A Primer on Modern Hardware Architectures

Alessandro Pellegrini
a.pellegrini@ing.uniroma2.it

Università di Roma Tor Vergata
Dicii

A Primer on Modern CPU Organization

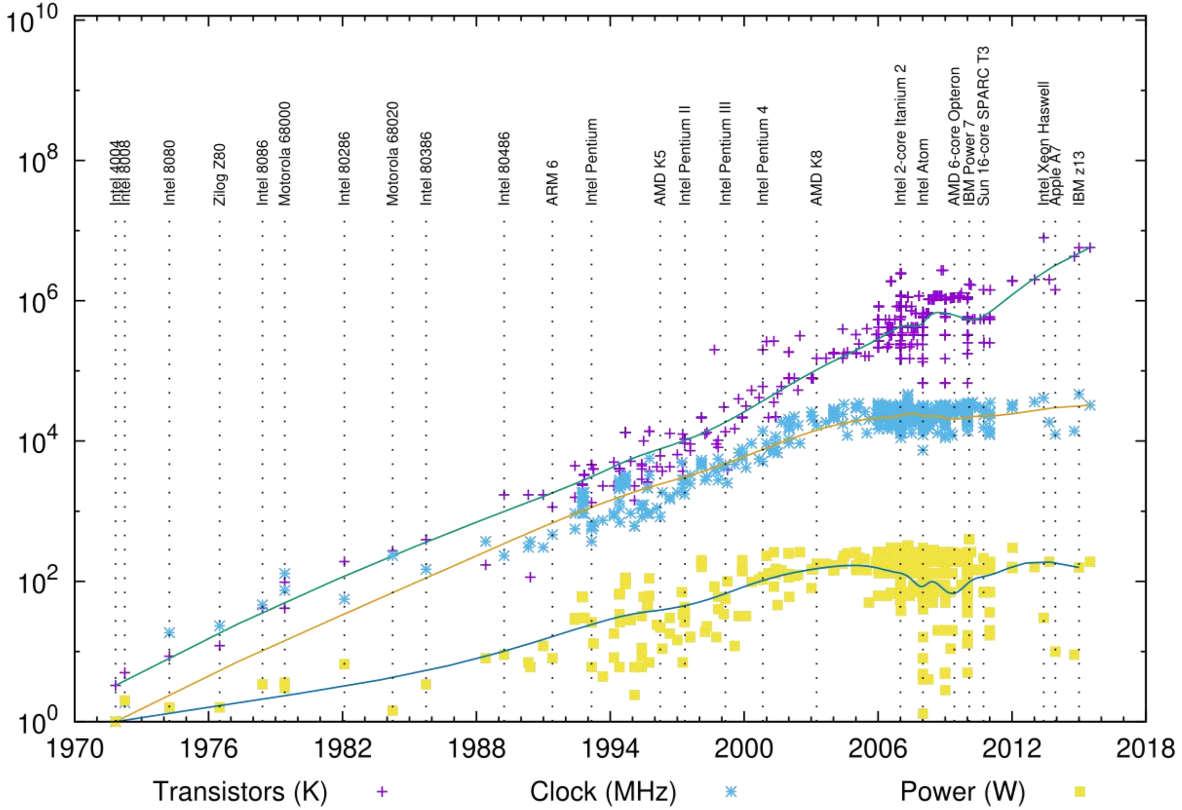
What's going on in your laptop right now?

Moore's Law (1965)

The number of transistors in a dense integrated circuit doubles approximately every two years

— Gordon Moore, Co-founder of Intel

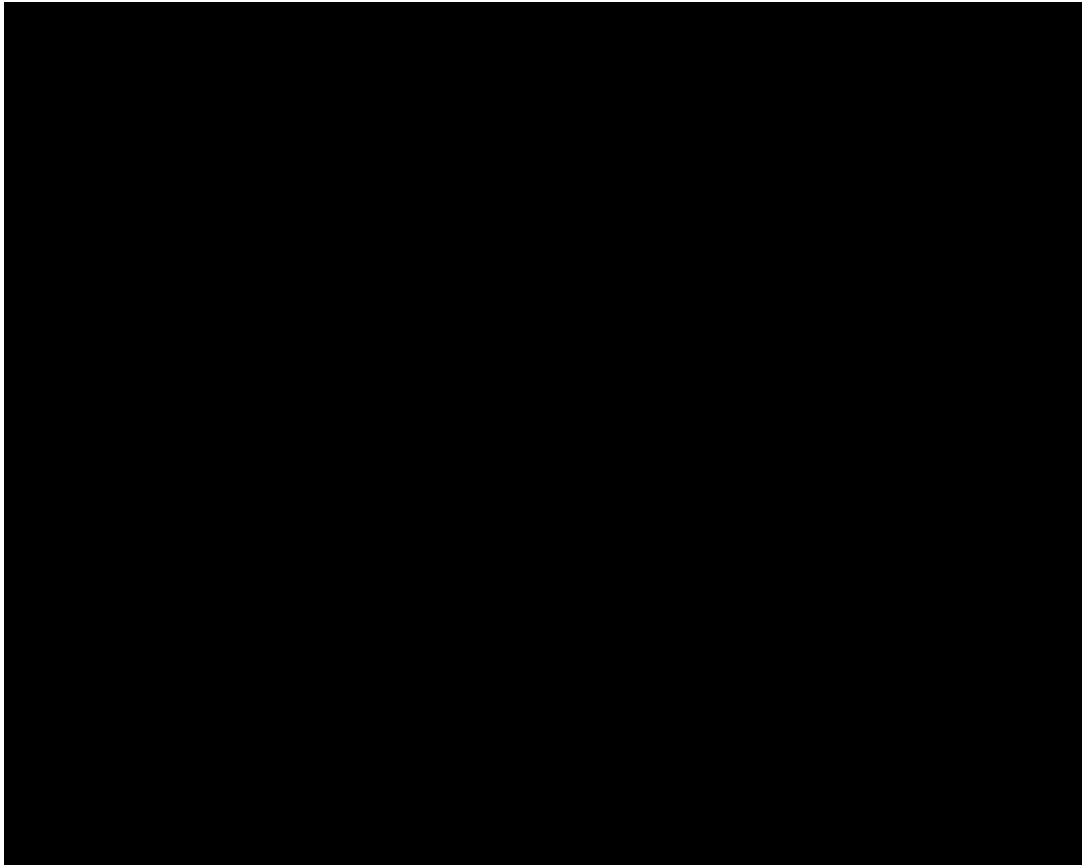
Effects of this Technological Trend



- Implications of Moore's Law have changed since 2003
- 130W is considered an upper bound (the *power wall*)

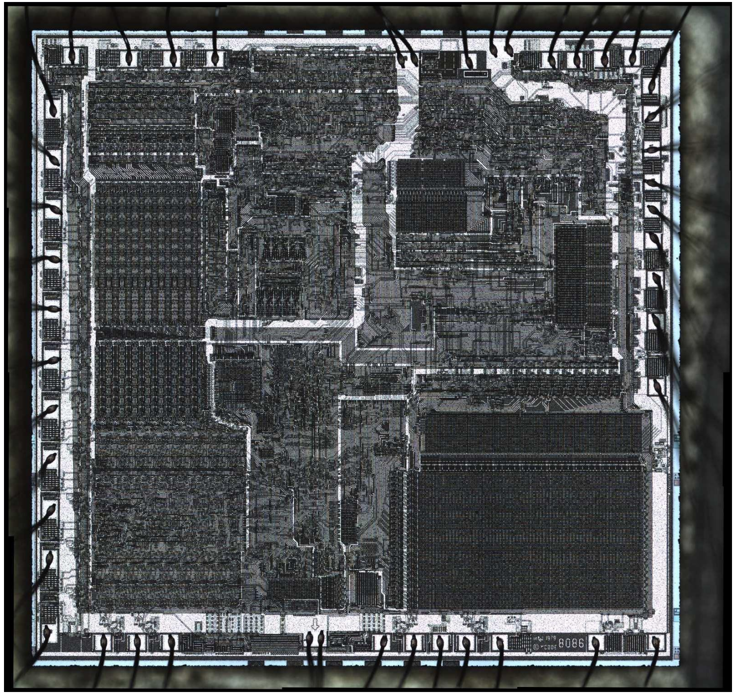
$$P = ACV^2 f$$

Effects of the Power Wall



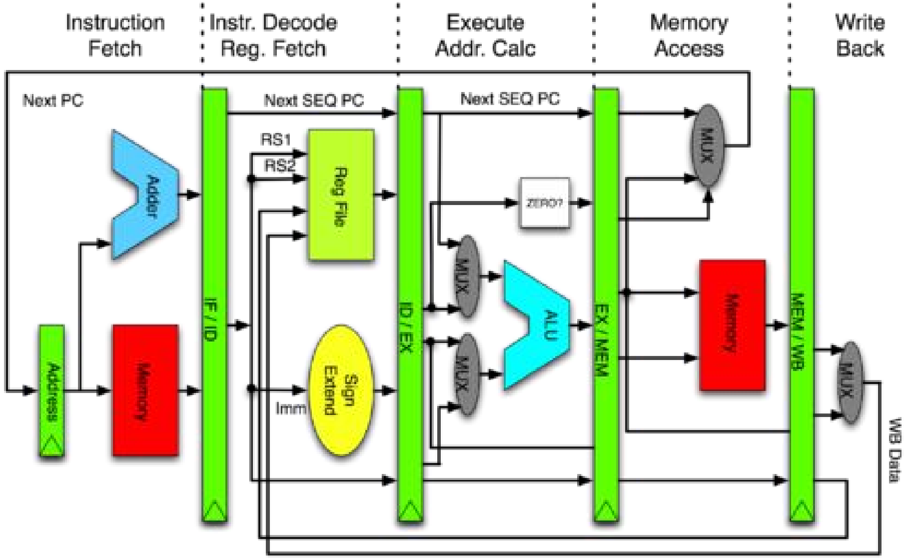
Single Cores

- Multicycle single core CPUs were already complex...
- ...but they were slow!



Trying to speedup: the pipeline (1980s)

- Temporal parallelism
- Number of stages increases with each generation
- Maximum Cycles Per Instructions (CPI)=1



Superascalar Architecture (1990s)

- More instructions are simultaneously executed on the same CPU
- There are redundant functional units that can operate in parallel
- Run-time scheduling (in contrast to compile-time)

ho molti stage doppiati!

IF	ID	EX	MEM	WB				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB
				IF	ID	EX	MEM	WB

Speculation

- In what stage does the CPU fetch the next instruction?
- If the instruction is a conditional branch, when does the CPU know whether the branch is taken or not?
- *Stalling* has a cost: $nCycles \cdot branchFrequency$
- A guess on the outcome of a compare is made
  - if wrong the result is discarded
  - if right the result is flushed

Inizia l'azzardo: invece di aspettare operazioni come if/else, perchè non azzardare la risposta?
se va bene, guadagno tempo. Se va male, flusho la pipeline (la pulisco), rallento. Per questo non posso mettere troppi stage, un conto è pulire 5 istruzioni in // dalla pipeline, un conto è pulirne 10!. E' come se avessi messo la cpu in stallo, ma tanto lo avrei fatto lo stesso per aspettare l'operazione.

cosa c'è in 'a' lo vedo solo con write back a fine pipeline, ma questo risultato mi serve per fare la fetch dell'istruzione dopo. Speculo.

```
a ← b + c
if a ≥ 0 then
    d ← b
else
    d ← c
end if
```

Branch Prediction

voglio massimizzare la probabilità di indovinare la predizione!

- Performance improvement depends on:
  - whether the prediction is correct
  - how soon you can check the prediction
- Dynamic branch prediction
  - the prediction changes as the program behaviour changes
  - implemented in hardware
  - commonly based on branch history
    - predict the branch as taken is it was taken previously
- Static branch prediction
  - compiler-determined
  - user-assisted (e.g., `likely` in kernel's source code; `0x2e`, `0x3e` prefixes for Pentium 4)

Branch Prediction Table

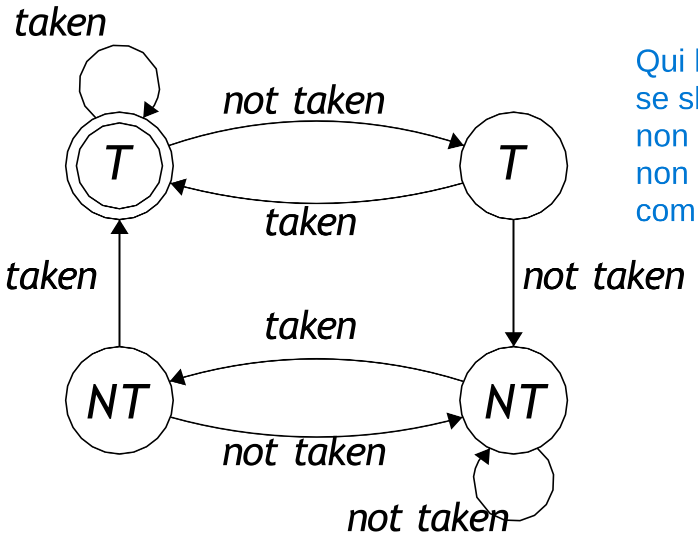
- Small memory indexed by the lower bits of the address of conditional branch instruction
- Each instruction is associated with a prediction
  - *Take* or *not take* the branch
- If the prediction is *take* and it is correct:
  - Only one cycle penalty
- If the prediction is *not take* and it is correct:
  - No penalty
- If the prediction is *incorrect*:
  - Change the prediction
  - Flush the pipeline
  - Penalty is the same as if there were no branch prediction

Se indovino, perchè ho un ciclo di penalità?
Perchè in automatico si carica sempre l'istruzione dopo (FETCH), ma io in DECODE (fase dopo) vedo che c'è il branch. Allora se salto devo togliere " l'istruzione dopo" messa prima, e far posto all'indirizzo di salto.

Per i motivi qui sopra, se invece non devo saltare ho caricato l'istruzione giusta.

Two-bit Saturating Counter

Dipende dagli eventi che occorrono, a differenza di usare 1 byte (2 stati), qui ne uso 2 (4 stati). Questo perchè qui c'è il concetto di "rinforzo della predizione". Non è detto che, se sbaglio una predizione, allora con la stessa predizione non posso aver ragione dopo.
(Se io prevedo X, esce Y. Dopo potrebbe uscire veramente X, e quindi in questo secondo caso avrei ragione, e non dovrei cambiare.



Qui ho T "rinforzato" (il primo T), se sbaglio predizione vado in T " non rinforzato", e a meno che non sbaglio nuovamente farò comunque T.

- How well does it work with nested loops?

A Nested Loop Example

```
        mov $0, %ecx
.outerLoop:
        cmp $10, %ecx
        je .done
        mov $0, %ebx

.innerLoop:
        ; actual code
        inc %ebx
        cmp $10, %ebx
        jnz .innerLoop

        inc %ecx
        jmp .outerLoop
.done:
```

metto in ecx il valore 0.

ciclo esterno:
confronto 10 con ecx (0 all'inizio), se uguali esco dal ciclo maggiore e finisco.
Altrimenti metto 0 in ebx (il counter del ciclo interno).

Qui dentro prima faccio qualcosa ("actual code"),
incremento ebx,
faccio la compare,
se non arrivo a 10 iterazioni rivado sopra, altrimenti incremento il counter del ciclo esterno e vado nel ciclo esterno.

A Nested Loop Example

(sarebbe il caso T - NT solamente)

- The 2-bit saturating counter does not work well with nested loops
- The outer loop suffers from the inner loop (~9% misprediction)

```
            mov $0, %ecx
.outerLoop:
            cmp $10, %ecx
            je .done            n-1 mispredictions
            mov $0, %ebx

.innerLoop:
            ; actual code
            inc %ebx
            cmp $10, %ebx
            jnz .innerLoop

            inc %ecx
            jmp .outerLoop
.done:
```

io partirei da T normale,
dovrei fare il salto "je .done", però ancora non ho iniziato, e quindi avrei sbagliato la predizione.
Dovrei mettere l'automa in NT.
Però quando vado in avanti in .InnerLoop, e trovo " jnz .innerLoop", secondo l'automa NON dovrei saltare (perchè starei in NT), ma qui devo saltare, quindi ho sbagliato due volte!

Se avessi usato l'automa a 4 bit, sarei passato da "T rinforzato" a "T non rinforzato", ed avrei sbagliato solo una volta!

osservazione: questi due salti non hanno nulla in comune, cioè non posso escludere nulla a priori! Per questo uso due bit, per l'indipendenza.

Branch Prediction is Important

- Conditional branches are around 20% of the instructions in the code
- Pipelines are deeper
  - A greater misprediction penalty
- Superscalar architectures execute more instructions at once
  - The probability of finding a branch in the pipeline is higher
- Object-oriented programming
  - Inheritance adds more branches which are harder to predict

- Two-bits prediction is not enough
  - Chips are denser: more sophisticated hardware solutions could be put in place

How to Improve Branch Prediction?

- Improve the prediction
  - Correlated (two-levels) predictors [Pentium]
  - Hybrid local/global predictor [Alpha]

- Determine the target earlier
  - Branch target buffer [Pentium, Itanium]
  - Next address in instruction cache [Alpha, UltraSPARC]
  - Return address stack [Consolidated into all architecture]

- Reduce misprediction penalty
  - Fetch both instruction streams [IBM mainframes]

Correlated Predictor

- Predictions cannot depend on only one branch
- Some branch outcomes are correlated

qui un if esclude l'altro!

```
if (d == 0)
...
if (d != 0)
```

```
if (d == 0)
    b = 1;
...
if (b == 1)
```

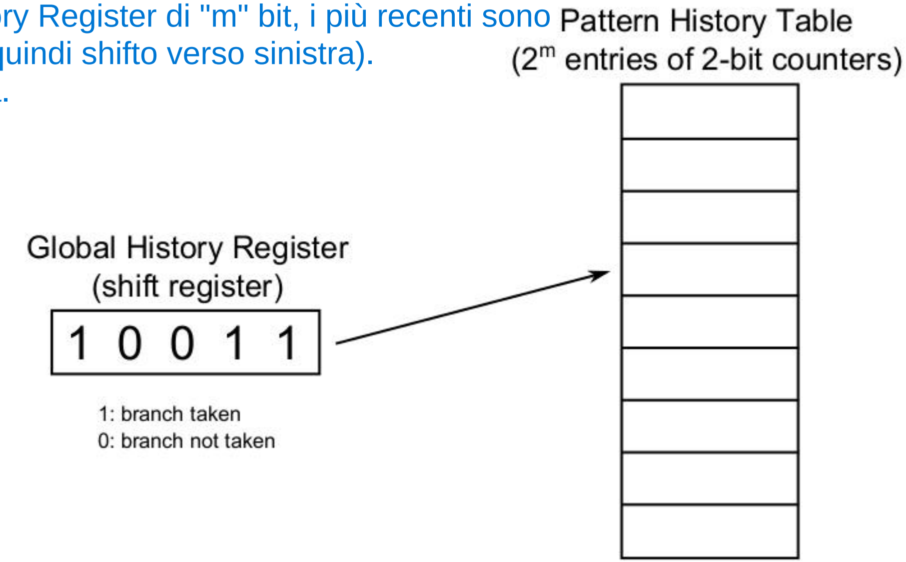
```
if (x == 2)
    x = 0;
if (y == 1)
    y = 0;
if (x != y)
```

- Use a *history* of past m branches, representing a path through the program

Pattern History Table (PHT)

- Put the global branch history in a global history register
- Use its value to access a PHT of 2-bit saturating counters

uso più counter a 2 bit, cioè per ogni entry ho uno stato dell'automa visto prima.
E' presente anche il Global History Register di "m" bit, i più recenti sono a destra, i più vecchi a sinistra (quindi shifto verso sinistra).
Mediante GHR punto alla tabella.



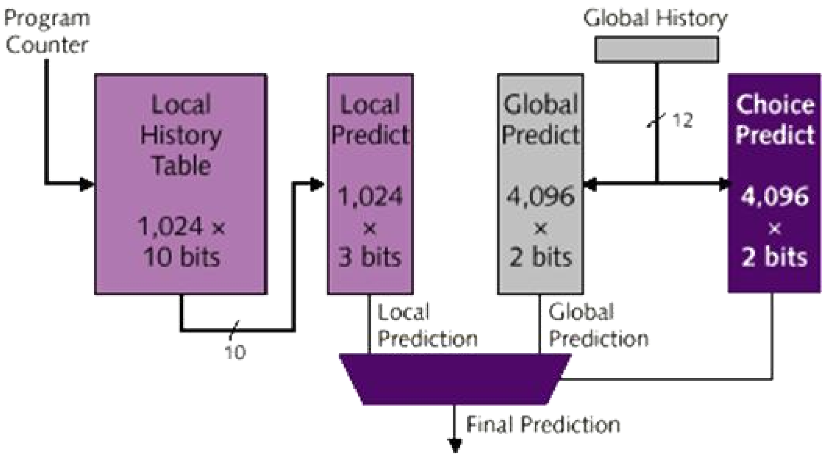
Questo predittore è fortemente orientato alle predizioni, quindi cerca di forzarle, con risultati non ottimali.

Tournament Predictor

- Combines different branch predictors
- A *local predictor*, selected using the branch address
- A *correlated predictor*, based on the last *m* branches, accessed by the local history
- An indicator of which has been the best predictor for this branch
  - A 2-bit counter, which is increased for one and decreased for the other

Confronto due predittori: uno basato sul recente passato (correlato) ed uno basato sulla singola istruzione (scorrelato)

DEC Alpha 21264 BPU

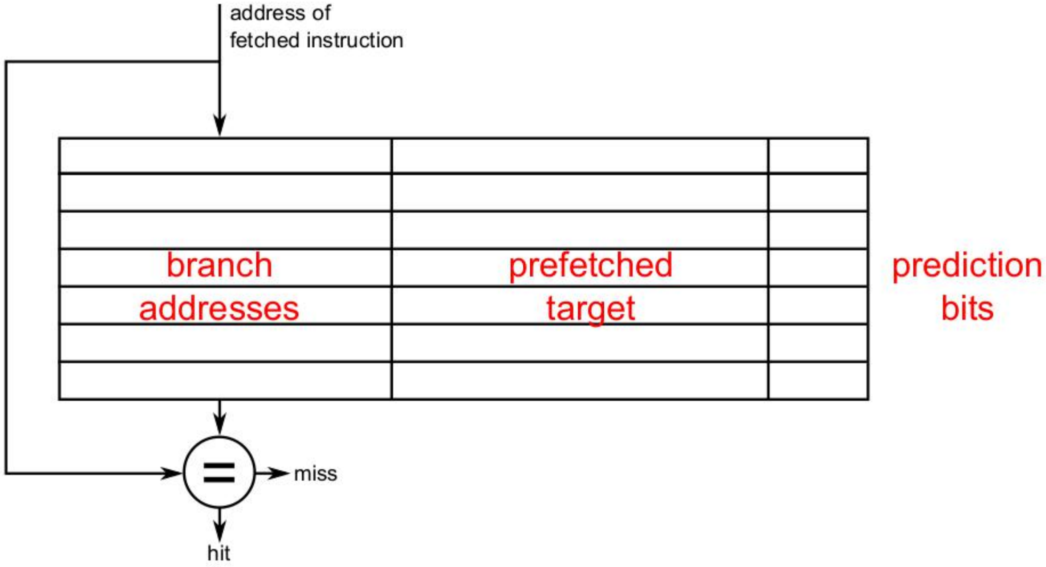


- Tournament Branch Prediction Algorithm
- 35Kb of prediction information
- 2% of total die size
- Claim 0.7-1.0% misprediction

molto buono, ma richiede
memoria e tempo nei vecchi calcolatori.

Branch Target Buffer

- Indirect jumps are hard to predict (e.g., `jmp *rax, ret`)
- It is a small cache memory, accessed via PC during the fetch phase
- Prediction bits are coupled with prediction target



Return Address Stack

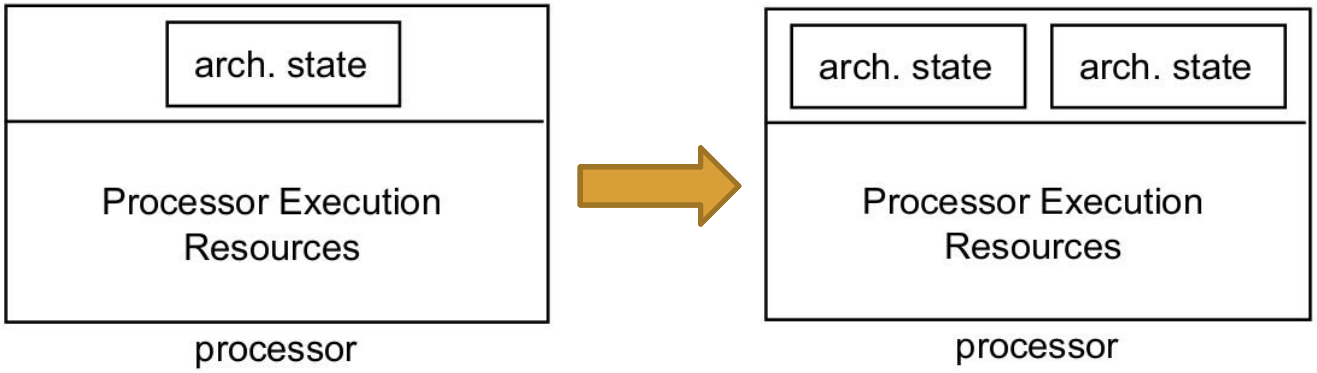
- Registers are accessed several stages after instruction's fetch
- Most of indirect jumps (85%) are function-call returns
- Return address stack:
  - it provides the return address early
  - this is pushed on call, popped on return
  - works great for procedures that are called from multiple sites
    - BTB would would predict the address of the return from the last call

Fetch Both Targets

- This technique fetches both targets of the branch
- Does not help in case of multiple targets (e.g., `switch` statements)
- Reduces the misprediction penalty
- Requires a lot of I-cache bandwidth

Simultaneous Multi-Threading—SMT (2000s)

- A physical processor appears as multiple logical processors
- There is a copy of the architecture state (e.g., control registers) for each logical processor
- A single set of physical execution resources is shared among logical processors
- Requires less hardware, but some sort of arbitration is mandatory



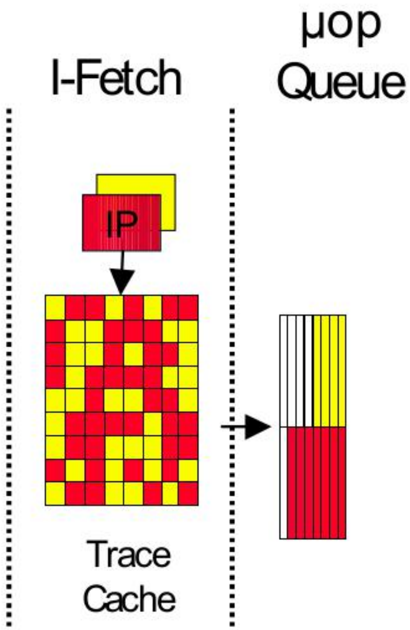
The Intel case: Hyper-Threading on Xeon CPUs

- **Goal 1**: minimize the die area cost (share of the majority of architecture resources)
- **Goal 2**: when one logical processor stalls, the other logical process can progress
- **Goal 3**: in case the feature is not needed, incur in no cost penalty
- The architecture is divided into two main parts:
  - Front end
  - Out-of-order execution engine

Xeon Front End

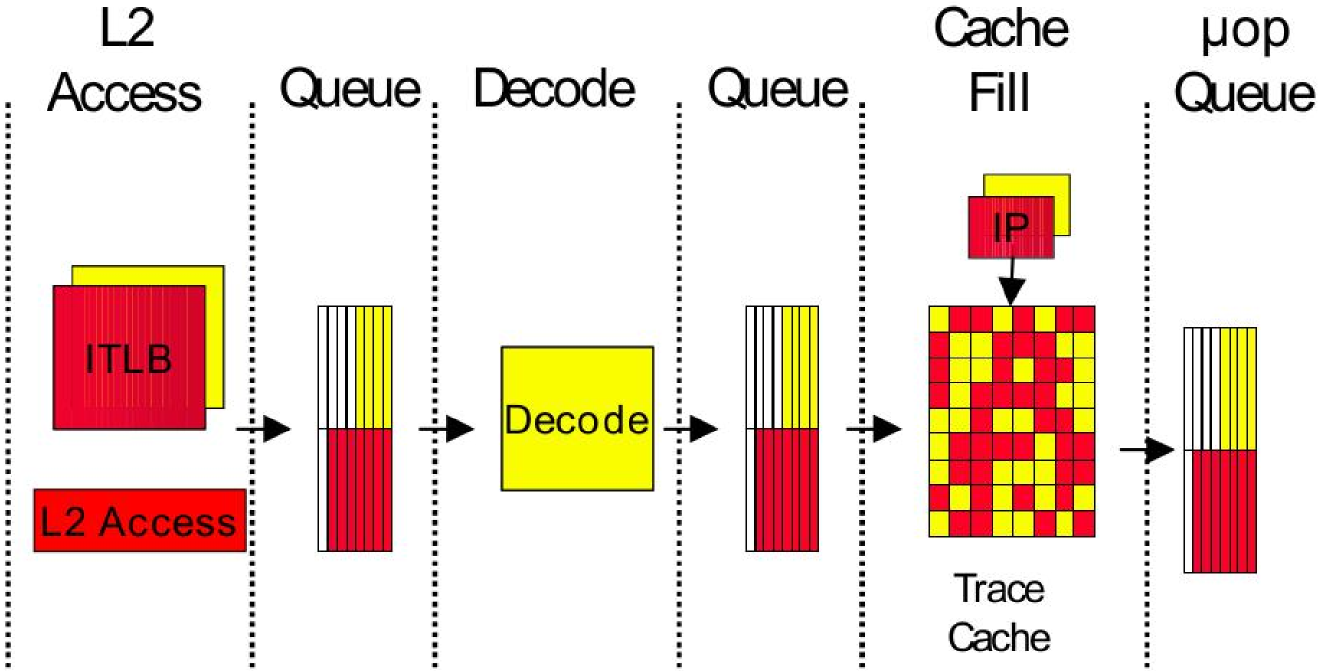
- The goal of this stage is to deliver instruction to later pipeline
- stages
- Actual Intel's cores do not execute CISC instructions
  - Intel instructions are cumbersome to decode: variable length, many different options
  - A Microcode ROM decodes instructions and converts them into a set of semantically-equivalent RISC μ-ops
- μ-ops are cached into the Execution Trace Cache (TC)
- Most of the executed instructions come from the TC

Trace Cache Hit

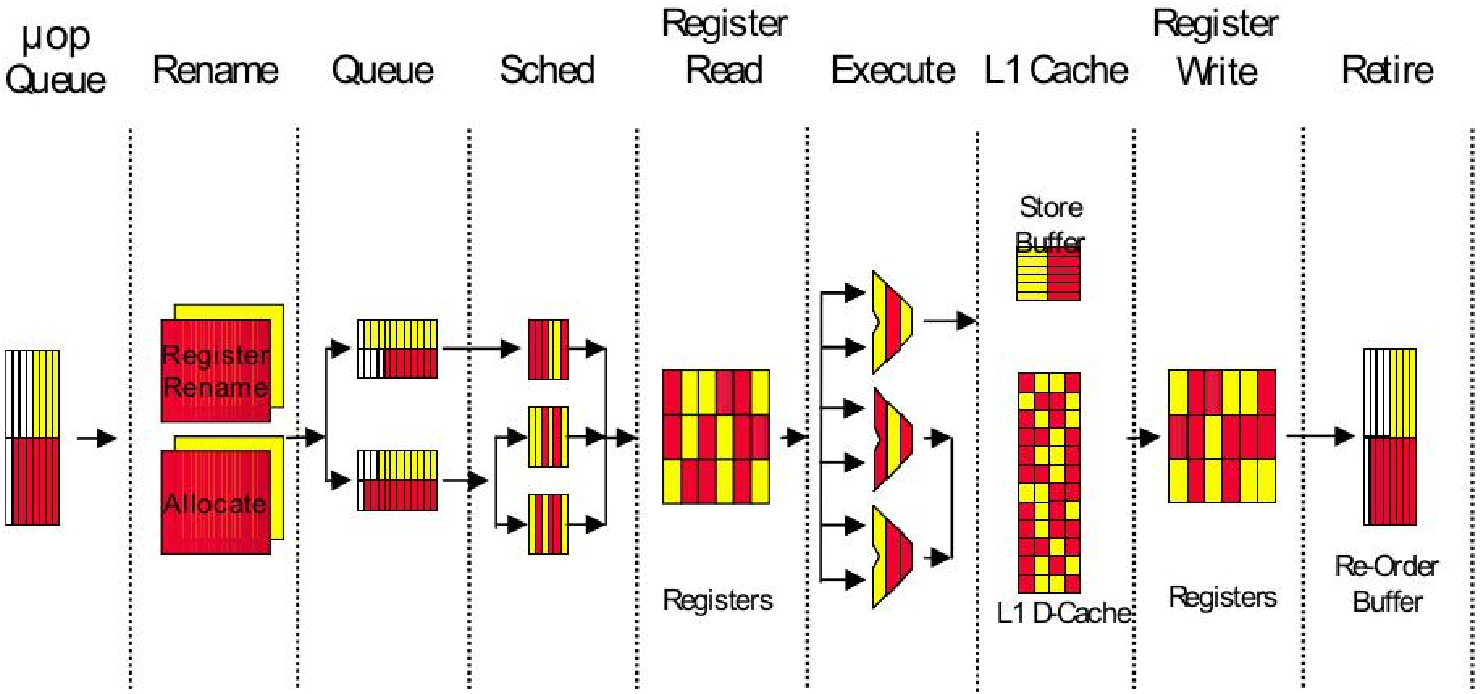


- Two sets of next-instruction pointers
- Access to the TC is arbitrated among logical processors at each clock cycle
- In case of contention, access is alternated
- TC entries are tagged with thread information
- TC is 8-way associative, entries are replaced according to a LRU scheme

Trace Cache Miss



Xeon Out-of-order Pipeline

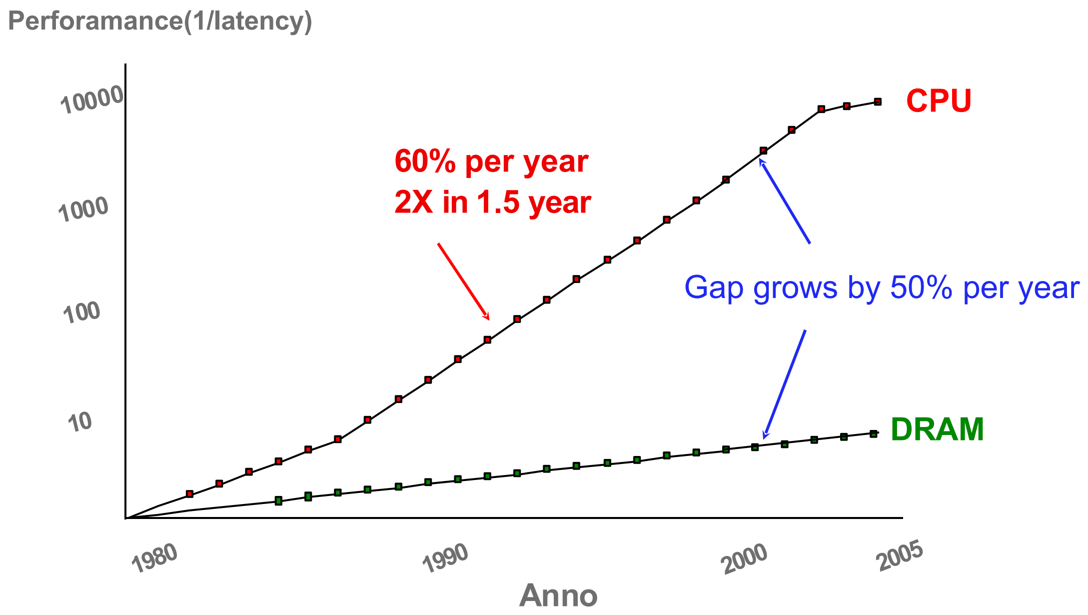


A Primer on Modern Cache Hierarchy

How does your CPU manages data?

CPU/Memory Performance Gap

- The performance gap between memory and CPU has grown over time
- How can this difference be managed?

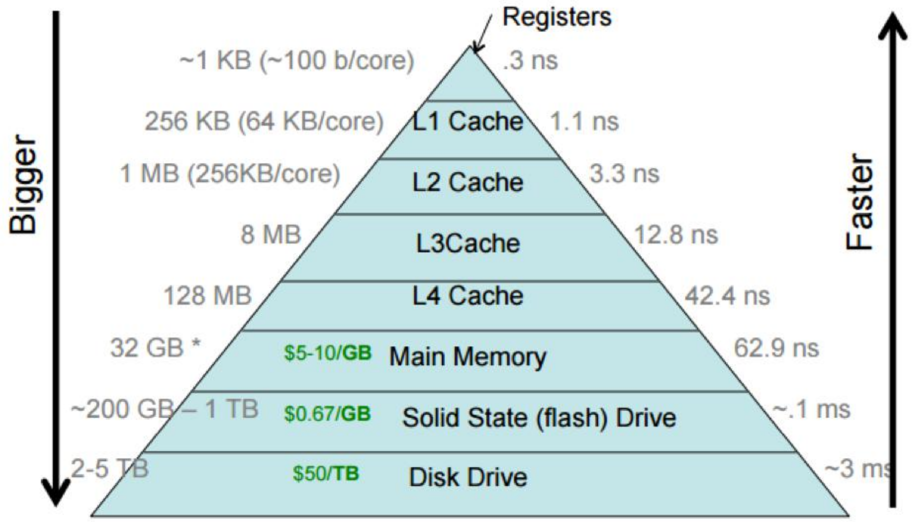


Memory Hierarchy

- The optimal solution for a memory system is:
  - minimum cost
  - maximum capacity
  - minimum access time
- An approximate solution is to implement a memory hierarchy:
  - different technologies to best meet each of the requirements
  - the hierarchy seeks to optimise parameters globally

Memory Hierarchy

- More memory levels
- As the distance from the CPU increases, the cost decreases and the storage capacity increases
- At each instant of time, data is only copied between each pair of adjacent layers
    - We can focus on two layers only



Cache Usage Strategies

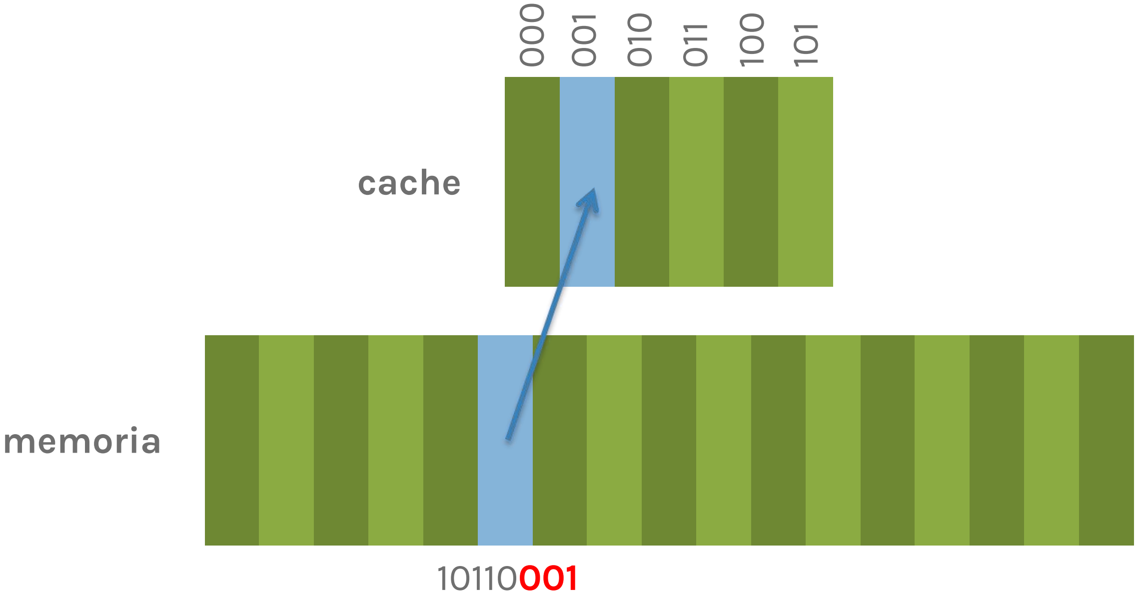
- The cache structured in **lines**
  - Each line contains one **block** (typically 64 bytes), which is the *smallest unit of information* that is transferred between levels
- The processor only interacts with the **first-level cache controller**
- If the requested data is in the first-level cache, a **cache hit** occurs
- If the requested data is not present, a **cache miss** occurs
  - The L1 level controller "requests" the data from the L2 level controller
  - Either a hit or a miss may occur at L2 level
  - If there is a hit in L2, the data is copied to L1
  - If there is a miss in L2, the L2 level controller "requests" the data from the L3 level controller
  - The same pattern occurs for RAM access

Inclusiveness Policies in Caches

- **Inclusive** cache:
  - An upper level of the hierarchy (closest to the processor) contains a subset of information from the lower levels
  - All information is stored in the lowest level
  - Only the highest cache level (L1 cache) is accessed directly by the processor
- **Non-inclusive** (or exclusive) caches:
  - Some levels may be non-inclusive
  - A miss at a higher level (e.g. L1), may result in direct access in memory
  - In this case, the data is written only in L1, but not in L2
  - If a block must be replaced to write to L1, it "falls" into L2

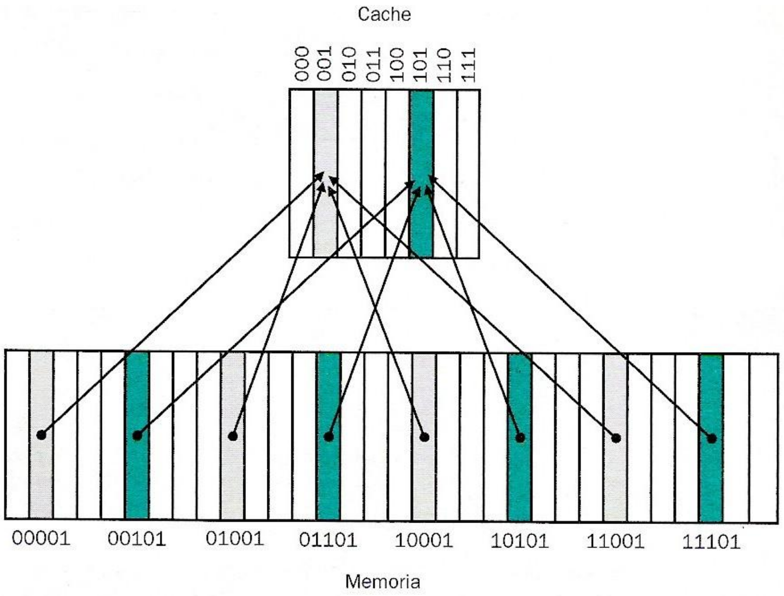
Determining Block Position

- A cache is smaller than memory
- How do we decide in which line to store a block?
- **Direct mapping**: *a hash function based on extraction* is used:
  - example: take n least significant bits of the block address
- It works because caches have a 2^nsize



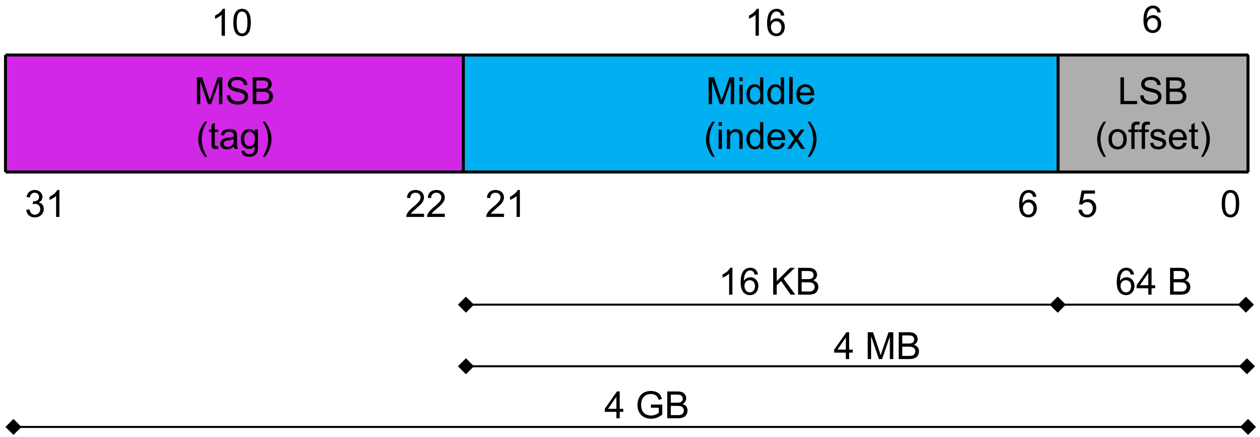
Cache line collision

- More memory words are associated with a single cache position (*synonims*)
- How can we know if the data in a cache line corresponds to the requested word?
- We can use a **tag** to detect collisions

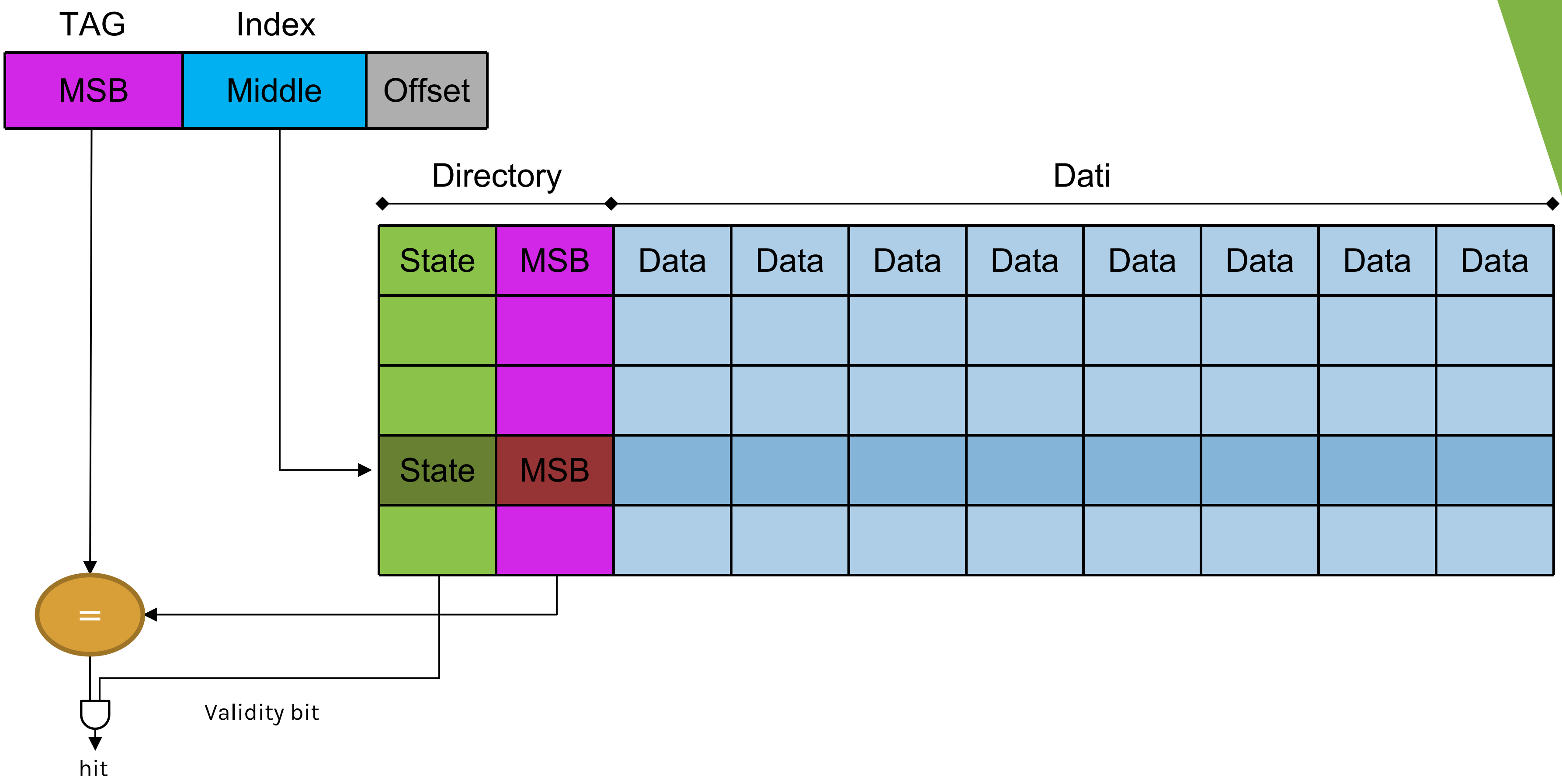


Direct Mapped Cache

- The memory address is split in three parts
  - Tag: the most significant bits, to discriminate *synonims*
  - Index: intermediate bits, to select a cache line
  - Offset: the least significant bits, to select a word in the block



Direct Mapped Cache



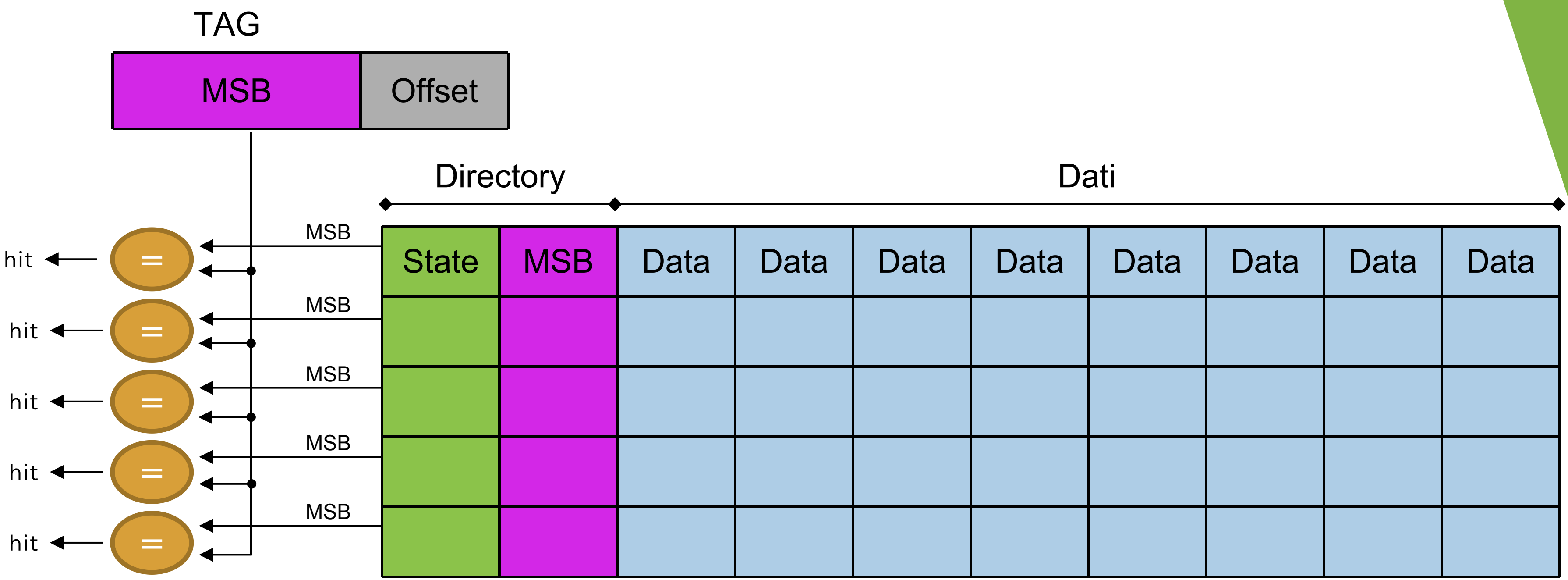
Fully-Associative Cache

- In a *fully-associative* cache, a block can be stored in *any* cache line

- **Pros**:
    - maximum utilization of the cache
    - conflict reduction

- **Cons**:
    - To find a block, we must search the entire cache
    - Sequential search would increase the latency too much
    - We must perform a *paralle search*: we need a comparator for each cache line
    - The cost is high

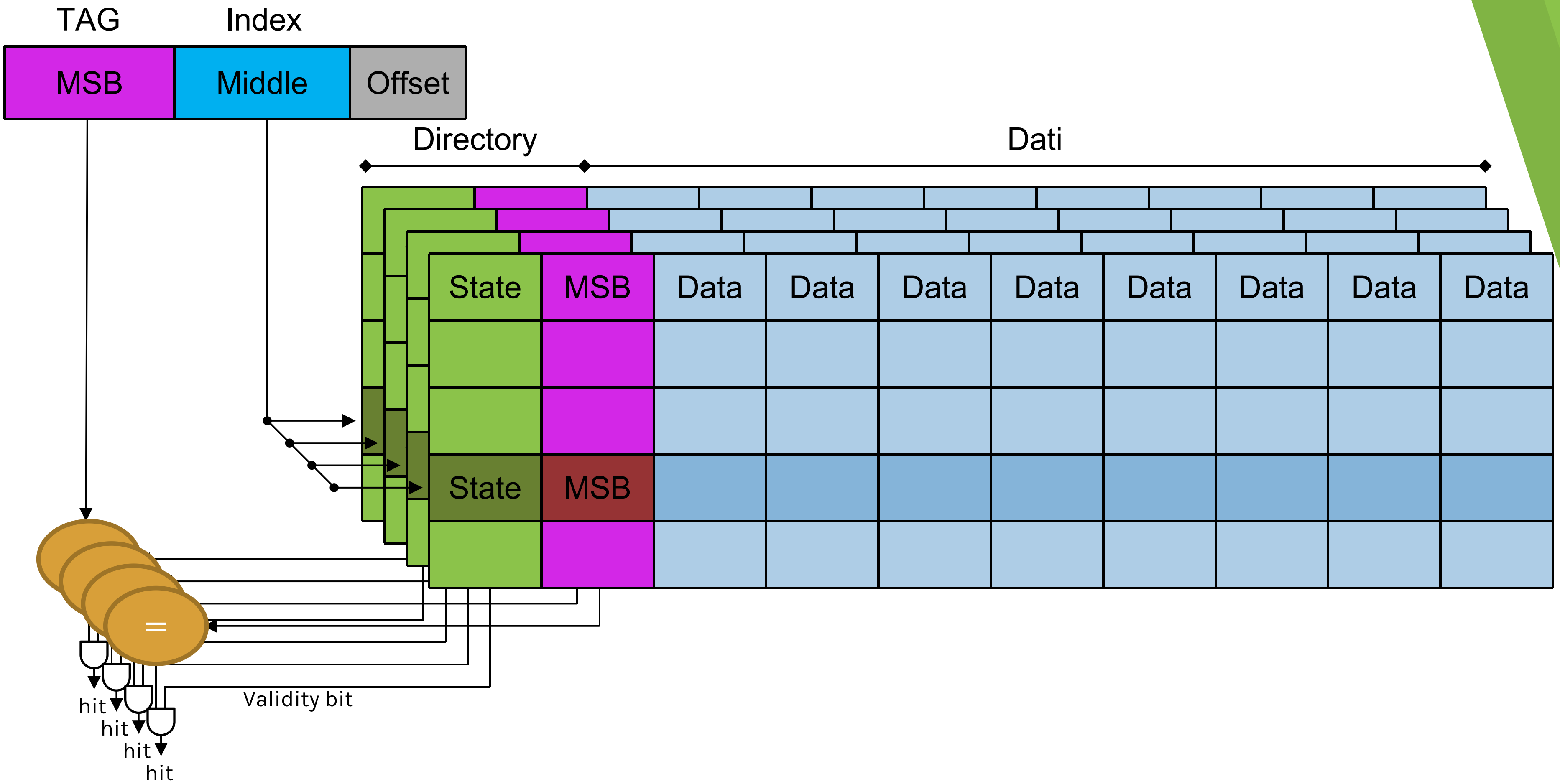
Fully Associative Cache: read access



Set-Associative Cache

- In a direct-mapped cache the number of conflicts can be high
- Fully-associative caches have a better utilization factor, but the cost is too high
- Set-associative (or *n-way set associative*) are a good tradeoff:
  - each block can be stored in a number of lines $n \geq 2$ (a *set*)
  - the search for a block requires the comparison of at most n tags
  - the cost is acceptable
  - it is the most used technology in modern processors

4-way Set Associative Cache: read access



Eviction Policies

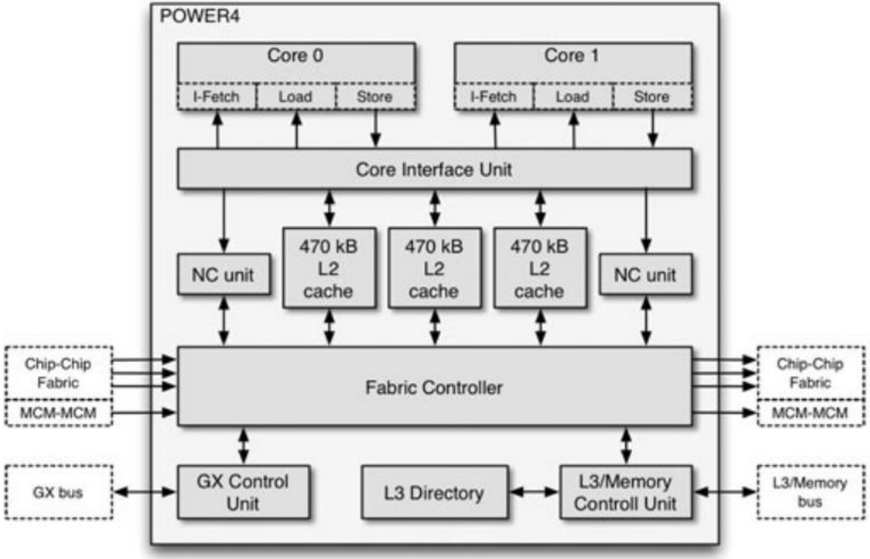
- When more than a cache entry is available (fully-associative and set-associative), we must select a *victim* for replacement
- Most common policies:
  - **Least Recently Used** (LRU): the directory keeps an *age counter*: it is reset when a block is loaded, upon each access the counters in the set are incremented by one
  - **First-in First-out** (FIFO): similar to LRU, but the increment occurs only upon replacement
  - **Least Frequently Used** (LFU): hard to implement
  - **Round Robin**: used in fully-associative caches. It uses a pointer to keep track of the next line to replace. It is updated upon each replacement.
  - **Random**: as round robin, but the pointer is updated upon any access

Write Operations Management

- The CPU interacts only with the L1 cache
- Caches keep *copies* of the data
- When a block is updated, we must keep consistent the other copies
  - We cannot write to memory is the data are not loaded into cache!
- Two main strategies:
  - **Write through**: upon each write to L1, the copies in the lower levels are immediately updated
    - Easy to implement
    - The performance is significantly lower
  - **Write back**: lower-level copies are updated only upon a replacement
    - More complex to implement
    - Much much better performance

Multicores (2000s)

- First multicore chip: IBM Power4 (1996)
- 1.3 GHz dual-core PowerPC-based CPU



- How do they access the data?

Cache Coherence (CC)

- CC defines the correct behaviour of caches, *regardless of how they are employed by the rest of the system*
- Typically programmers don't see caches, but caches are usually part of shared-memory subsystems

t	**Core C1**	**Core C2**	*C1 Cache*	*C2 Cache*
0	load r1, A		A: 42	
1		load r1, A	A: 42	A: 42
2	add r1, r1, $1			
	store r1, A		A: 43	A: ??

- What is the value of A in C2?

Strong Coherence

- Most intuitive notion of CC is that cores are *cache-oblivious*:
  - All cores see at any time the same data for a particular memory address, *as they should have if there were no caches in the system*
- Alternative definition:
  - All memory read/write operations to a single location A (coming from all cores) must be executed in a *total order* that respects the order in which each core commits its own memory operations

Strong Coherence

- A sufficient condition for strong coherence is jointly given by implementing two *invariants*:
  1. Single-Writer/Multiple-Readers (SWMR)
     - For any memory location A, at any given epoch, either a single core may read and write to A <u>or</u> some number of cores may only read A
  2. Data-Value (DV)
     - The value of a memory location at the start of an epoch is the same as its value at the end of its latest read-write epoch

Weaker Coherence

- Weaker forms of coherence may exist for performance purposes
  - Caches can respond faster to memory read/write requests
- The SWMR invariant might be completely dropped
  - Multiple cores might write to the same location A
  - One core might read A while another core is writing to it
- The DV invariant might hold only *eventually*
  - Stores are guaranteed to propagate to all cores in d epochs
  - Loads see the last value written only after *d* epochs
- The effects of weak coherency are usually visible to programmers
  - Might affect the memory consistency model (see later)

No Coherence

- The fastest cache is *non-coherent*
  - All read/write operations by all cores can occur simultaneously
  - No guarantees on the value observed by a read operation
  - No guarantees on the propagation of values from write operations
- Programmers must explicitly coordinate caches across cores
  - Explicit invocation of coherency requests via C/Assembly APIs

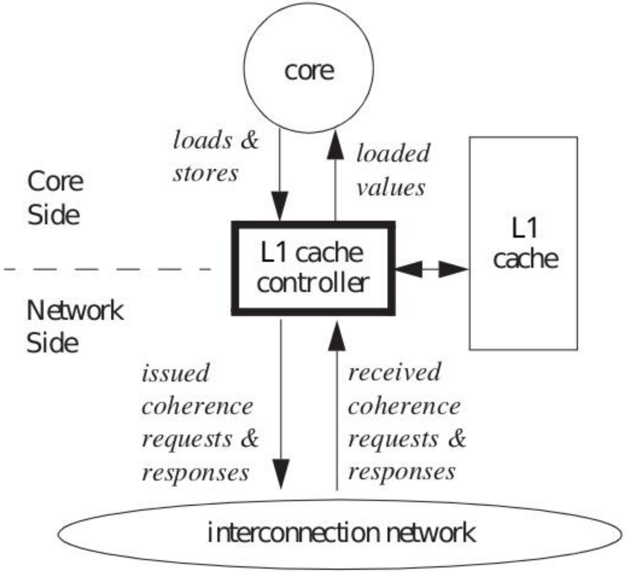
CC Protocols

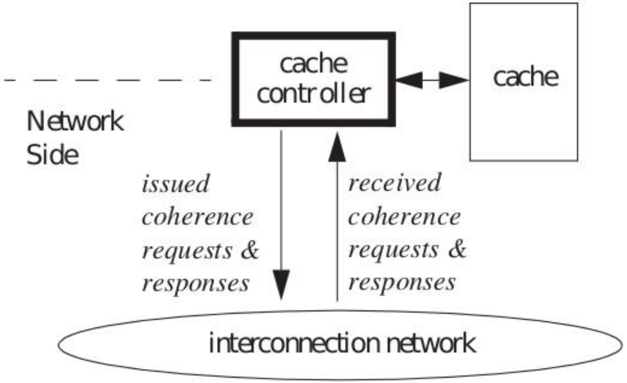
- A CC protocol is a distributed algorithm in a message-passing distributed system model
- It serves two main kinds of memory requests
  - *Load(A)* to read the value of memory location A
  - *Store(A, v)* to write the value v into memory location A
- Itinvolves two main kinds of actors
  - Cache controllers (i.e., L1, L2, ..., LLC)
  - Memory controllers
- It enforces a given notion of coherence
  - Strong, weak, no coherence

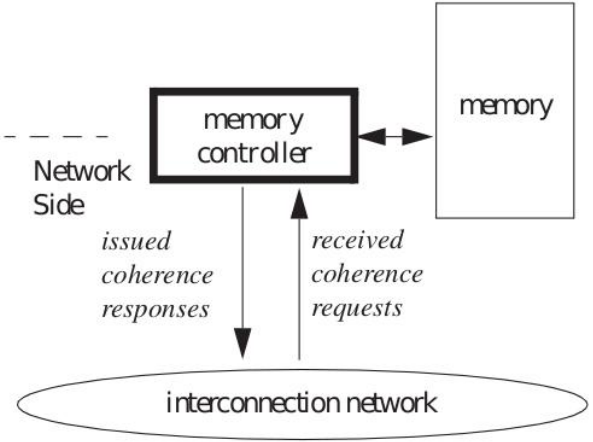
Coherency Transactions

- A memory request may traduce into some *coherency transactions* and produce the exchange of multiple *coherence messages*
- There are two main kinds of coherency transactions:
  - *Get*: Load a cache block b into cache line l
  - *Put*: Evict a cache block b out of cache line l

Cache and Memory Controllers







Finite-State Machines

- Cache controllers manipulate local finite-state machines (FSMs)
- A single FSM describes the state of a copy of a block (<u>not</u> the block itself)
- States:
  - Stable states, observed at the beginning/end of a transaction
  - Transient states, observed in the midst of a transaction
- Events:
  - Remote events, representing the reception of a coherency message
  - Local events, issued by the parent cache controller
- Actions:
  - Remote action, producing the sending of a coherency message
  - Local actions, only visible to the parent cache controller

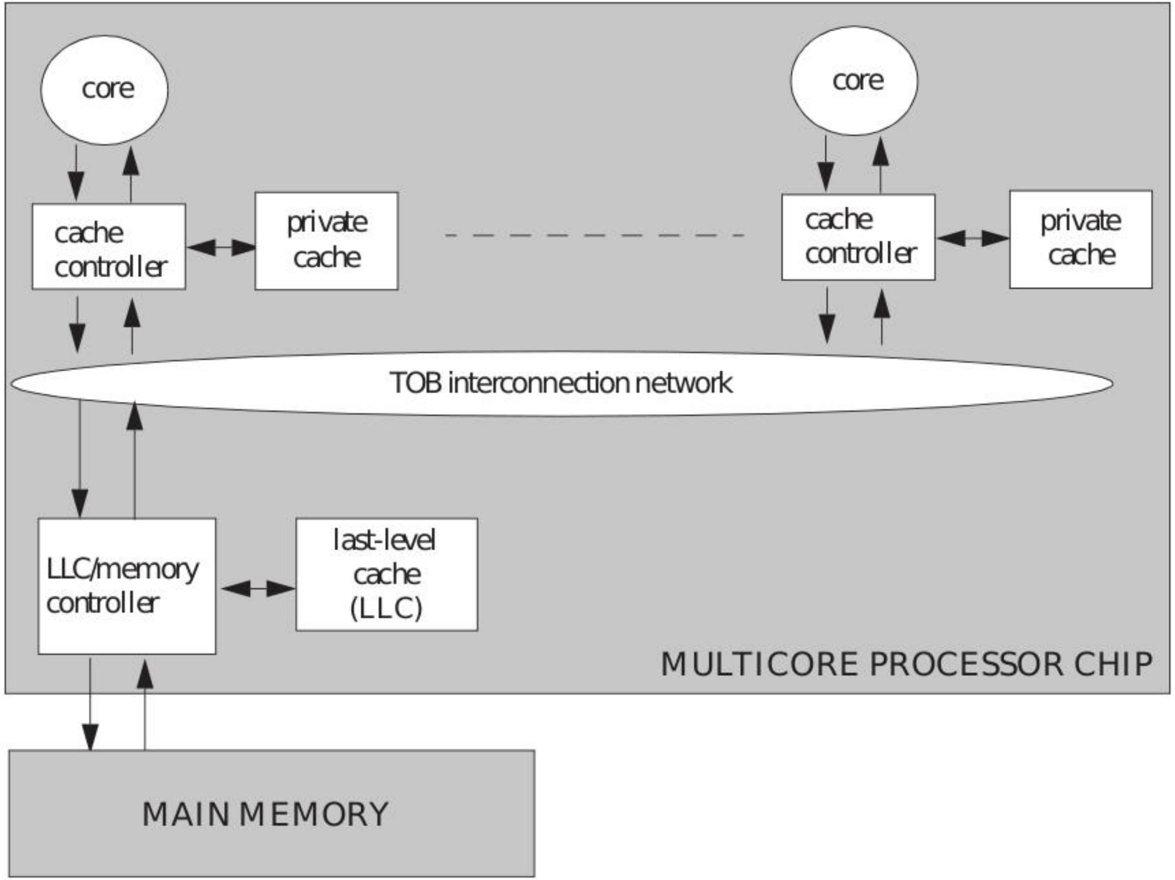
Families of Coherence Protocols

- **Invalidate protocols**:
  - When a core writes to a block, all other copies are invalidated
  - Only the writer has an up-to-date copy of the block
  - Trades latency for bandwidth
- **Update protocols:**
  - When a core writes to a block, it updates all other copies
  - All cores have an up-to-date copy of the block
  - Trades bandwidth for latency

Families of Coherence Protocols

- **Snooping Cache:**
  - Coherence requests for a block are broadcast to all controllers
  - Require an interconnection layer which can total-order requests
  - Arbitration on the bus is the serialization point of requests
  - Fast, but not scalable
- **Directory Based:**
  - Coherence requests for a block are unicast to a directory
  - The directory forwards each request to the appropriate core
  - Require no assumptions on the interconnection layer
  - Arbitration at the directory is the serialization point of requests
  - Scalable, but not fast

Snooping-Cache System Model

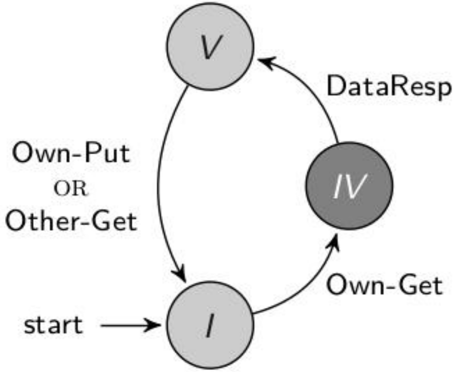


The VI Protocol

- Only one cache controller can read and/or write the block in any epoch
- Supported transactions:
  - Get: to request a block in read-write mode from the LLC controller
  - Put: to write the block's data back to the LLC controller
- • List of events:
  - `Own-Get`: Get transaction issued from local cache controller
  - `Other-Get`: Get transaction issued from remote cache controller
  - `Any-Get`: Get transaction issued from any controller
  - `Own-Put`: Put transaction issued from local cache controller
  - `Other-Put`: Put transaction issued from remote cache controller
  - `Any-Put`: Put transaction issued from any controller
  - `DataResp`: the block's data has been successfully received

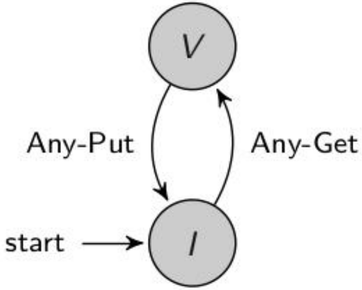
The VI Protocol

Cache controllers



V	Valid read-write copy
IV	Waiting for an up-to-date copy (transient state)
I	Invalid copy

LLC/Memory controller



V	One valid copy in L1
I	No valid copies in L1

The VI Protocol

- It has an implicit notion of *dirtiness* of a block
  - When in state V , the L1 controller can either read-write or just read the block (can’t distinguish between the two usages)
- It has an implicit notion of *exclusiveness* for a block
  - When in state V , the L1 controller has exclusive access to that block (no one else has a valid copy)
- It has an implicit notion of ownership of a block
  - When in state V , the L1 controller is responsible for transmitting the updated copy to any other controller requesting it
  - In all other states, the LLC is responsible for the data transfer
- This protocol has minimal space overhead (only a few states), but it is quite inefficient—why?

What a CC Protocol should offer

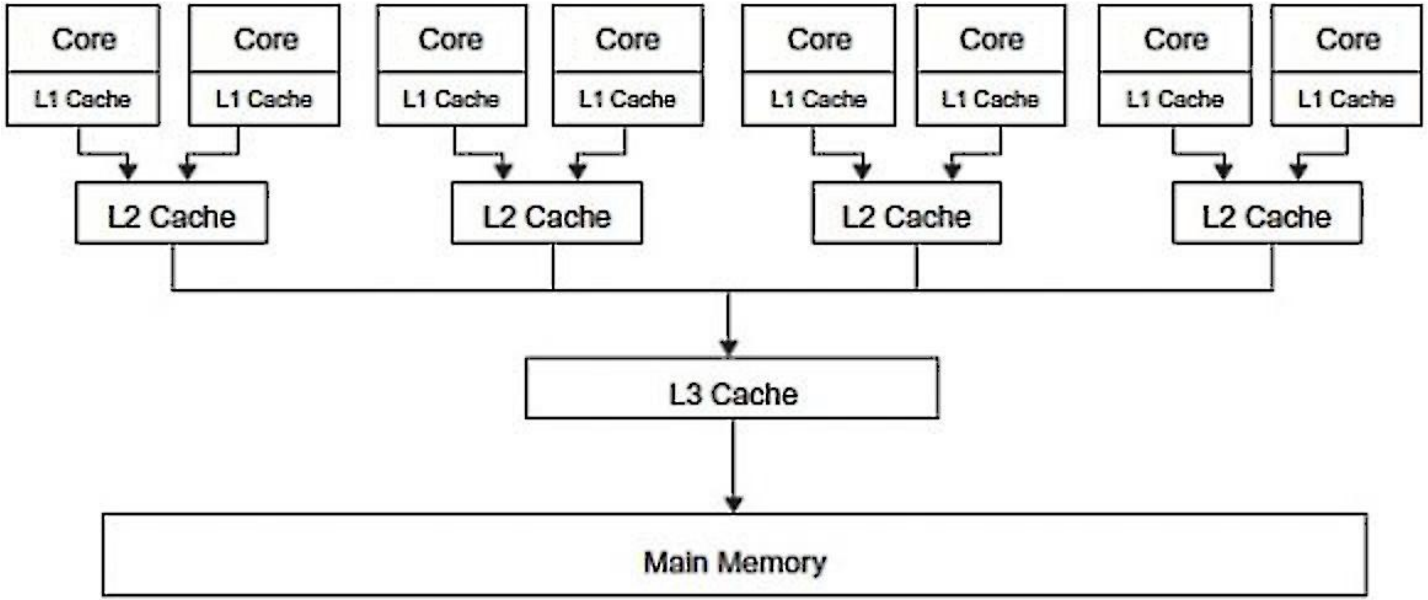
- We are interested in capturing more aspects of a cache block
  - *Validity*: A valid block has the most up-to-date value for this block. The block can be read, but can be written only if it is exclusive.
  - *Dirtiness*: A block is dirty if its value is the most up-to-date value, and it differs from the one stored in the LLC/Memory.
  - *Exclusivity*: A cache block is exclusive if it is the only privately cached copy of that block in the system (except for the LLC/Memory).
  - *Ownership*: A cache controller is the owner of the block if it is responsible for responding to coherence requests for that block.

- In principle, the more properties are captured, the more aggressive is the optimization (and the space overhead!)

MOESI Stable States

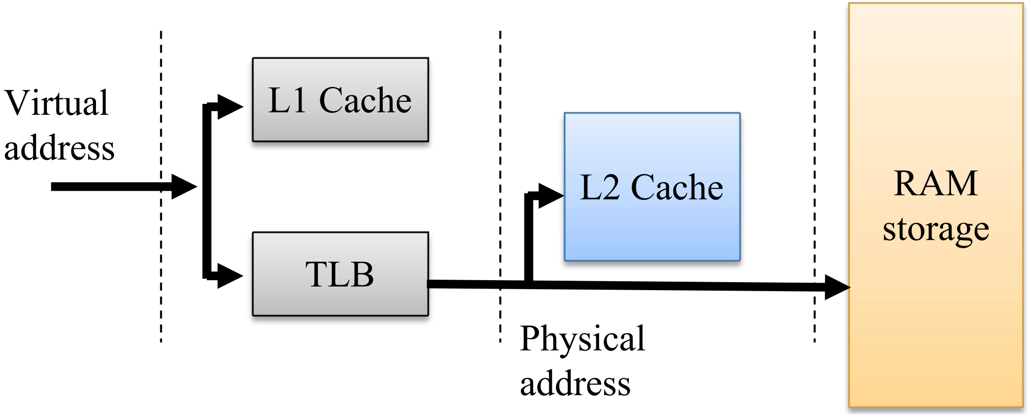
- **Modified (M)**: The block is valid, exclusive, owned and potentially dirty. It can be read or written. The cache has the only valid copy of the block.
- **Owned (O):** The block is valid, owned, and potentially dirty, but not exclusive. It can be only read, and the cache must respond to block requests.
- **Exclusive (E):** The block is valid, exclusive and clean. It can be only read. No other caches have a valid copy of the block. The LLC/Memory block is up-to-date.
- **Shared (S):** The block is valid but not exclusive, not dirty, and not owned. It can be only read. Other caches may have valid or read-only copies of the block.
- **Invalid (I):** The block is invalid. The cache either does not contain the block, or the block is potentially stale. It cannot be read nor written.

Typical Off-the-Shelf Multicore



Virtual vs. Physical Cache Indexing

- Caches may be indexed by either the virtual or the physical addresses
  - This depends on the details of the specific architecture
- Virtual address indexing is generally only used in L1 caches
  - The virtual-to-physical translation is typically not yet available there
- Physically-indexed caches are used at the outer levels
- Also the virtual-to-physical translation is cached
  - This is the purpose of the Translation Lookaside Buffer



In-Memory Transactions

Hardware Transactional Memory Support

Hardware Transactional Memory

- First proposed in 1993 by Herlihy, Eliot and Moss
- Motivations
  - lock-free operations on data structures will not be prevented if other threads stall amidst execution
  - avoid common problems with mutual exclusion
  - simplify code development
  - provide good performance (sometimes better than with locking)
- Introduce additional assembly instructions to denote transaction
- If a conflict on data is detected, portions of the execution flow are dropped
  - Also side effects are squashed
- Implemented 20 years after the original proposal by Herlihy, Eliot and Moss (June 2013)

How a Hardware Transaction Works (Intel TSX)

- Take advantage of the first-level cache and the cache coherency protocol (MESIFT)
  - Not all Intel CPUs support TSX
  - Need to test CPUID.07H.EBX.RTM [bit 11] = 1
- Tentative changes are made to the first level cache only
  - Also the *architectural state* is affected
- Tentative changes are made visibile to other cores *atomically* upon a successful commit

How a Hardware Transaction Works (Intel TSX)

- Four new assembly instructions are introduced (Restricted Transactional Memory algorithm):
  - `xbegin`: transaction begin
  - `xend`: transaction end
  - `xabort`: transaction abort
  - `xtest`: test if code is running within a transaction
- Transactions can be nested up to an implementation limit
  - The transactions will commit only if nesting count is zero
- Intrinsics headers are provided to access transactional instructions from C code
- They generate Undefined Instruction exception if the processor does not support RTM

HTM Transaction Example

```
while (1) {
        int status = _xbegin();
        if (status == _XBEGIN_STARTED) {
        //transaction body starts here
        ....
        ....
        //transaction body ends here
        _xend();
        break;
        } else {
        //actions on aborts
        ....
        ....
        }
}
```

HTM Transaction Example (assembly code)

```
retry:
            or eax, 0FFFFFFFFh
            xbegin L0
L0:
            cmp eax, 0FFFFFFFFh
            jne L1 ;jump to 'else' branch
            ;transaction body starts here
            ....
            ....
            ;transaction body ends here
            xend
            jmp L2

L1: ;'else branch'
            ;actions on aborts
            ....
            ....
            jmp retry
L2:
             ....
```

Why does a Transaction Abort?

- Instructions inside a transaction read and write memory locations
    - This builds up the *read set* and *write set* in the caches
    - The write set is marked in the L1 data cache, the read set in the L1–L3 caches
- A transaction will abort if *any other CPU*:
    - read a location in its write set
    - writes to a location in its read or write set
- Transactions may *also* abort due to hardware limitations:
    - context switches
    - interrupts
    - page faults
    - update of virtual-to-physical memory translation data
    - L1/L3 cache size limitations
- The code *must* provide a non-transactional execution path that can be executed if a transaction fails continuously

Transaction Abort Codes

- If a transaction is aborted, a status code is returned via the `eax` register

Eax Bit	Meaning
0	set if abort caused by XABORT instruction
1	transaction may succeed on retry (it is always clear if bit 0 is set)
2	set if another logical processor conflicts with a transaction memory address
3	set if internal buffer overflowed
4	set if debug breakpoint was hit
5	set if an abort occurred during a nested transaction
6..23	reserved
24..31	XABORT argument

- An aborted transaction can return 0 in `eax` (no bits set)!

An Example

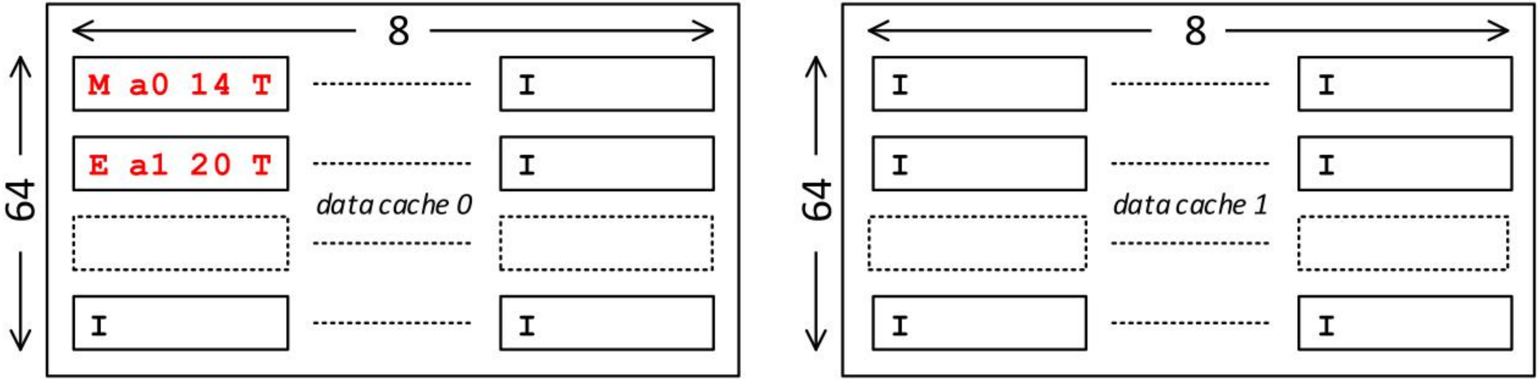
- Assume initially `a0 = 10` and `a1 = 20`

```
xbegin
    a0 += 4;     // add 4
    a1 -= 4;     // subtract 4
xend
```

- Simulates an atomic transfer of 4€ from one bank account to another
- The transaction only involves two memory locations

An Example: Execution in L1 cache

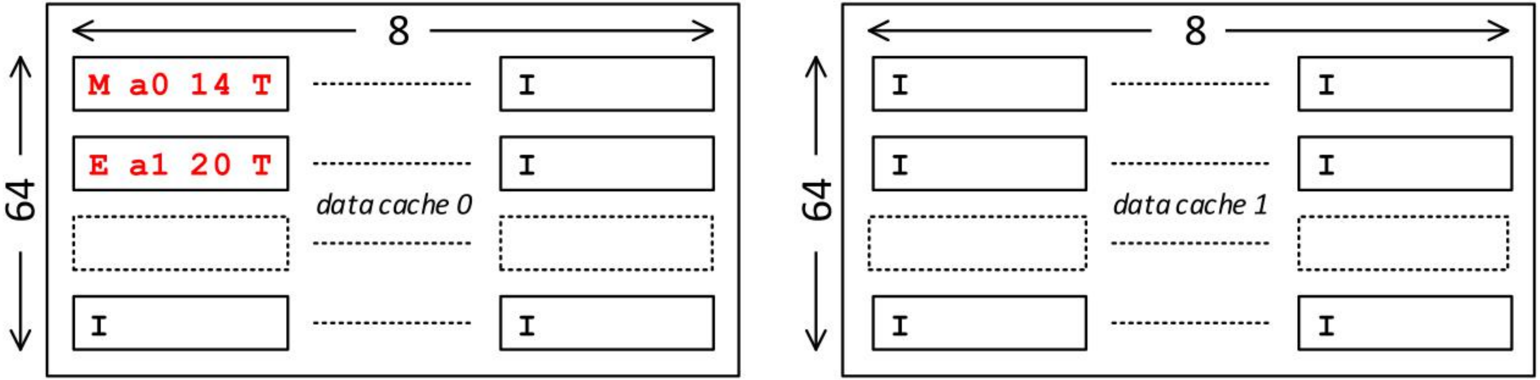
- CPU0 starts the transaction
- CPU0 reads `a0` into cache (Exclusive) and sets T bit (`a0` in the read set)
- CPU0 writes `a0` (`a0 += 4`) in cache only (Modified) (`a0` in the write set)
- CPU0 reads `a1` into cache (Exclusive) and sets T bit (`a1` in the read set)



- CPU0 writes `a1` (`a1 -= 4`) in cache only (Modified) (`a1` in the write set)
- xend is executed:
  - Transaction commits by clearing all T bits atomically
  - Modified cache lines are now visible and accessible (Modified)

An Example: Execution in L1 cache

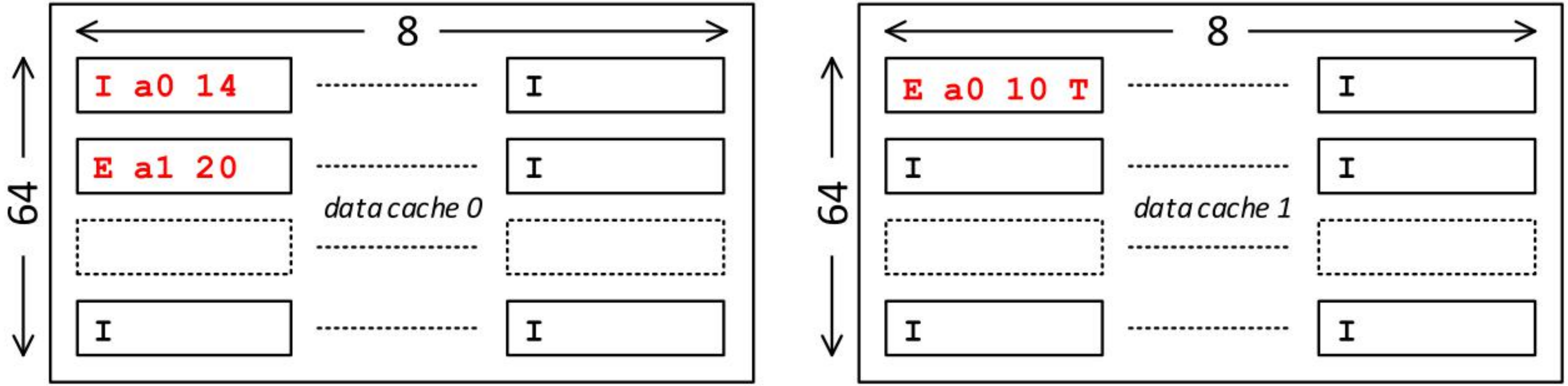
- Let’s consider a concurrent execution starting from this same state:



- CPU1 tries to read `a0` into its cache
- CPU0 detects a conflict because CPU1 is attempting to read data from its write set (T=1 and Modified)
- CPU0 aborts the transaction by invalidating all Modified cache lines with T=1 and clearing all T bits atomically

An Example: Execution in L1 cache

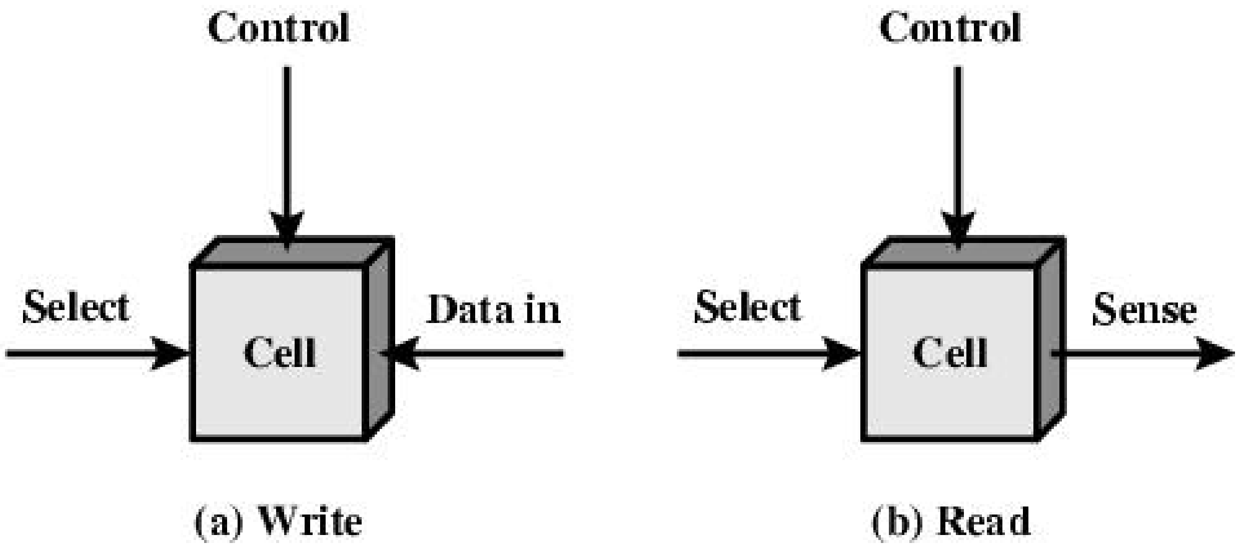
- CPU1 will read `a0` directly from memory



- Even though CPU0 had nearly completed its transaction, it's CPU0 that aborts
- CPU0 would also abort if CPU1 reads `a0` outside of a transaction
  - A transaction is aborted if data from the read/write set is accessed in any case

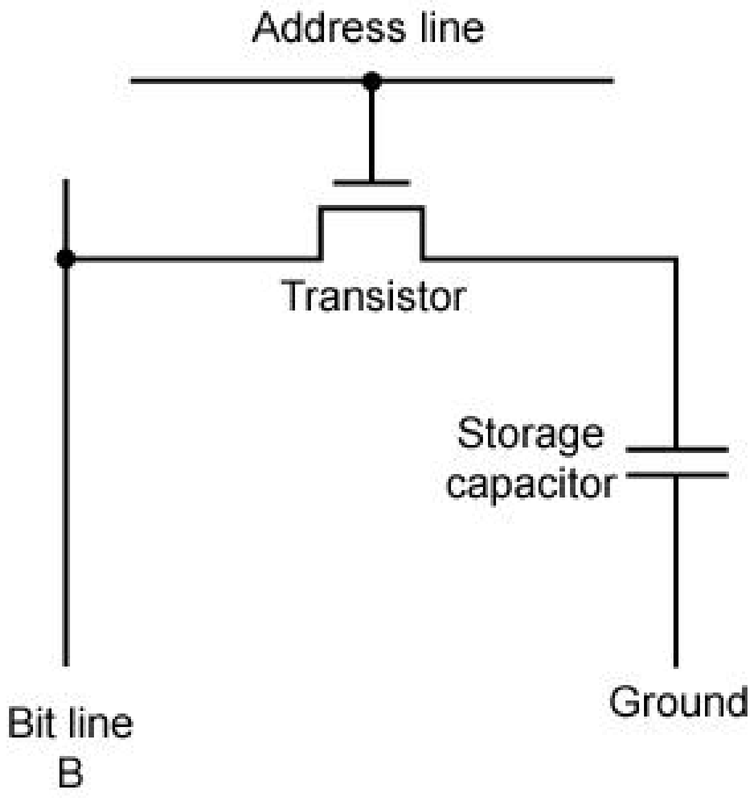
Recap on DRAM

Memory Cell Operation

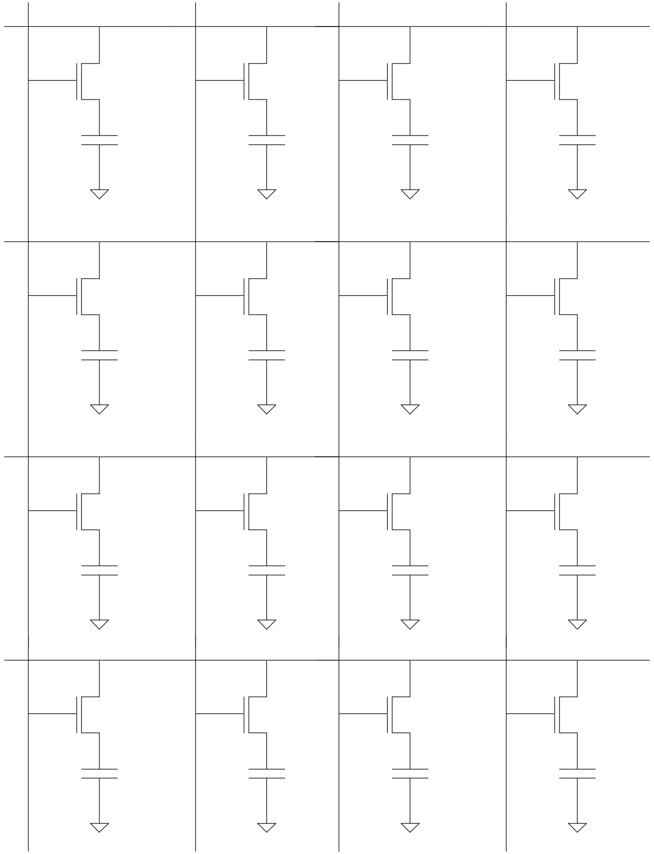


Dynamic RAM

- Bits are stored as charge in capacitors
  - Charges leak
  - Need refreshing even when powered
- Simpler construction
- Smaller per bit
- Less expensive
- Slower
- Level of charge determines value



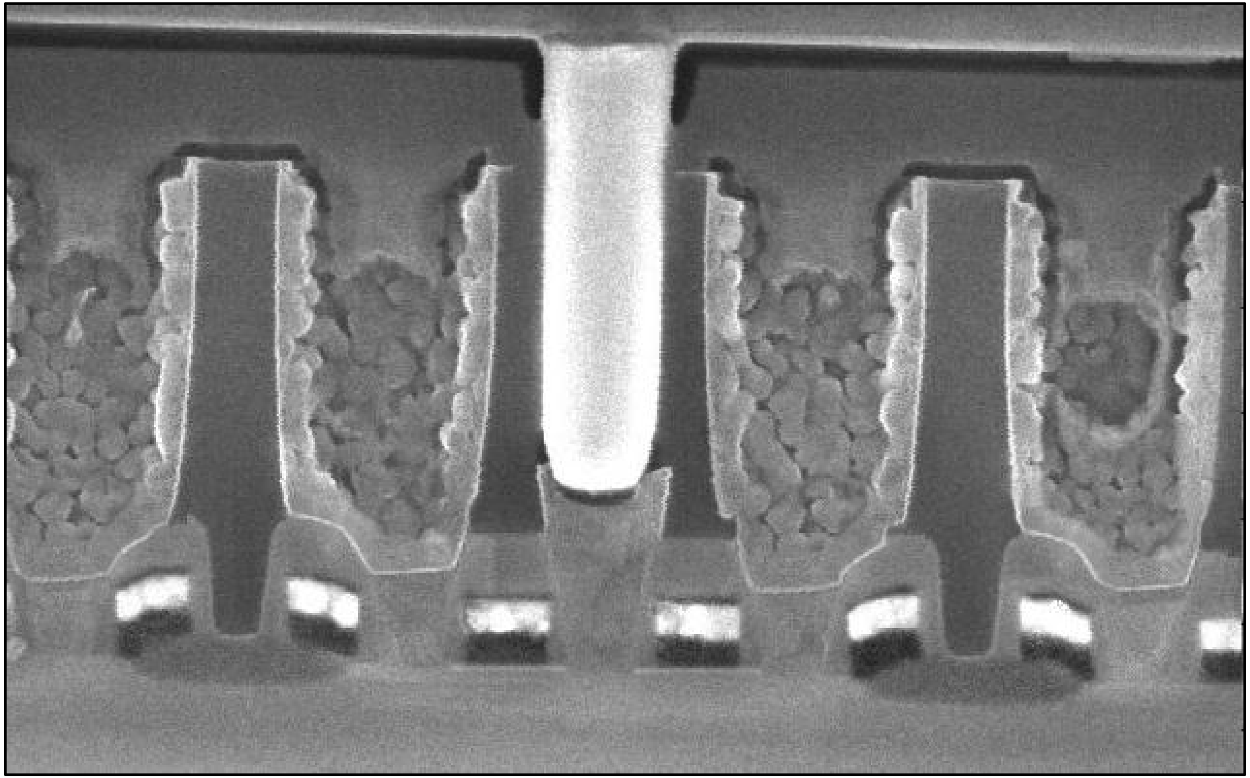
DRAM Memory Array



DRAM Operations

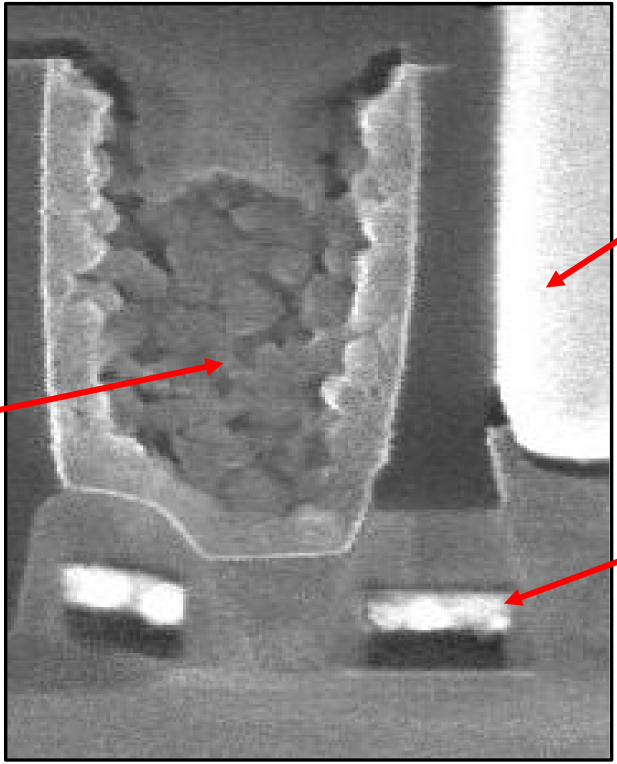
- Address line active when bit read or written
  - transistor switch closed (current flows)
- Write
  - voltage to bit line
    - high for 1, low for 0
  - then signal address line
    - transfers charge to capacitor
- Read
  - address line selected
    - transistor turns on
  - charge from capacitor fed via bit line to sense amplifier
    - compares with reference value to determine 0 or 1
  - capacitor charge must be restored

What It Really Looks Like



DRAM Memory Cell

1 bit

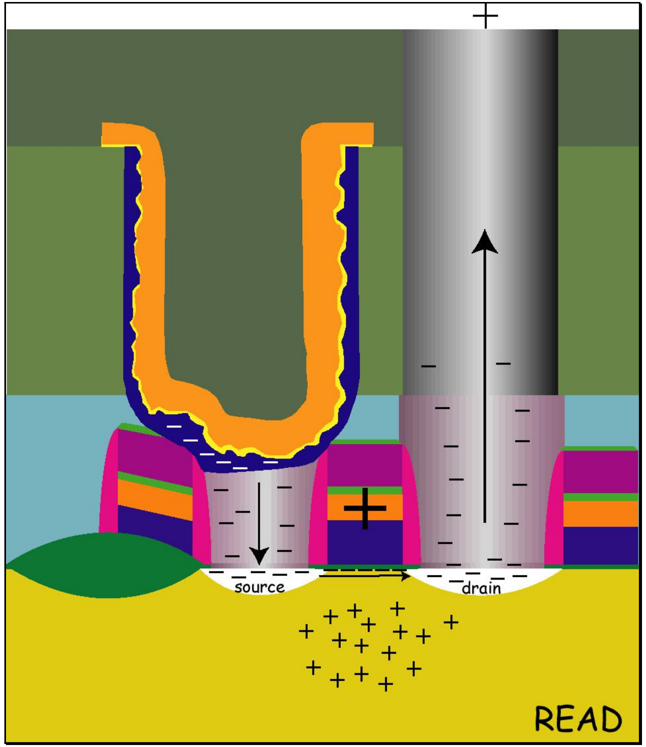


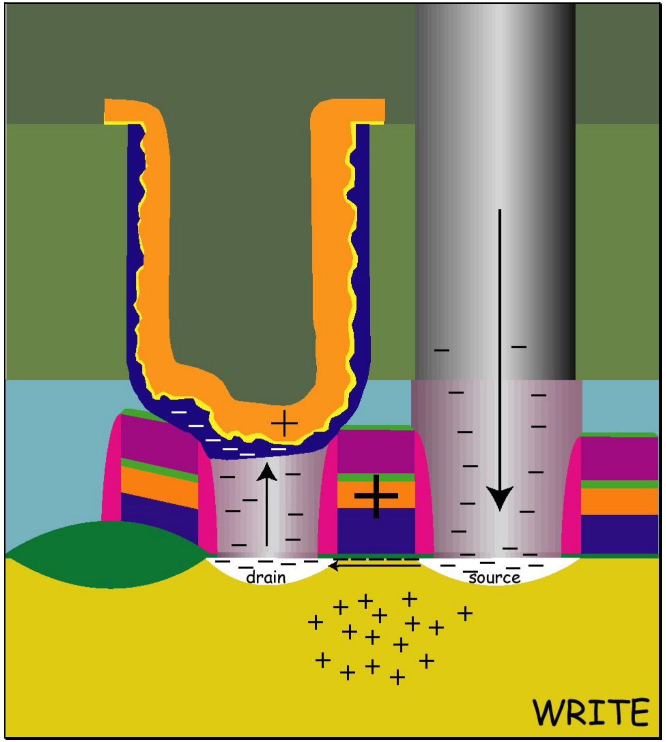
Capacitor

Column Line

Gate or Row Line

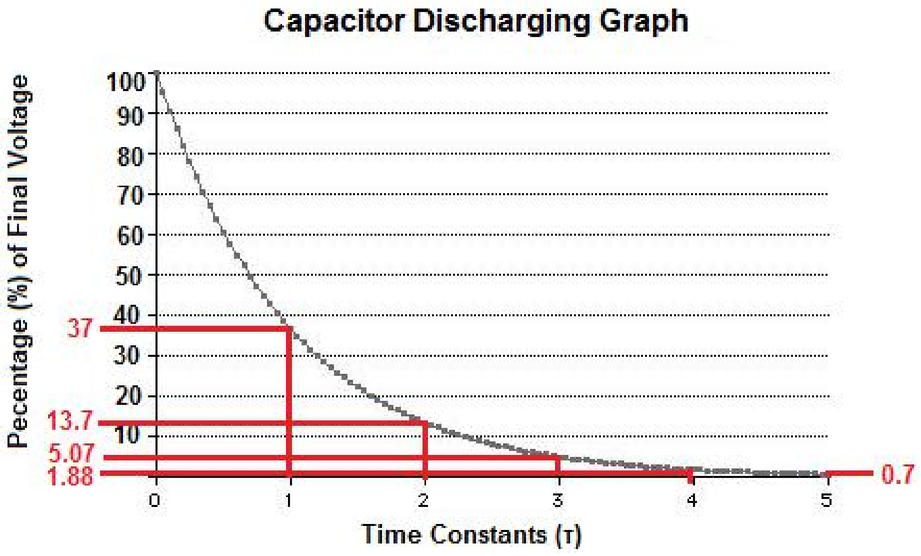
Read and Write Operations





DRAM Refresh

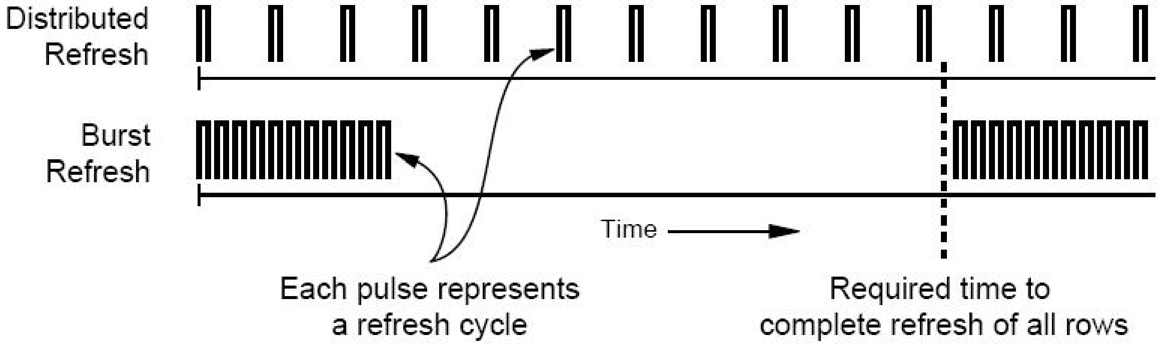
- Capacitors discharge over time
- The time it takes for a capacitor to discharge 63% of its fully charged voltage is equal to one time constant.
- 2 time constants: 86.3% of the supply voltage discharged.
- 3 time constants: 94.93%
- 4 time constants: 98.12%
- 5 time constants: 99.3%



DRAM Refresh

- The memory controller needs to refresh each row periodically to restore charge
- Typically, this is done every 64 ms
- Downsides of refresh
  - *Energy consumption*: Each refresh consumes energy
  - *Performance degradation*: DRAM row unavailable while refreshed
  - *QoS/predictability impact*: (Long) pause times during refresh
  - Refresh rate limits DRAM capacity scaling

Distributed Refresh



- Distributed refresh eliminates long pause times
- How else can we reduce the effect of refresh on performance/QoS?
- Does distributed refresh reduce refresh impact on energy?
- Can we reduce the number of refreshes?

Refresh Overhead: Performance