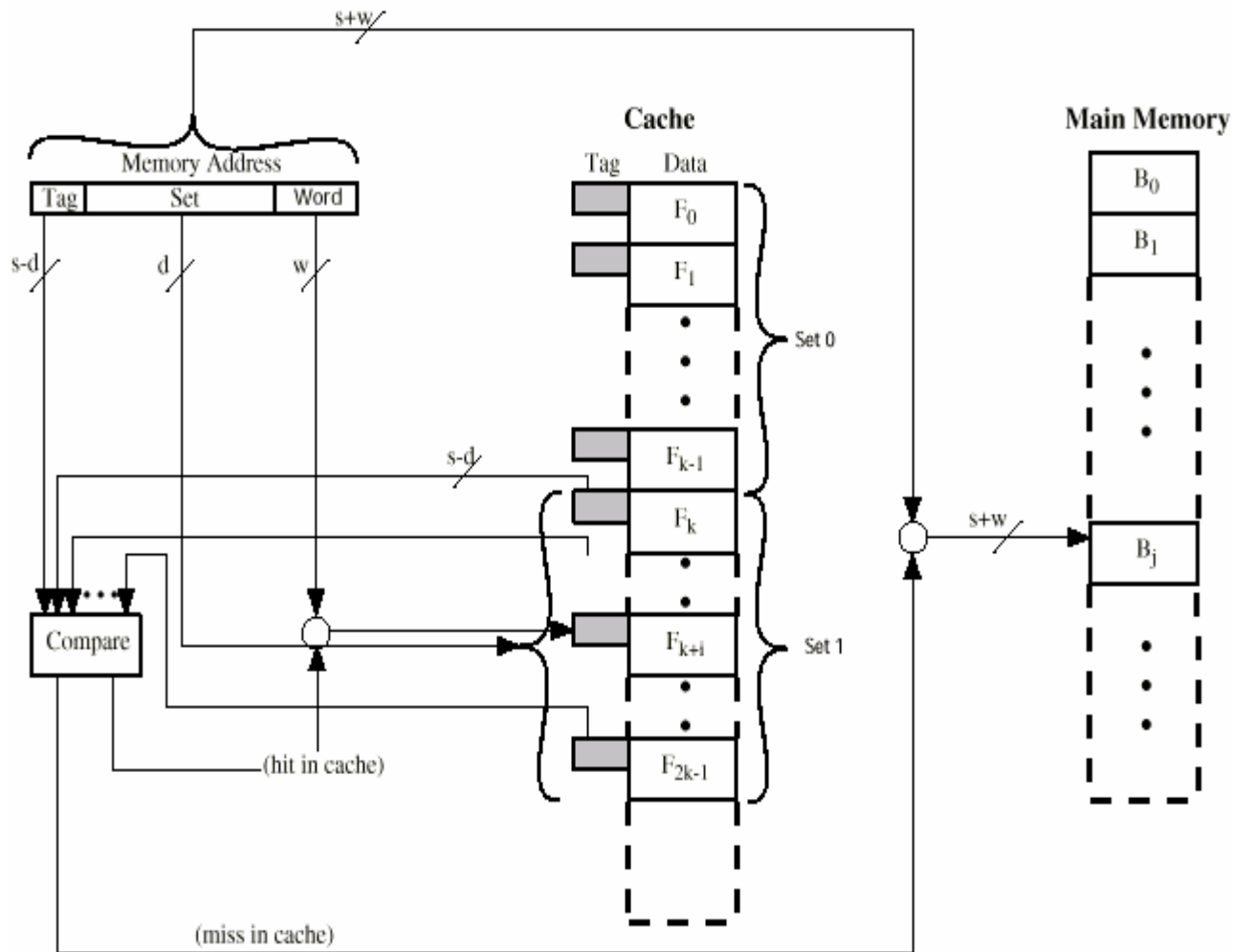
# Set Associative Mapping Address Structure

Tag 9 bit	Set 13 bit	Word 2 bit
-----------	------------	---------------

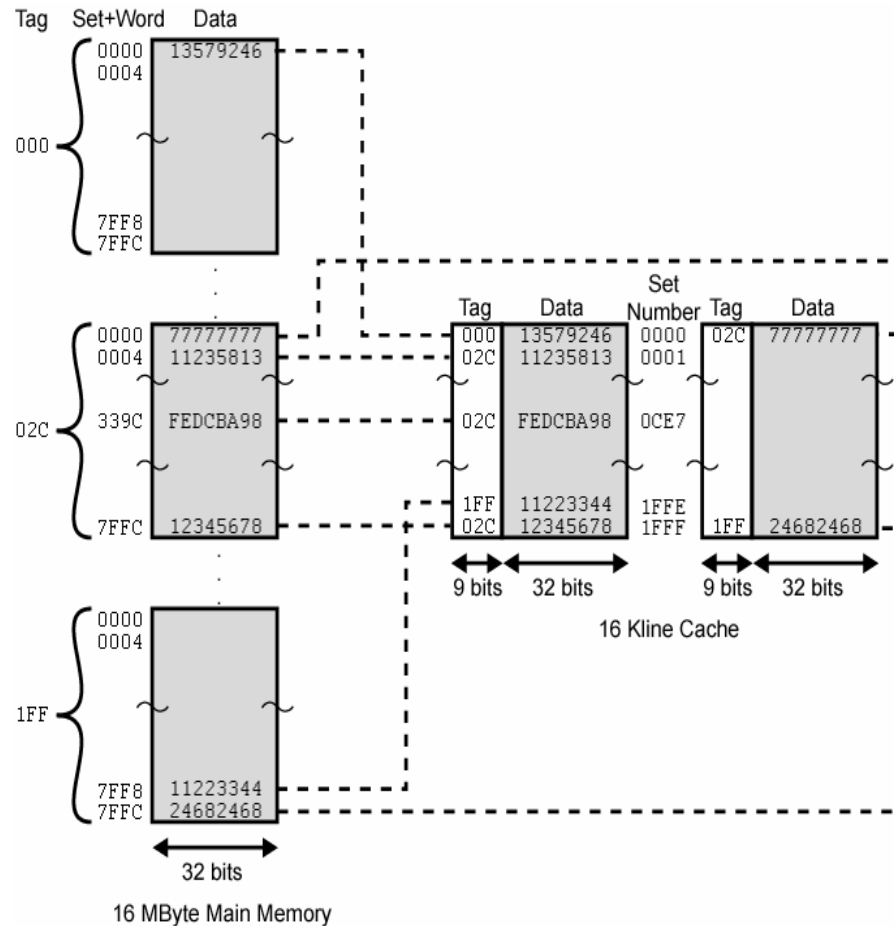
- ❖ Use set field to determine cache set to look in
- ❖ Compare tag field to see if we have a hit
- ❖ e.g

Address	Tag	Data	Set number
1FF 7FFC	1FF	12345678	1FFF
001 7FFC	001	11223344	1FFF

# Two Way Set Associative Cache Organization



# Two Way Set Associative Mapping Example



	Tag	Set	Word
Main Memory Address =	9	13	2



# Set Associative Mapping Summary

- ❖ Address length =  $(s + w)$  bits
- ❖ Number of addressable units =  $2^{s+w}$  words or bytes
- ❖ Block size = line size =  $2^w$  words or bytes
- ❖ Number of blocks in main memory =  $2^d$
- ❖ Number of lines in set =  $k$
- ❖ Number of sets =  $v = 2^d$
- ❖ Number of lines in cache =  $k v = k * 2^d$
- ❖ Size of tag =  $(s - d)$  bits

# Replacement policy

## ❖ Replacement policy

- strategy for choosing which cache entry to throw out to make room for a new memory location

## ❖ Two popular strategies:

- Random
- Least-recently used (LRU)

# Write operations

## ❖ Write Policy

- Must not overwrite a cache block unless main memory is up to date
- Multiple CPUs may have individual caches
- I/O may address main memory directly

## ❖ Write-through

- immediately copy write to main memory
- immediately propagates update through various levels of caching
  - For critical data
- All writes go to main memory as well as cache
- Multiple CPUs can monitor main memory traffic to keep local (to CPU) cache up to date
- Lots of traffic
- Slows down writes

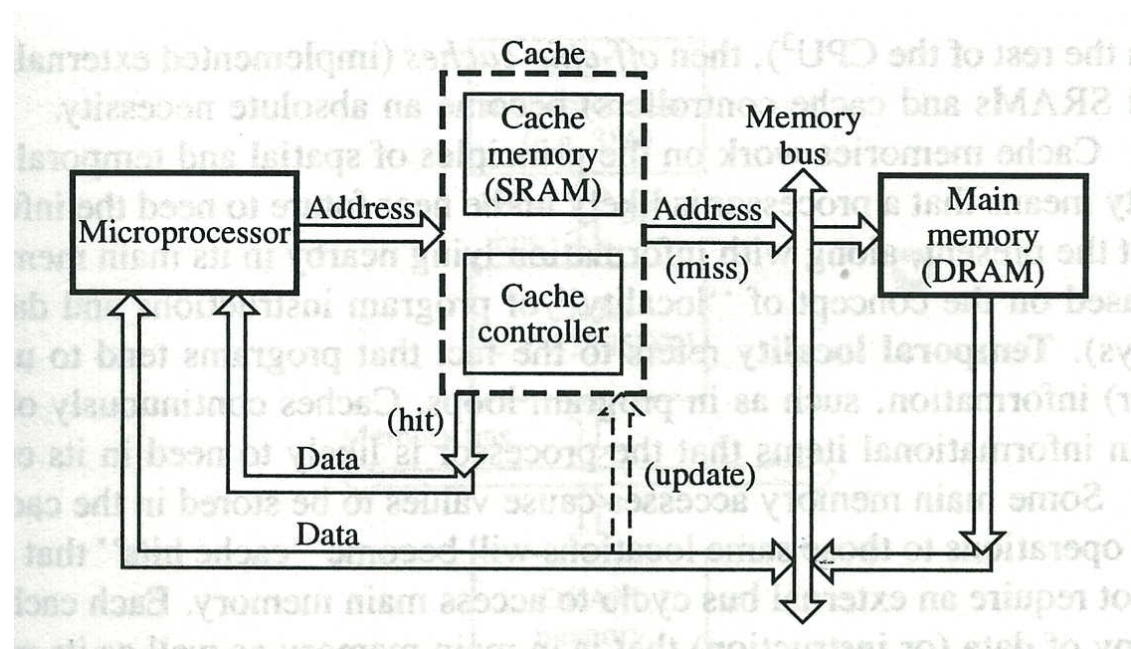
# Write operations

## ❖ Write-back

- write to main memory only when location is removed from cache
- delays the propagation until the cached item is replaced
  - Goal: spread the cost of update propagation over multiple updates
  - Less costly
- Updates initially made in cache only
- Update bit for cache slot is set when update occurs
- Other caches get out of synchronization
- If block is to be replaced, write to main memory only if update bit is set

# Generic Issues in Caching

- ❖ Cache hit: requested instruction/data by microprocessor is found in the cache
- ❖ Cache miss: request instruction/data is not in the cache => read from main memory (DRAM) and the associated data is copied in the cache (cache update)



# Reasons for Cache Misses

- ❖ Compulsory misses: data brought into the cache for the first time
  - e.g., booting
- ❖ Capacity misses: caused by the limited size of a cache
  - A program may require a hash table that exceeds the cache capacity
    - Random access pattern
    - No caching policy can be effective

# Reasons for Cache Misses

- ❖ Misses due to competing cache entries: a cache entry assigned to two pieces of data
  - When both active
  - Each will preempt the other
- ❖ Policy misses: caused by cache replacement policy, which chooses which cache entry to replace when the cache is full

# Improving Cache Performance

## ❖ Goal: reduce the Average Memory Access Time (AMAT)

- $AMAT = \text{Hit Time} + \text{Miss Rate} * \text{Miss Penalty}$

## ❖ Approaches

- Reduce Hit Time
- Reduce or Miss Penalty
- Reduce Miss Rate

## ❖ Notes

- There may be conflicting goals
- Keep track of clock cycle time, area, and power consumption



# Effective Access Time

- ❖ Cache hit rate: 99%
  - Cost: 2 clock cycles
- ❖ Cache miss rate: 1%
  - Cost: 4 clock cycles
- ❖ Effective access time:
  - $99\% * 2 + 1\% * (2 + 4)$   
 $= 1.98 + 0.06 = \mathbf{2.04 \text{ (clock cycles)}}$

# Tuning Cache Parameters

## ❖ Size:

- Must be large enough to fit working set (temporal locality)
- If too big, then hit time degrades

## ❖ Associativity

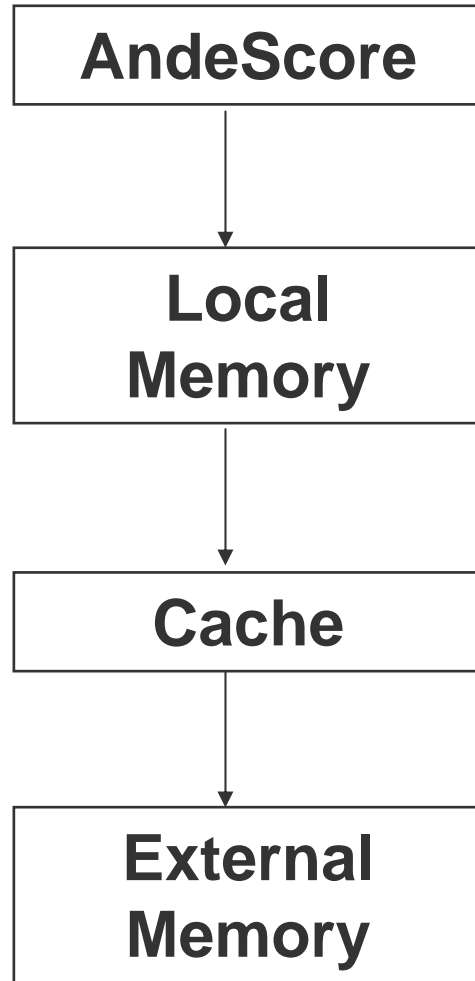
- Need large to avoid conflicts, but 4-8 way is as good a FA
- If too big, then hit time degrades

## ❖ Block

- Need large to exploit spatial locality & reduce tag overhead
- If too large, few blocks  $\Rightarrow$  higher misses & miss penalty

*Configurable architecture allows designers to make the best performance/cost trade-offs*

# Processor Core Start procedure

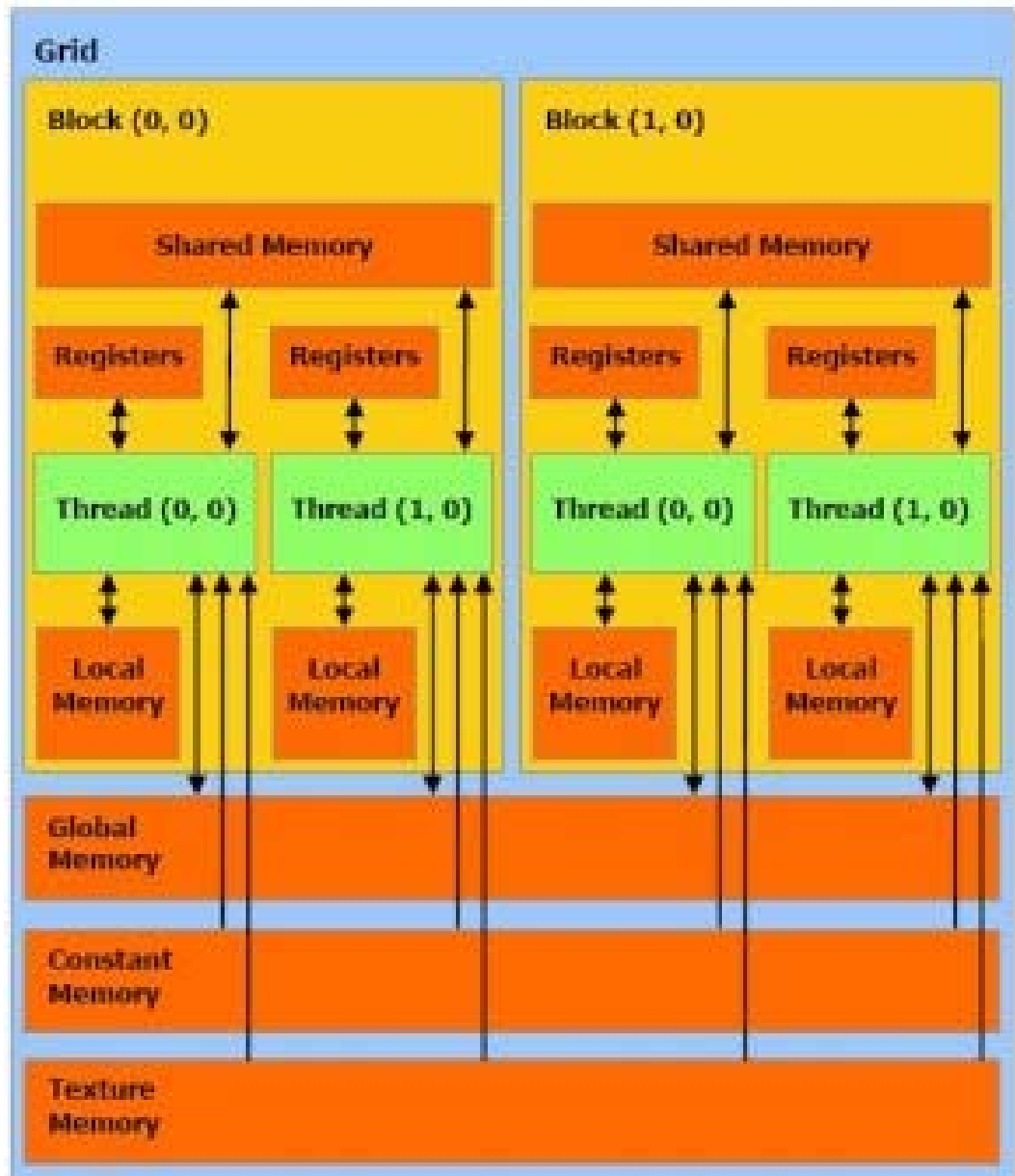


# Local memory

- ❖ Local memory is also used to describe a portion of memory that a software program or utility only has access to once obtained.
- ❖ On-chip memory, based on thread unit to use.
- ❖ The memory used by a single CPU or allocated to a single program or function.

# Local memory

## ❖ Memory model



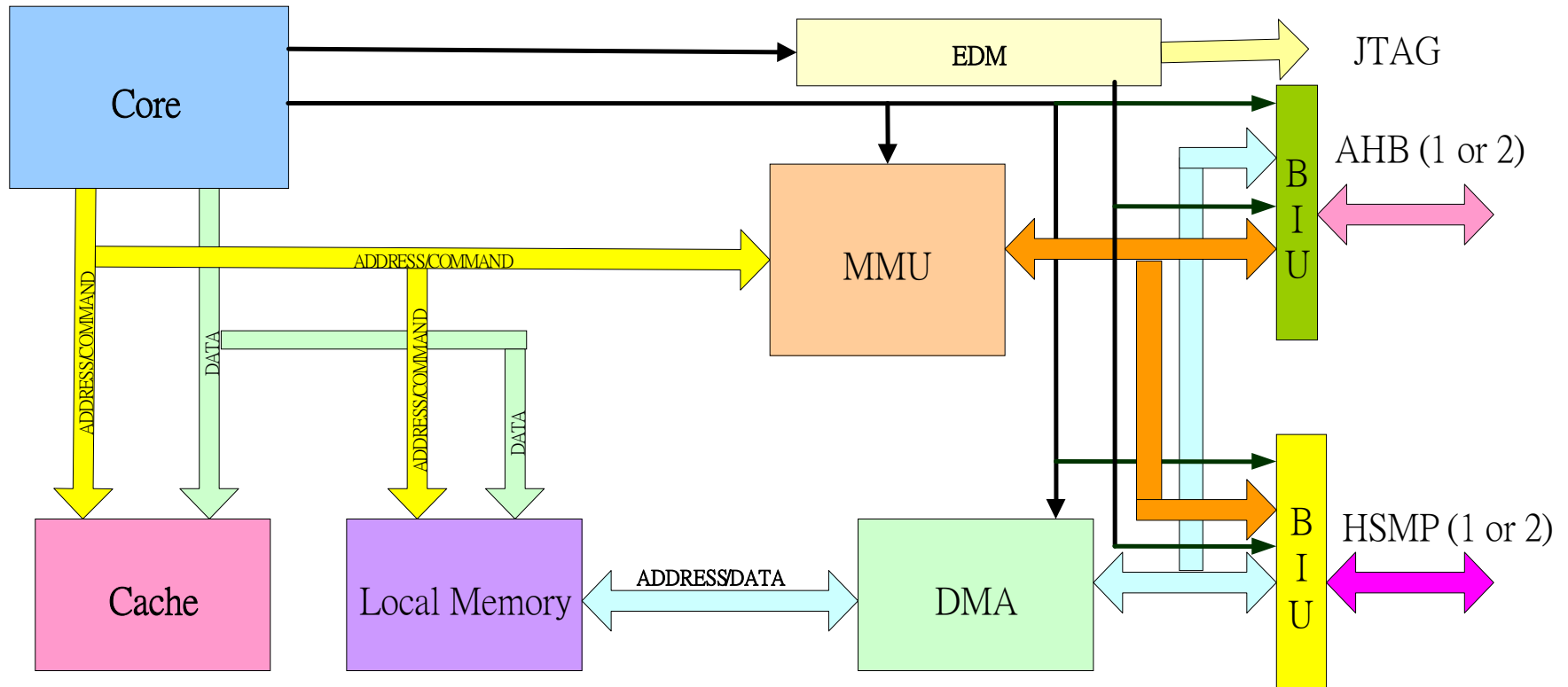
# Cache/Local Memory Design

- ❖ Size
- ❖ Mapping Function
- ❖ Replacement Algorithm
- ❖ Write Policy
- ❖ Block Size
- ❖ Number of Caches/Local Memories

# Memory Management Unit (MMU)

- ❖ Hardware device that maps virtual to physical address.
- ❖ In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- ❖ The user program deals with logical addresses; it never sees the real physical addresses.

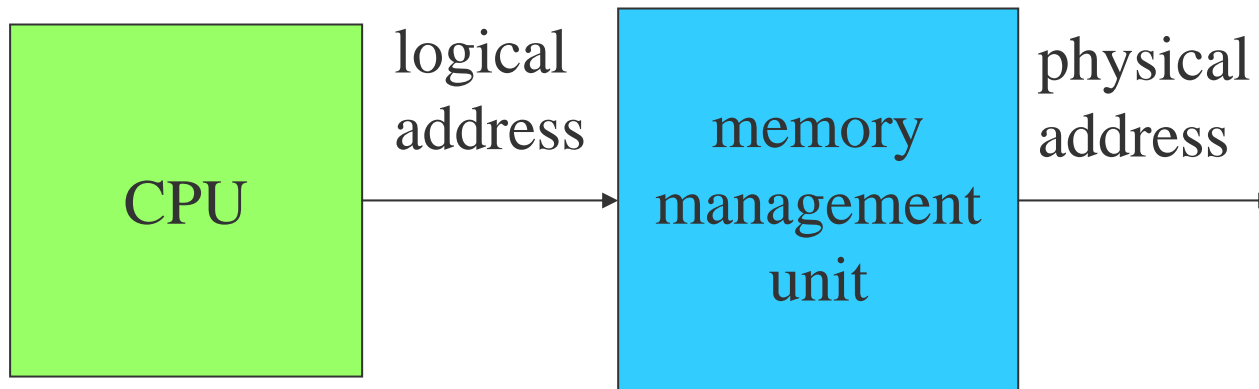
# Block diagram





# MMU Functionality

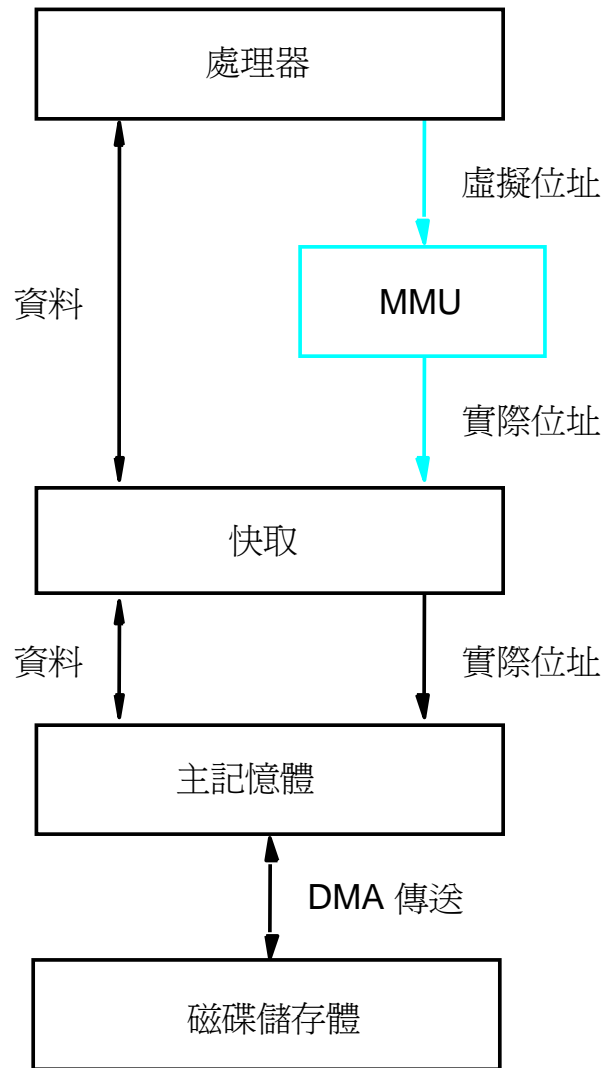
- ❖ Memory management unit (MMU) translates addresses



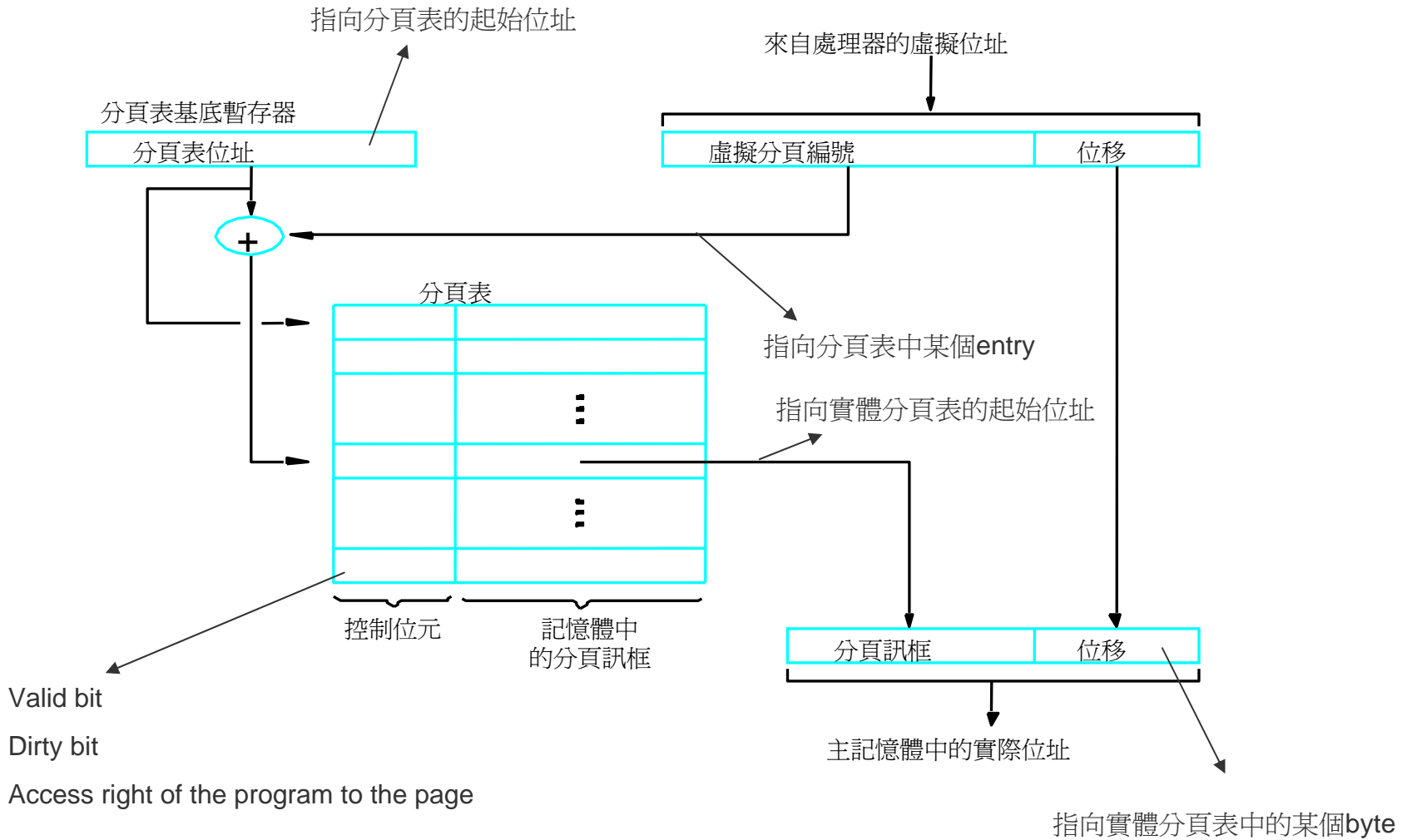
# Virtual memory

- ❖ Virtual address (logical address)
- ❖ MMU (built in CPU)
- ❖ Physical address
- ❖ Page table (in Main Memory)
- ❖ Page frame
- ❖ Address translation
- ❖ TLB
  - Cache built within CPU for holding translated address just used
- ❖ Page fault
- ❖ Replacement algorithm
  - LRU

# Virtual memory



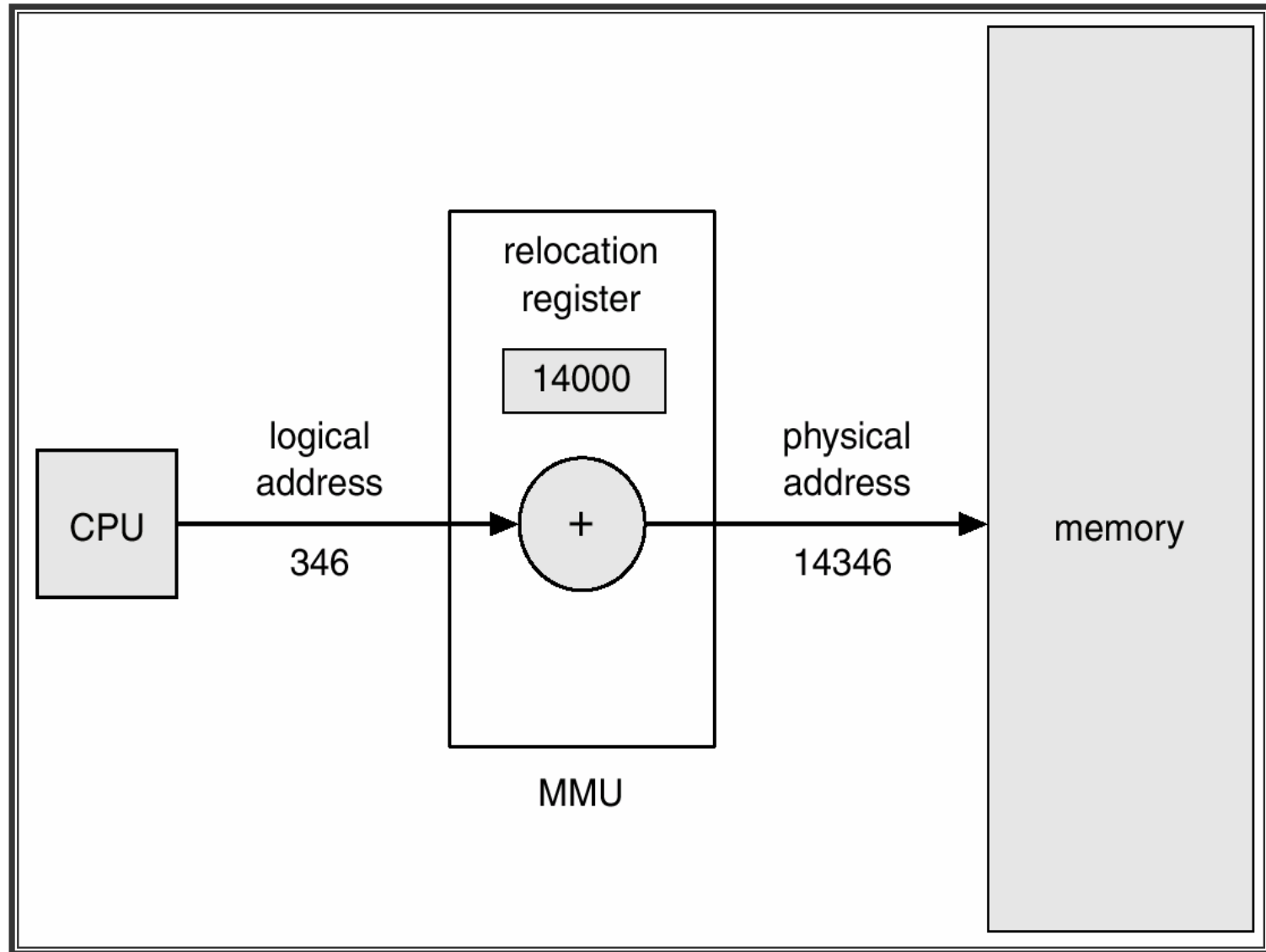
# Virtual memory



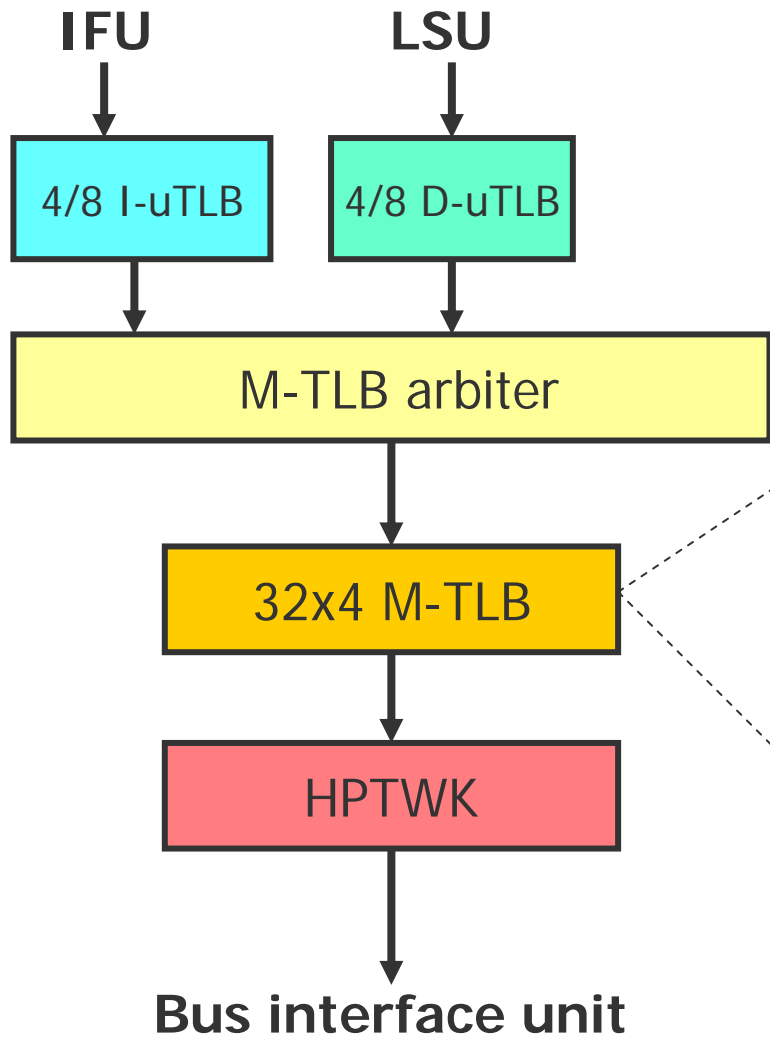
# Logical vs. Physical Address Space

- ❖ The concept of a logical *address space* that is bound to a separate physical address space is central to proper memory management.
  - Logical address – generated by the CPU; also referred to as virtual address.
  - Physical address – address seen by the memory unit.
- ❖ Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.

# Dynamic relocation using a relocation register



# MMU Architecture

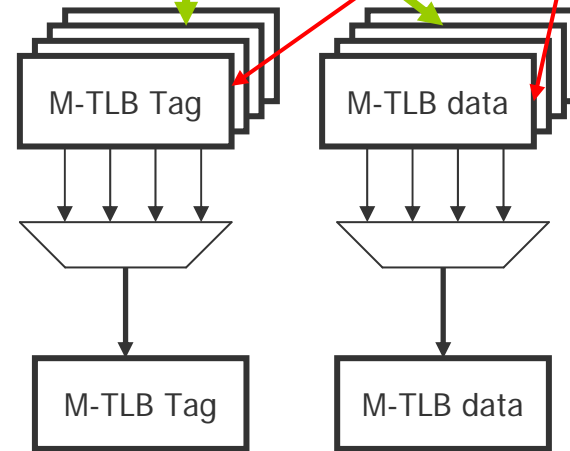


## M-TLB entry index

$N(=32)$  sets  $k(=4)$  ways = 128-entry



$\text{Log}_2(N \cdot K) - 1$        $\text{Log}_2(N)$        $\text{Log}_2(N) - 1$       0

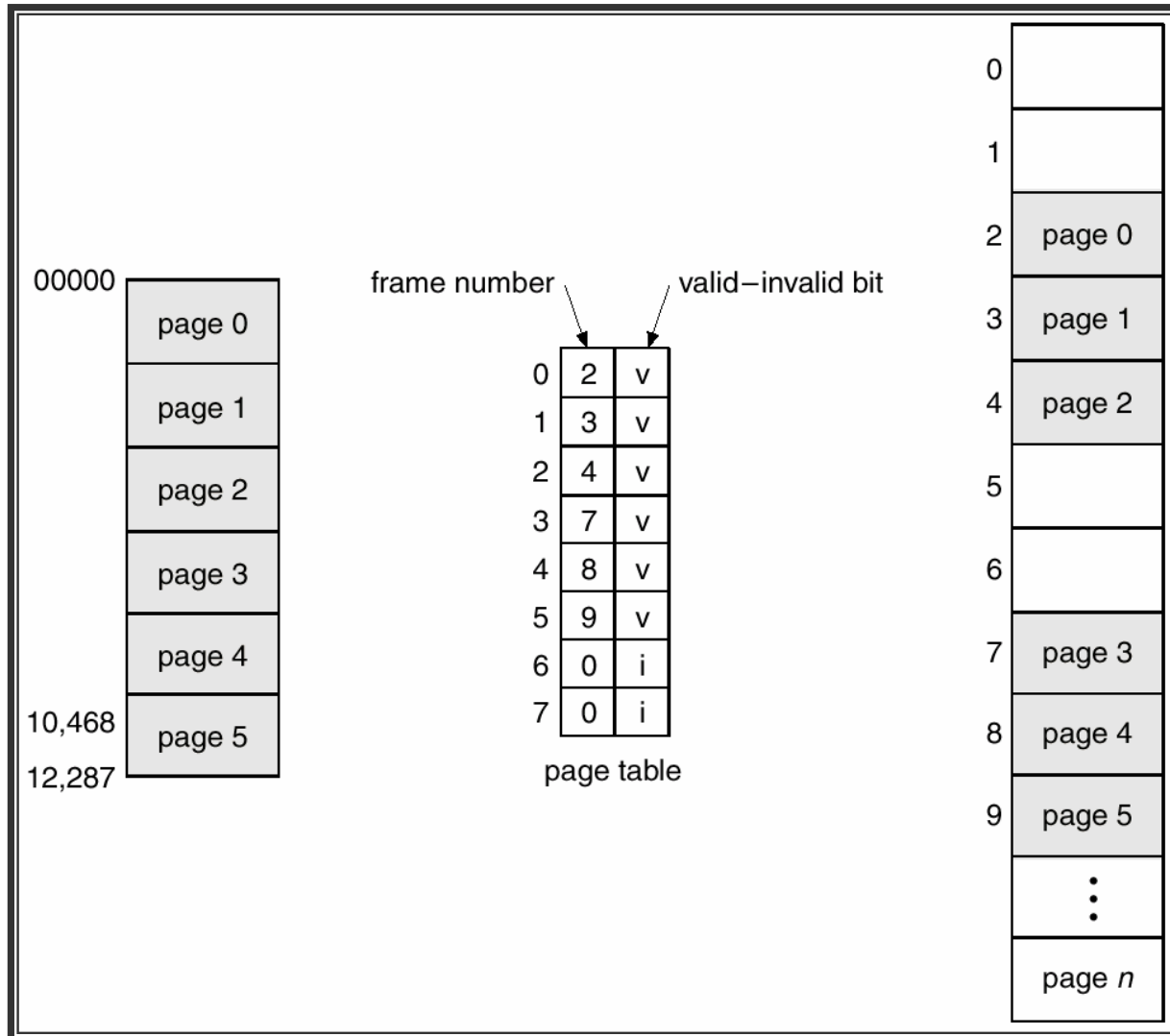


# MMU Functionality

- ❖ Virtual memory addressing
  - Better memory allocation, less fragmentation
  - Allows shared memory
  - Dynamic loading
- ❖ Memory protection (read/write/execute)
  - Different permission flags for kernel/user mode
  - OS typically runs in kernel mode
  - Applications run in user mode
  - Implemented by associating protection bit with each frame.
  - *Valid-invalid* bit attached to each entry in the page table:
    - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page.
    - “invalid” indicates that the page is not in the process’ logical address space.
- ❖ Cache control (cached/uncached)
  - Accesses to peripherals and other processors needs to be uncached.



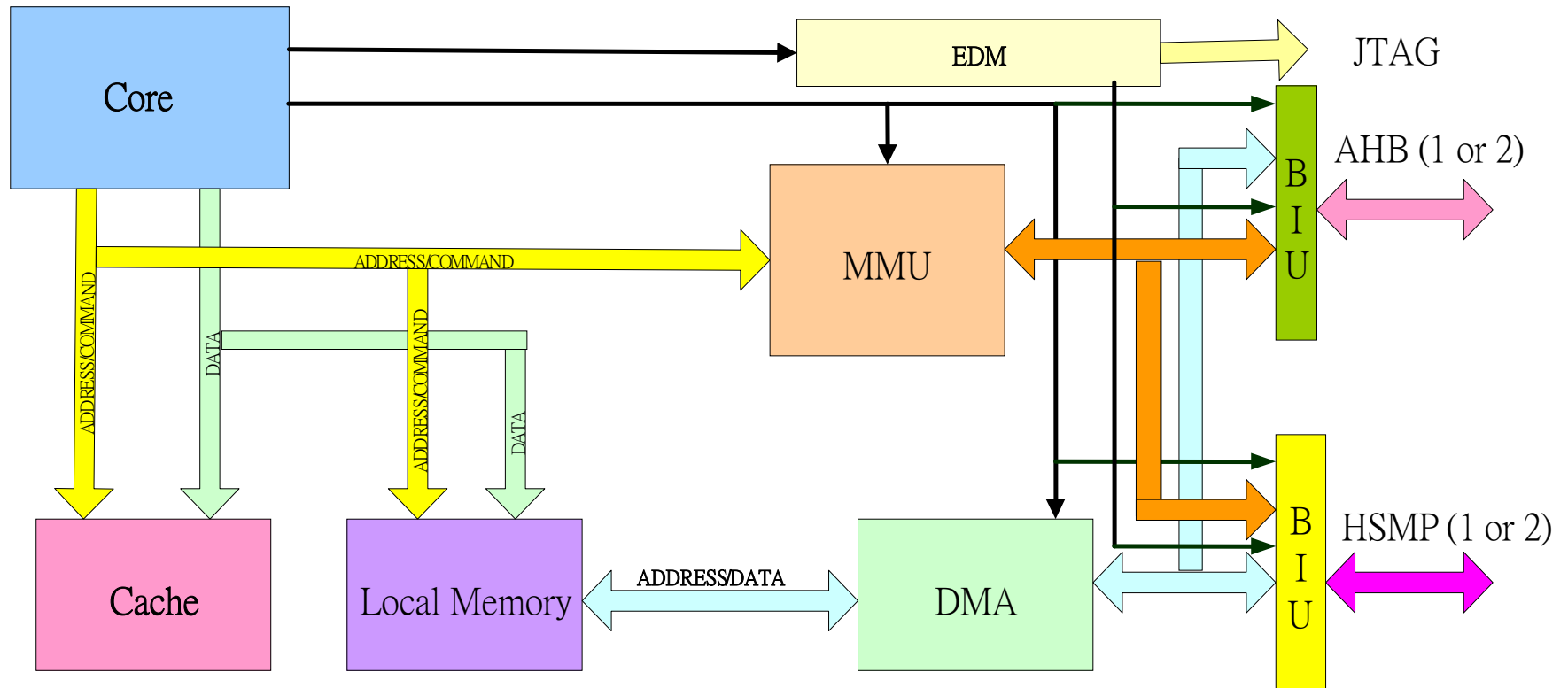
# Valid (v) or Invalid (i) Bit In A Page Table



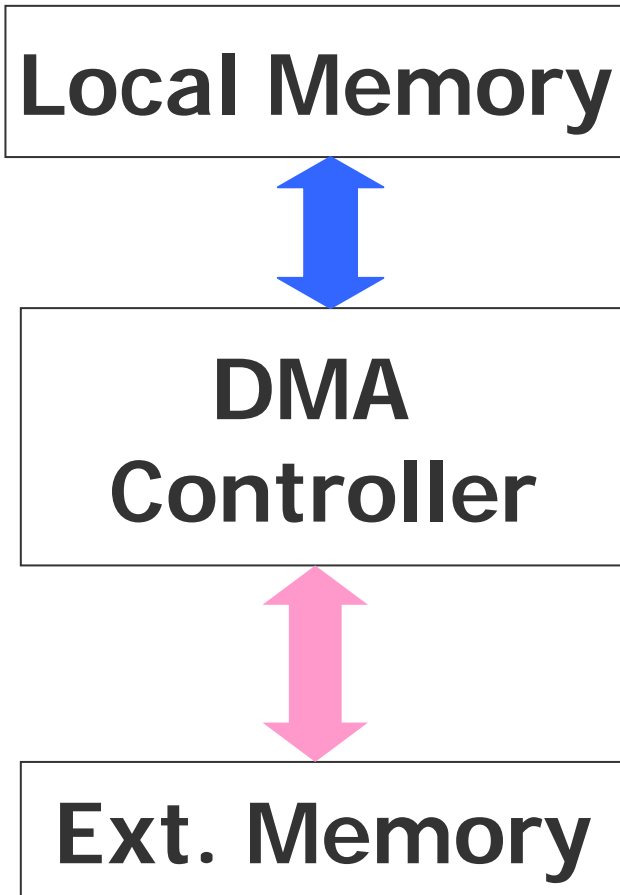
# Direct Memory Access (DMA)

- ❖ **Direct memory access (DMA)** is a feature of modern computers and microprocessors that allows certain hardware subsystems within the computer to access system memory for reading and/or writing independently of the central processing unit.
- ❖ Many hardware systems use DMA including disk drive controllers, graphics cards, network cards and sound cards.
- ❖ DMA is also used for intra-chip data transfer, especially in multiprocessor system-on-chips, where its processing element is equipped with a local memory and DMA is used for transferring data between the local memory and the main memory.
- ❖ Computers that have DMA channels can transfer data to and from devices with much less CPU overhead than computers without a DMA channel.

# Block diagram

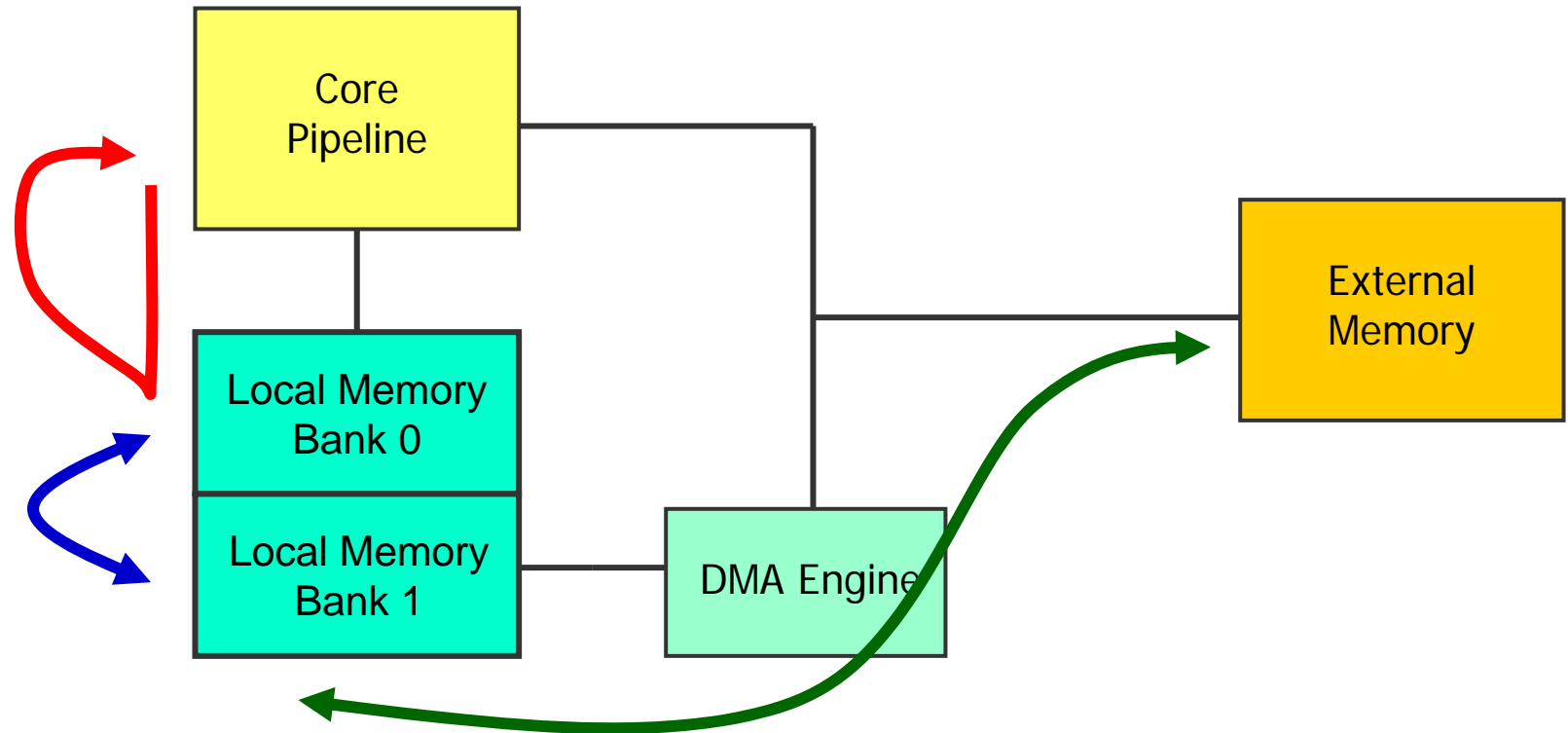





# DMA overview



- ❖ Two channels
- ❖ One active channel
- ❖ Programmed using physical addressing
- ❖ For both instruction and data local memory
- ❖ External address can be incremented with stride
- ❖ Optional 2-D Element Transfer (2DET) feature which provides an easy way to transfer two-dimensional blocks from external memory.

# LMDMA Double Buffer Mode



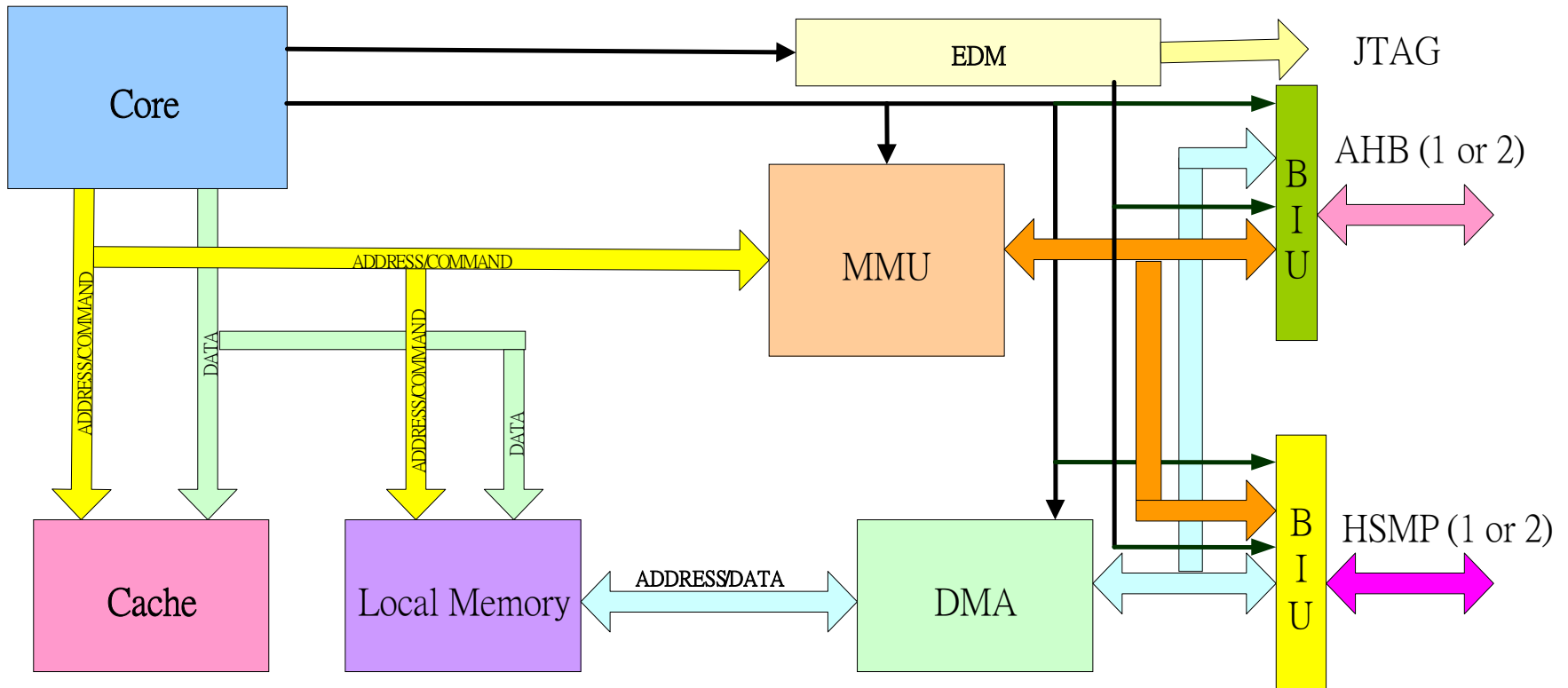
-  Computation
-  Data Movement
-  Bank Switch between core and DMA engine

Width byte stride (in DMA Setup register)=1

# Bus Interface Unit (BIU)

- ❖ The bus interface unit is the part of the processor that interfaces with the rest of the PC.
- ❖ It deals with moving information over the processor data bus, the primary conduit for the transfer of information to and from the CPU.
- ❖ The bus interface unit is responsible for responding to all signals that go to the processor, and generating all signals that go from the processor to other parts of the system.
- ❖ Bus Interface unit is responsible for off-CPU memory access which includes
  - System memory access
  - Instruction/data local memory access
  - Memory-mapped register access in devices.

# Block diagram



# Bus Interface

- ❖ Compliance to AHB/AHB-Lite/APB
- ❖ High Speed Memory Port
- ❖ Andes Memory Interface
- ❖ External LM Interface



# HSMP – High speed memory port

- ❖ N12 also provides a high speed memory port interface which has higher bus protocol efficiency and can run at a higher frequency to connect to a memory controller.
- ❖ The high speed memory port will be AMBA3.0 (AXI) protocol compliant, but with reduced I/O requirements.

# Examples

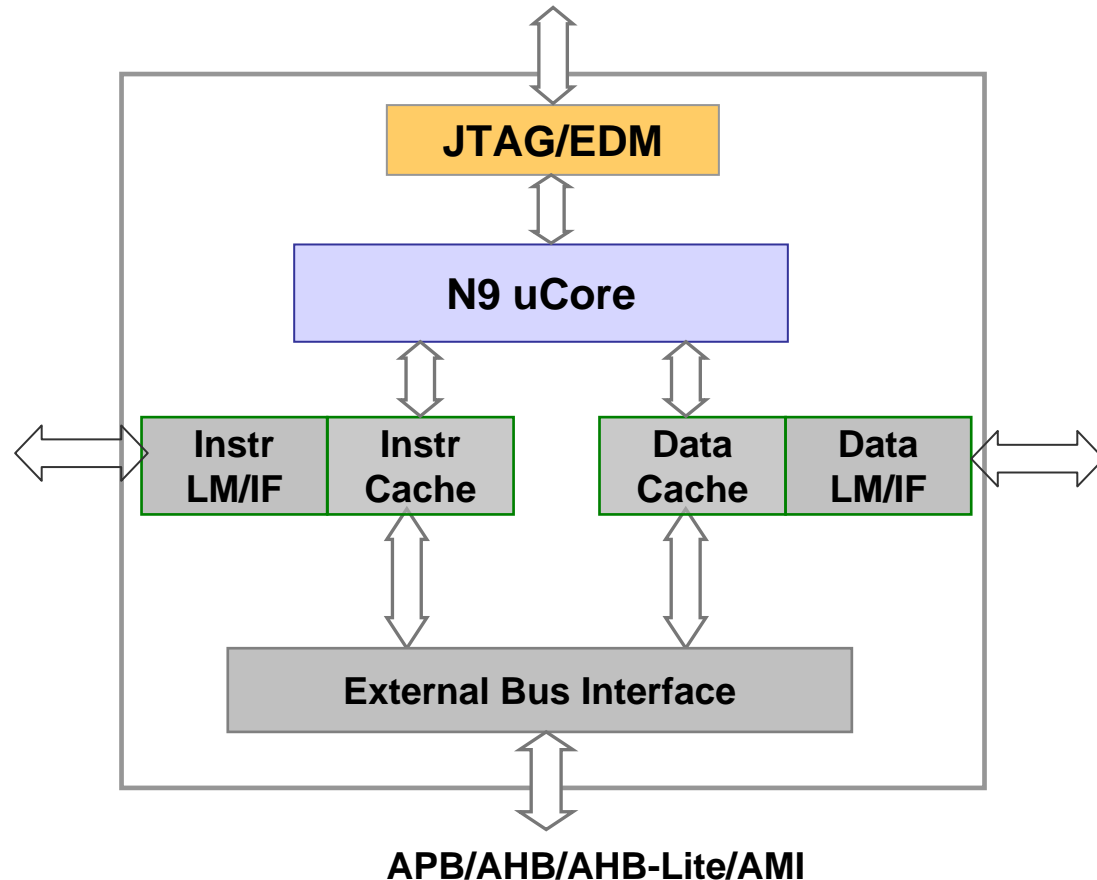
# N903: Low-power Cost-efficient Embedded Controller

## ❖ Features:

- Harvard architecture, 5-stage pipeline.
- 16 general-purpose registers.
- Static branch prediction
- Fast MAC
- Hardware divider
- Fully clock gated pipeline
- 2-level nested interrupt
- External instruction/data local memory interface
- Instruction/data cache
- APB/AHB/AHB-Lite/AMI bus interface
- Power management instructions
- 45K ~ 110K gate count
- 250MHz @ 130nm

## ❖ Applications:

- MCU
- Storage
- Automotive control
- Toys



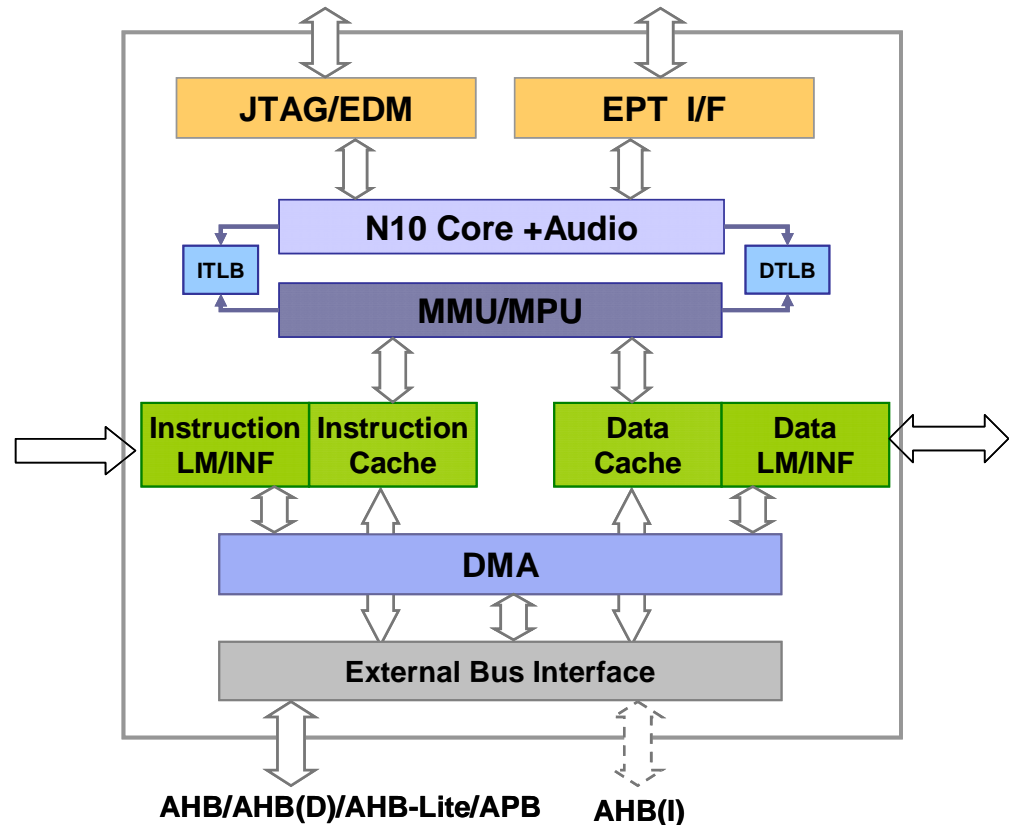
# N1033A: Low-power Cost-efficient Application Processor

## ❖ Features:

- Harvard architecture, 5-stage pipeline.
- 32 general-purpose registers
- Dynamic branch prediction
- Fast MAC
- Hardware divider
- Audio acceleration instructions
- Fully clock gated pipeline
- 3-level nested interrupt
- Instruction/Data local memory
- Instruction/Data cache
- DMA support for 1-D and 2-D transfer
- AHB/AHB-Lite/APB bus
- MMU/MPU
- Power management instructions

## ❖ Applications:

- Portable audio/media player
- DVB/DMB baseband
- DVD
- DSC
- Toys, Games



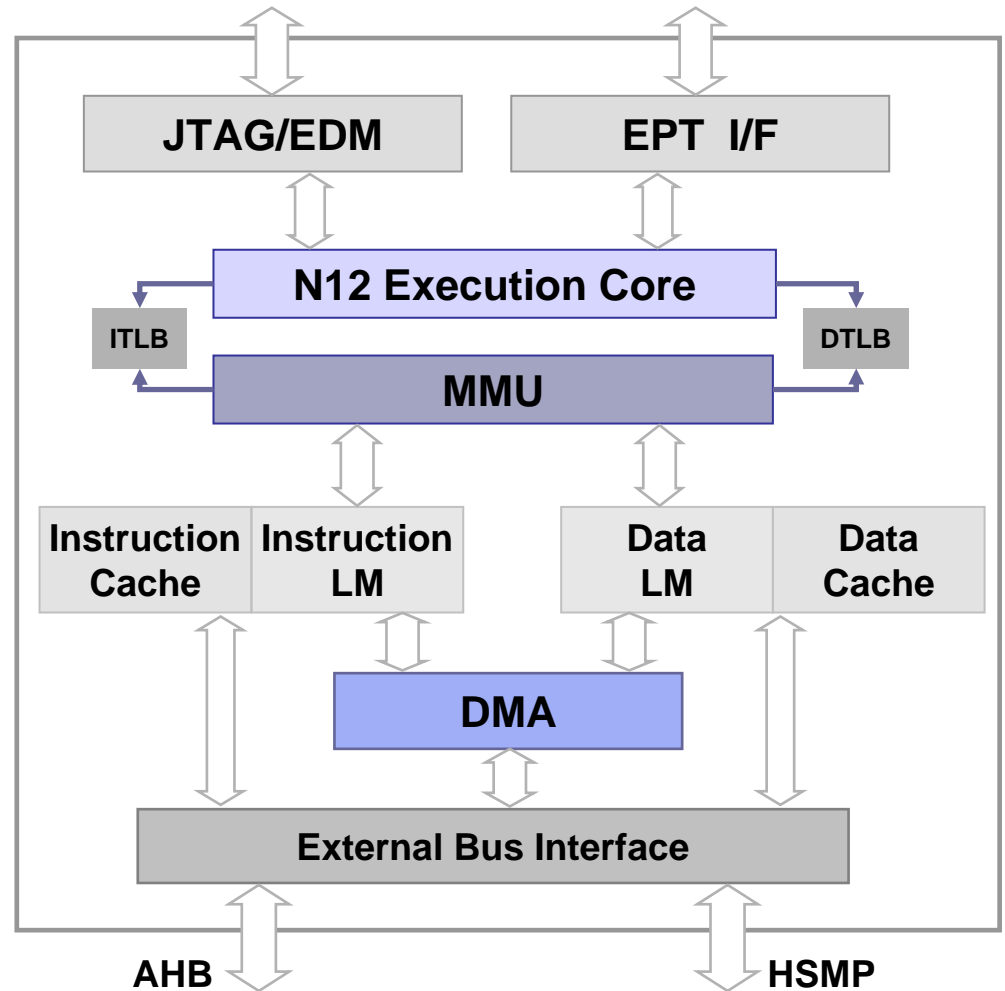
# N1213 – High Performance Application Processor

## ❖ Features:

- Harvard architecture, 8-stage pipeline.
- 32 general-purpose registers
- Dynamic branch prediction.
- Multiply-add and multiply-subtract instructions.
- Divide instructions.
- Instruction/Data local memory.
- Instruction/Data cache.
- MMU
- AHB or HSMP (AXI like) bus
- Power management instructions

## ❖ Applications:

- Portable media player
- MFP
- Networking
- Gateway/Router
- Home entertainment
- Smartphone/Mobile phone



# Thank You

