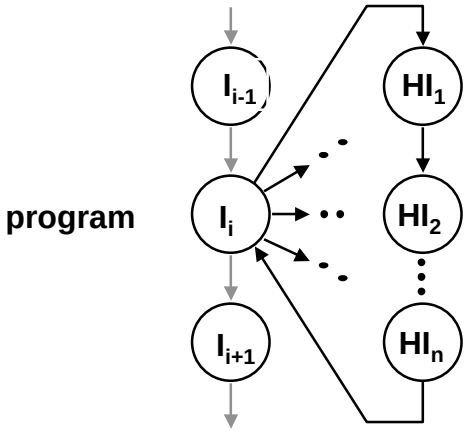
Interrupts, Caches

Krste Asanovic
Laboratory for Computer Science
M.I.T.

http://www.csg.lcs.mit.edu/6.823

Interrupts

alter normal flow of control



An *external or internal event* that needs to be processed by another (system) program. The event is usually unexpected or rare from program's point of view.

Causes of Interrupts

Interrupt is an *event* that requests the attention of the processor

Asynchronous: **an *external event***
- **input/output device service-request**
- **timer expiration**
- **power disruptions, hardware failure**

Synchronous: **an *internal event*** ***(aka exceptions)***
- **undefined opcode, privileged instruction**
- **arithmetic overflow, FPU exception**
- **misaligned memory access**
- ***virtual memory exceptions:***
  page faults, TLB misses, protection violations
- ***traps:*** **system calls (i.e., jumps into kernel code)**

Asynchronous Interrupts:
invoking the interrupt handler

An I/O device requests attention by asserting one of the *prioritized interrupt request lines*

When the processor decides to process the interrupt
- **it stops the current program at instruction I_i , completing all the instructions up to I_{i-1}**
  (*precise interrupt)*
- **it saves the PC of instruction I_i in a special register (EPC)**
- **disables interrupts and transfers control to a designated interrupt handler running in kernel mode**

Interrupt Handler

To allow nested interrupts, EPC is saved before enabling interrupts ⇒

- ***need an instruction to move EPC into GPRs***
- ***need a way to mask further interrupts at least until EPC can be saved***

There is a *status register* which indicates the cause of the interrupt - it must be visible to an interrupt handler

The return from an interrupt handler is a simple indirect jump but usually involves

- **enabling interrupts**
- **restoring the processor to the user mode**
- **restoring hardware status and control state**

⇒ ***a special return-from-exception instruction (RFE)***

Synchronous Interrupts

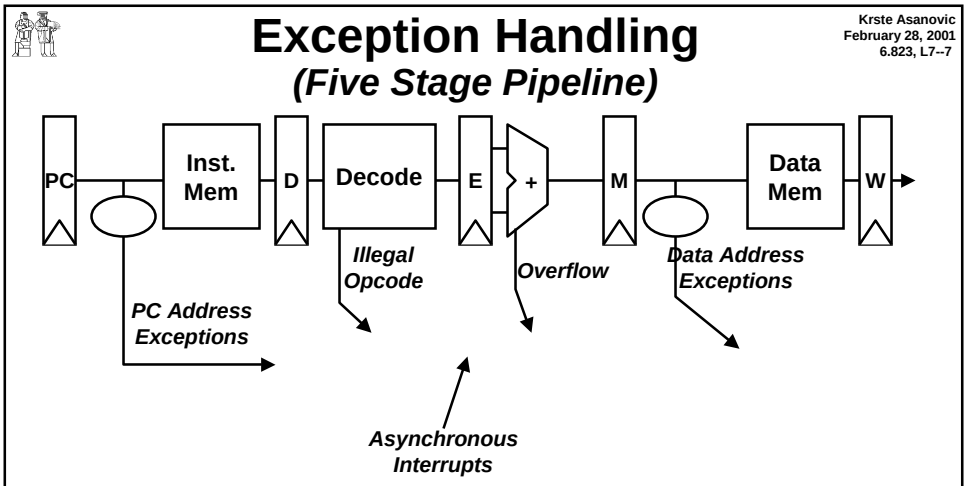
A synchronous interrupt (exception) is caused by a *particular instruction*

In general, the instruction cannot be completed and needs to be *restarted* after the exception has been handled

requires undoing the effect of one or more partially executed instructions

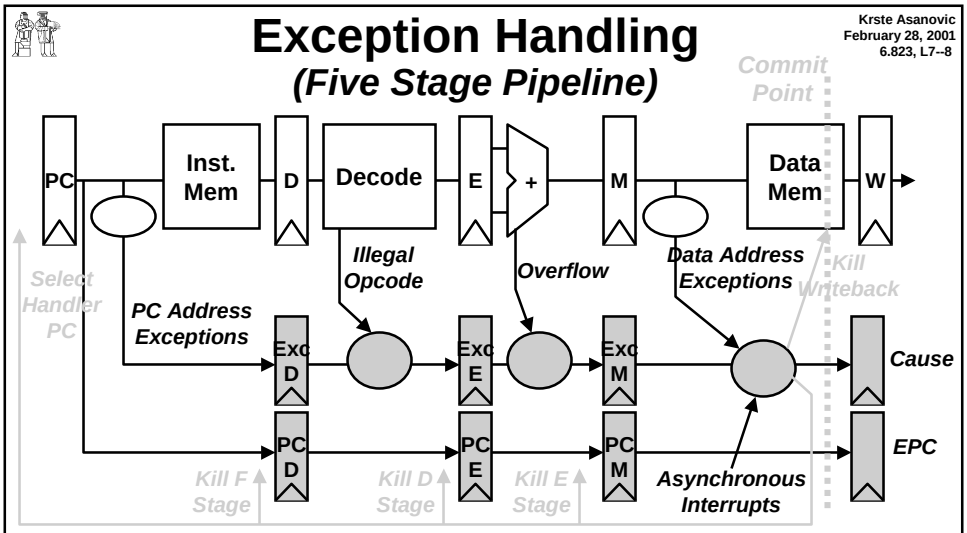
In case of a trap (system call), the instruction is considered to have been completed

a special jump instruction involving a change to privileged kernel mode



How to handle multiple simultaneous exceptions in different pipeline stages?

How and where to handle external asynchronous interrupts?



- **Hold exception flags in pipeline until commit point (M stage)**
- **Exceptions in earlier pipe stages override later exceptions**
- **Inject external interrupts at commit point (override others)**
- **If exception at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage**

Interrupts in Branch Delay Slot

- **If interrupt occurs in branch delay slot, can we restart execution by jumping to EPC?**

```
096 ADD R1, R2, R3
100 BEQZ R1, +300
104 SUB R4, R4, #1 (delay slot)  ← Interrupted
108 XOR R3, R1, R2
....
404 ADD R3, R4, R1
...
```

A DLX ISA-specific Solution

On an interrupt, if I_i is in a delay slot then save PC_{i-1} instead of PC_i

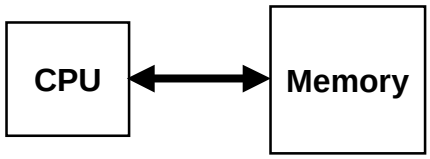
Execution can always be correctly resumed from the saved PC.

Why?

Hint: **can a jump instruction be re-executed?**

	modify state?	***OK to re-execute?***
J		
JR		
BZ		
JAL		
JALR		

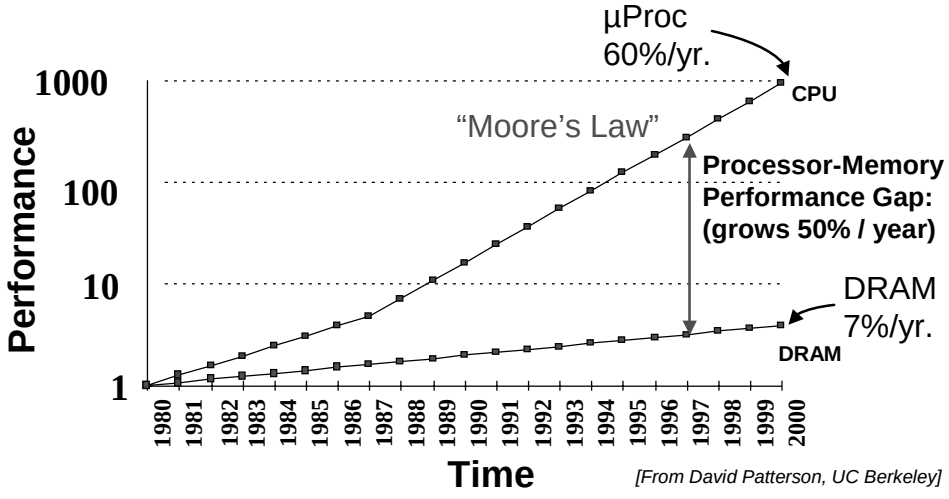
CPU-Memory Bottleneck



Performance of high speed computers is usually limited by memory *bandwidth* & *latency*

- ***Latency (time for a single access)***
  Memory access time >> Processor cycle time
- ***Bandwidth (number of accesses per unit time)***
  if fraction *m* of instructions access memory,
  ⇒ 1+*m* memory references / instruction
  ⇒ CPI = 1 requires 1+*m* memory refs / cycle

Processor-DRAM Gap (latency)



[From David Patterson, UC Berkeley]

Today, DRAM access >100 processor cycles!
Four-issue superscalar executes >400 instructions during cache miss!

Locality

Real program memory reference patterns do not touch whole address space uniformly and randomly

- ***Temporal Locality***

when a program accesses a memory location, it is likely to access it again in the near future

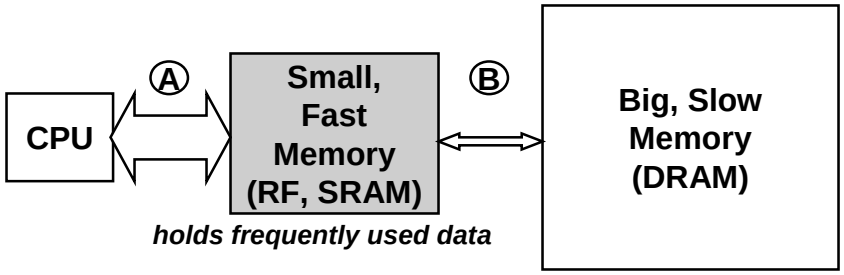
Examples?_____

- ***Spatial Locality***

when a program accesses a memory location, it is likely to access nearby locations in the near future

Examples?_____

Memory Hierarchy



- ***size*:** Register << SRAM << DRAM (due to cost)
- ***latency:*** Register << SRAM << DRAM (due to size)
- ***bandwidth:*** on-chip >> off-chip (due to cost & wiring)

On a data access:

***hit* (data $\in$ fast memory)** $\Rightarrow$ **low latency access**

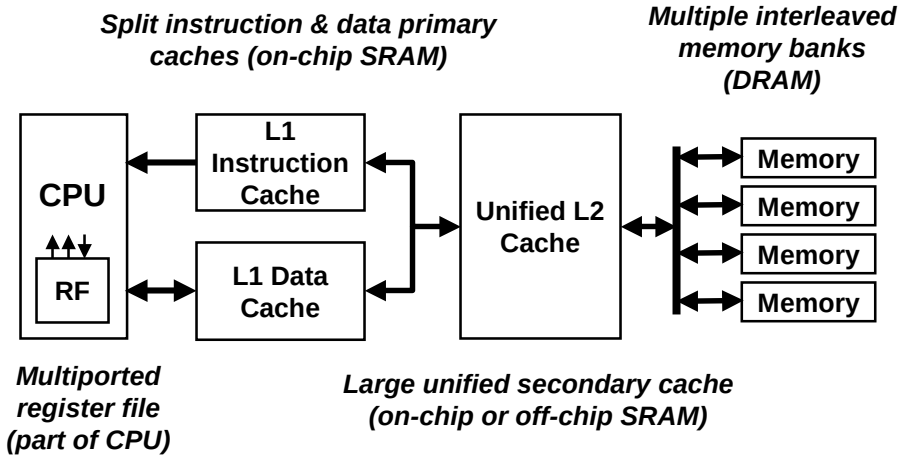
miss* (data $\notin$ fast memory)** $\Rightarrow$ **long latency access *(DRAM)

Fast memory is effective only if bandwidth requirement at B << A (i.e., many more hits than misses)

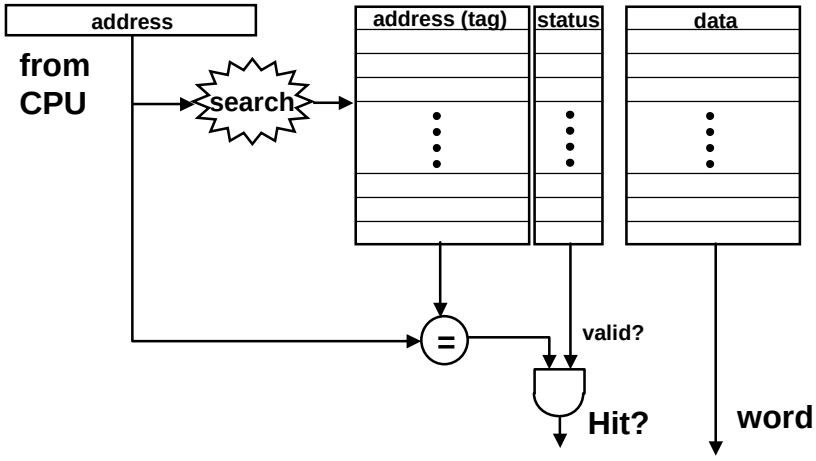
Management of Memory Hierarchy

- **Software managed, e.g., registers**
  - **part of the software-visible processor state**
  - **software in complete control of storage allocation**
    - » ***but hardware might do things behind software's back, e.g., register renaming***
- **Hardware managed, e.g., caches**
  - **not part of the software-visible processor state**
  - **hardware automatically decides what is kept in fast memory**
    - » ***but software may provide "hints", e.g., don't cache or prefetch***

A Typical Memory Hierarchy c.2000



Basic Cache Organization



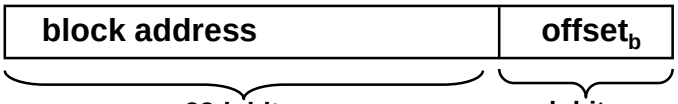
read hit ⇒ **data is immediately available**
write hit ⇒ **data is written to the cache**
miss ⇒ **request is passed on to the memory**

Block Size and Spatial Locality

Block is unit of transfer between the cache and memory

4 word block, b=2

Split CPU address into block address and offset within block



32-b bits **b bits**

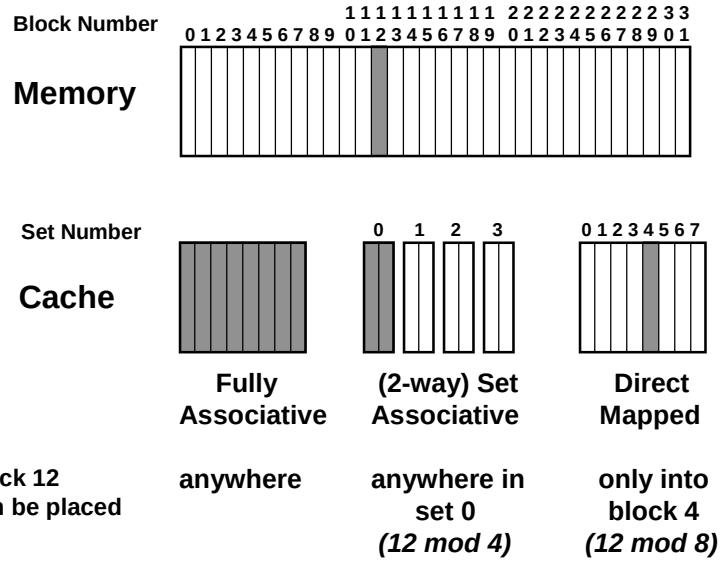
2^b = block size *aka* line size (in bytes)

Larger block size has distinct hardware advantages

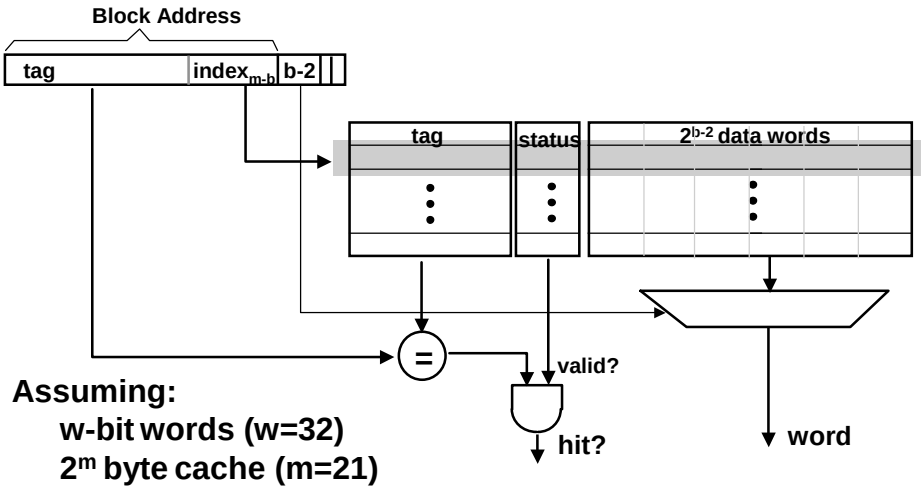
- ***less tag overhead***
- ***exploit fast burst transfers from DRAM***
- ***exploit fast burst transfers over wide busses***

Is it profitable to fetch several consecutive words at a time?
Effect on overall performance?

Placement Policy



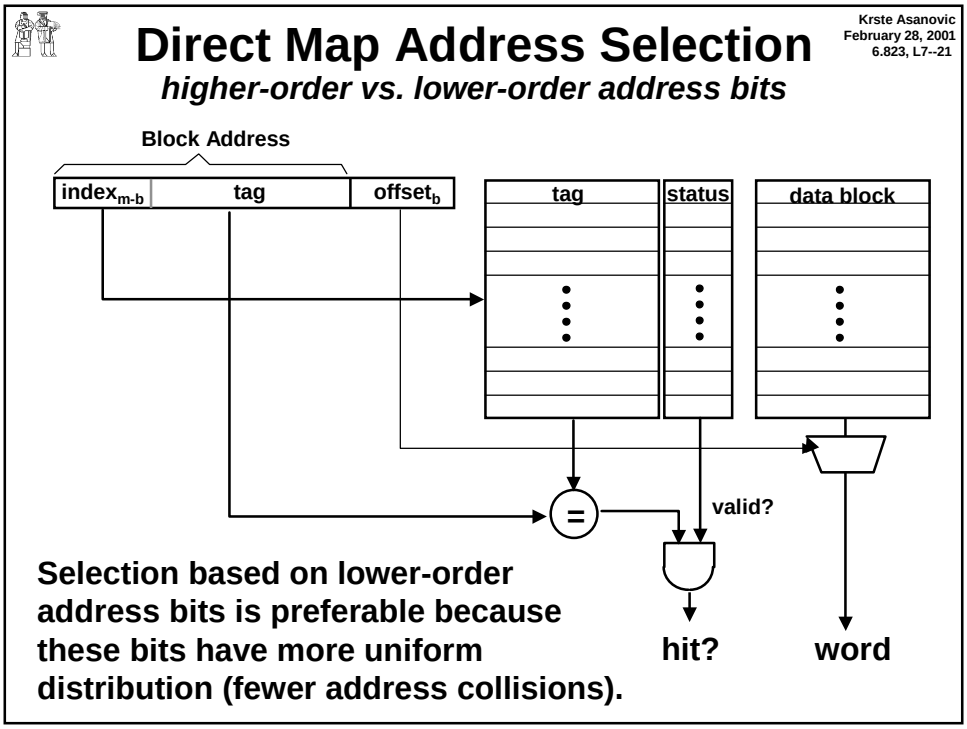
Direct-Mapped Cache

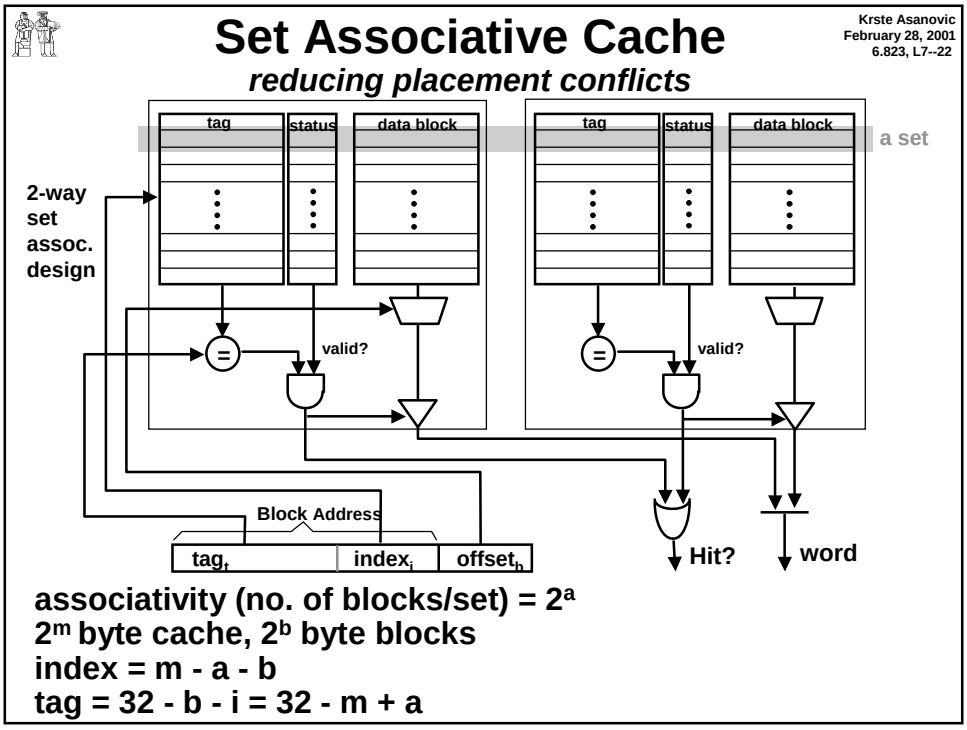


Assuming:

- w-bit words (w=32)
- 2^m byte cache (m=21)
- 2^b-byte cache blocks (b=2, 4)
- 2^w byte addresses

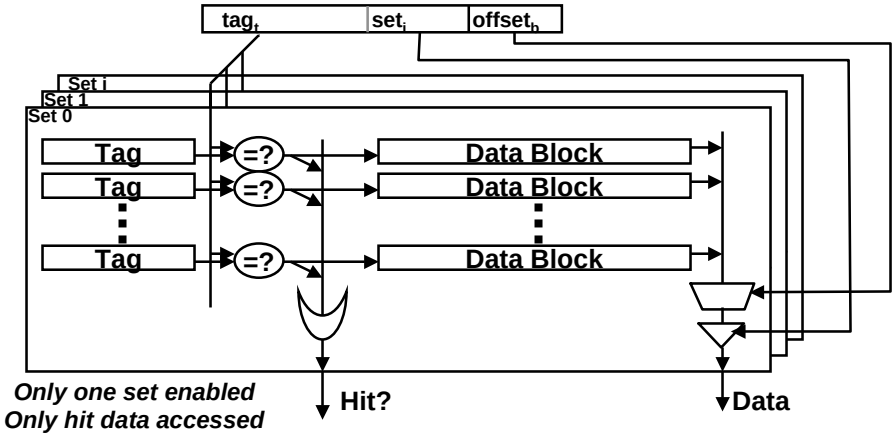
Tag storage overhead per byte _____?


Direct Map Address Selection
higher-order vs. lower-order address bits
Krste Asanovic
February 28, 2001
6.823, L7--21
Block Address
index$_{m-b}$
tag
offset$_b$
tag
status
data block
=
valid?
hit?
word
Selection based on lower-order address bits is preferable because these bits have more uniform distribution (fewer address collisions).


Set Associative Cache
reducing placement conflicts
Krste Asanovic
February 28, 2001
6.823, L7--22
tag
status
data block
a set
2-way set assoc. design
=
valid?
Block Address
tag$_t$
index$_i$
offset$_b$
Hit?
word
associativity (no. of blocks/set) = 2^a
2^m byte cache, 2^b byte blocks
index = m - a - b
tag = 32 - b - i = 32 - m + a

Highly-Associative Caches

- For high associativity, use content-addressable memory (CAM) for tags
- Used in low-power microprocessors, e.g. StrongARM is 32-way set-associative. *(High hit rates at lower energy than 2-4 way set assoc. designs.)*
- Comparator per tag requires more transistors (~double area per tag bit)



Replacement Policy

In an associative cache, which block from a set should be evicted when the set becomes full?

- ***Random***

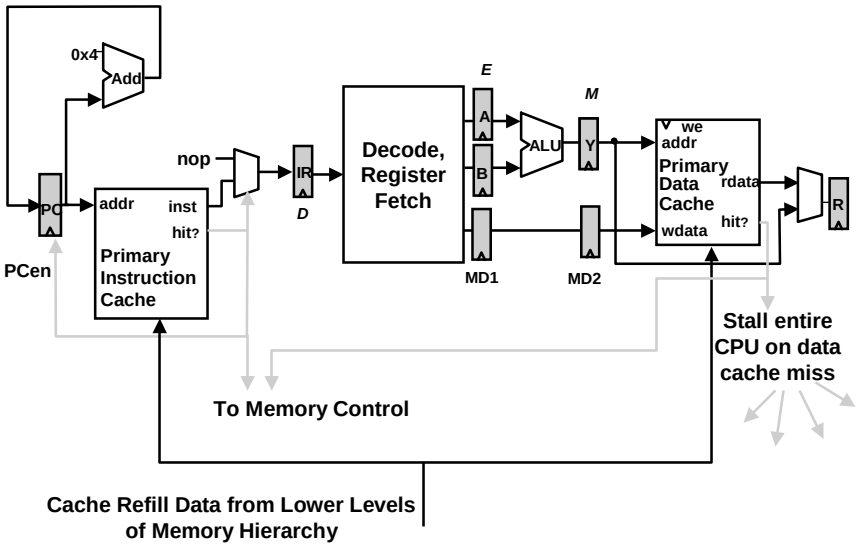
- ***Least Recently Used (LRU)***
  - LRU cache state must be updated on every access
  - true implementation only feasible for small sets (2-way easy)
  - pseudo-LRU binary tree often used for 4-8 way

- ***First In, First Out (FIFO)*** **aka. *Round-Robin***
  - used in highly associative caches

This is a second-order effect. How often does replacement happen?_____

CPU-Cache Interaction

(Simple 5-stage pipeline)



Write Policy

Cache hit:

write through: **write both cache & memory**

- generally higher traffic but simplifies cache coherence

write back: **write cache only (memory is written only when the entry is evicted)**

- a dirty bit per block can further reduce the traffic

Cache miss:

no write allocate: **only write to main memory**

write allocate: (aka fetch on write)

fetch block into cache

Common combinations:

write through and no write allocate

write back with write allocate

Managing Cache Writes

In a direct-mapped cache, can we write cache data RAM in same cycle as cache tag RAM read?

In a highly-associative cache with CAM tags, can we write cache data in the same cycle as tag CAM search?

Pipelining Cache Writes

Possible solutions:

- **Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit**
- **Design data RAM that can perform read *and* write in one cycle, restore old value after tag miss**
- **Hold write data for store in single buffer ahead of cache, write cache data during next store's tag check**
  - **Need to bypass from write buffer if read matches write buffer tag**
- **Use CAM tags, data write only enabled if hit**

Cache Performance

Average memory access time =
Hit time + Miss rate x Miss penalty

To improve performance:
- **reduce the hit time**
- **reduce the miss rate (e.g., larger cache)**
- **reduce the miss penalty (e.g., L2 cache)**

First order effect: *the size and the hit time*
⇒ design the largest primary cache without slowing down the clock or adding pipeline stages

Causes for Cache Misses

- ***Compulsory:*** first-reference *aka* cold start misses
  - misses that would occur even with infinite cache
- ***Capacity:*** cache is too small to hold all data needed by the program
  - misses that would occur even under perfect placement & replacement policy
- ***Conflict:*** misses that occur because of collisions due to block-placement strategy
  - misses that would not occur with full associativity

Determining the type of a miss requires running program traces on a cache simulator

Effect of Cache Parameters on Performance

- **Larger cache size**
  - **+ reduces capacity and conflict misses**
  - **- hit time may increase**
- **Larger block size**
  - **+ spatial locality reduces compulsory misses and capacity reload misses**
  - **- fewer blocks may increase conflict miss rate**
  - **- larger blocks may increase miss penalty**
- **Higher associativity**
  - **+ reduces conflict misses (up to around 4-8 way)**
  - **- may increase access time**