Processor Micro-Architecture: Part 2

Mark McDermott

Fall 2009

Agenda

- Branch Unit Design
- Memory System Design
  - Cache Memory
  - Virtual Memory

TLAs

- TLA – Three Letter Acronym
- RTN – Register Transfer Notation
- RTL – Register Transfer Language
- MAC – Multiply Accumulate
- ISA – Instruction Set Architecture
- CPI – Clocks per Instruction
- IPC – Instructions per clock
- ALU – Arithmetic Logic Unit
- FSM – Finite State Machine
- L2 – Level Two Cache
- GPP - General Purpose Processor
- FPGA – Field Programmable Gate Array
- ASIC - Application Specific Integrated Circuits
- VM – Virtual Memory
- VA – Virtual Address
- PA – Physical Address
- CPI – Clocks per Instruction
- BTC – Branch Target Cache
- BTAC – Branch Target Address Cache
- LRU – Least Recently Used
- FIFO - First-In-First-Out
- NMRU - Not most recently used
- PMI – Page miss per instruction
- PF – Page Fault

Why Branch Prediction?

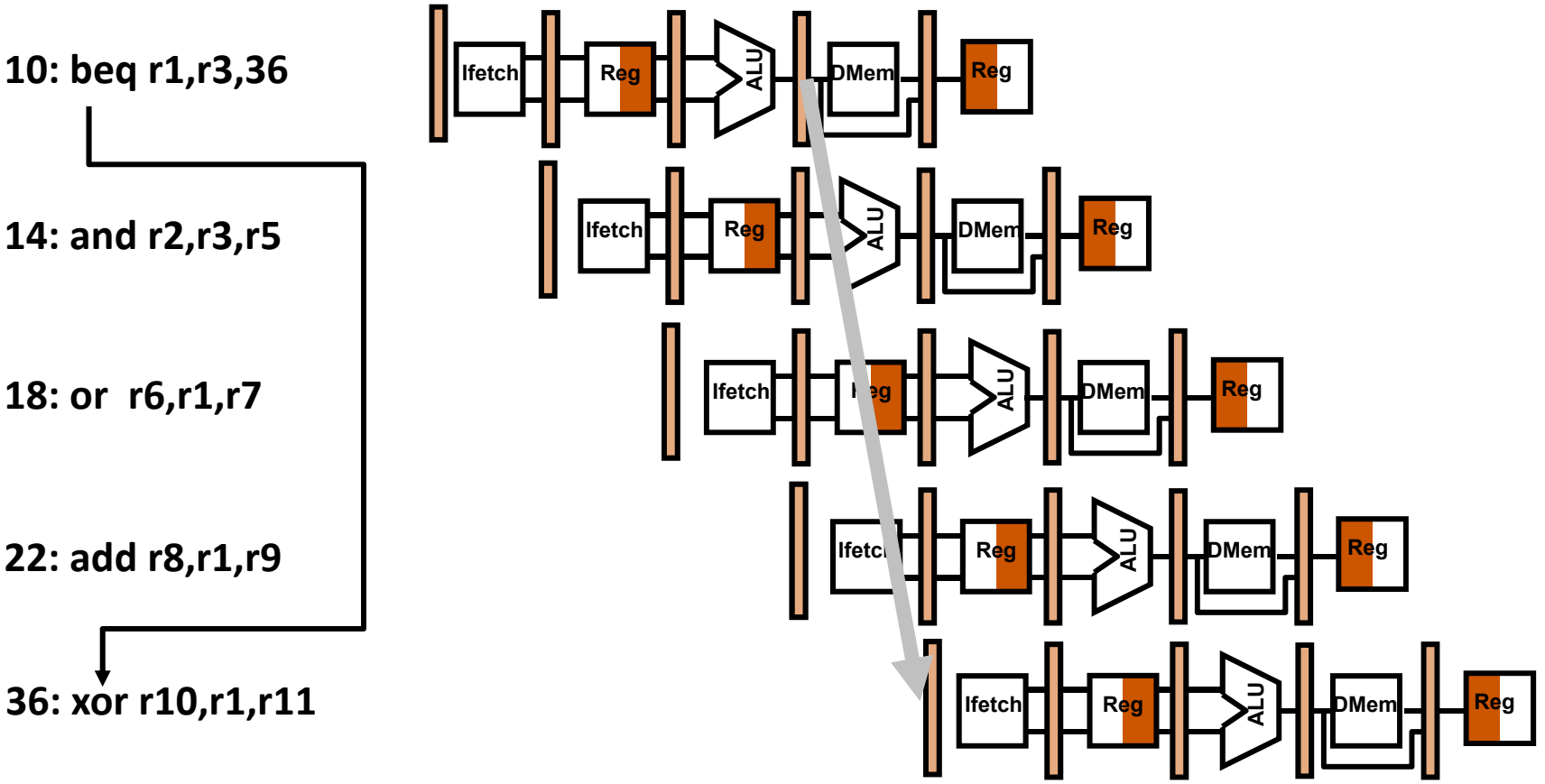
- Datapath parallelism only useful if you can keep it fed.
- Easy to fetch multiple (consecutive) instructions per cycle
  - Essentially speculating on sequential flow
  - Change of flow will always occur unless you are a pure dataflow machine
- Jump: unconditional change of control flow
  - Always taken
- Branch: conditional change of control flow
  - Taken about 50% of the time
  - Backward Taken: 30% of the time
  - Forward Taken: 70% of the time

- Solution: Predict the outcome of the branch
  - *Reactive*: past actions cause system to *adapt* use
  - *Proactive*: uses past actions to *predict* future actions

Branch Prediction Schemes

- Static Branch Prediction
- 1-bit Branch-Prediction Buffer
- 2-bit Branch-Prediction Buffer
- Correlating (Two-level) Branch Prediction Buffer
- Tournament Branch Predictor
- Branch Target Buffer
- Integrated Instruction Fetch Units
- Return Address Predictors

Control Hazard on Branches Three Stage Stall



What do you do with the 3 instructions in between?
How do you do it?
Where is the "commit"?

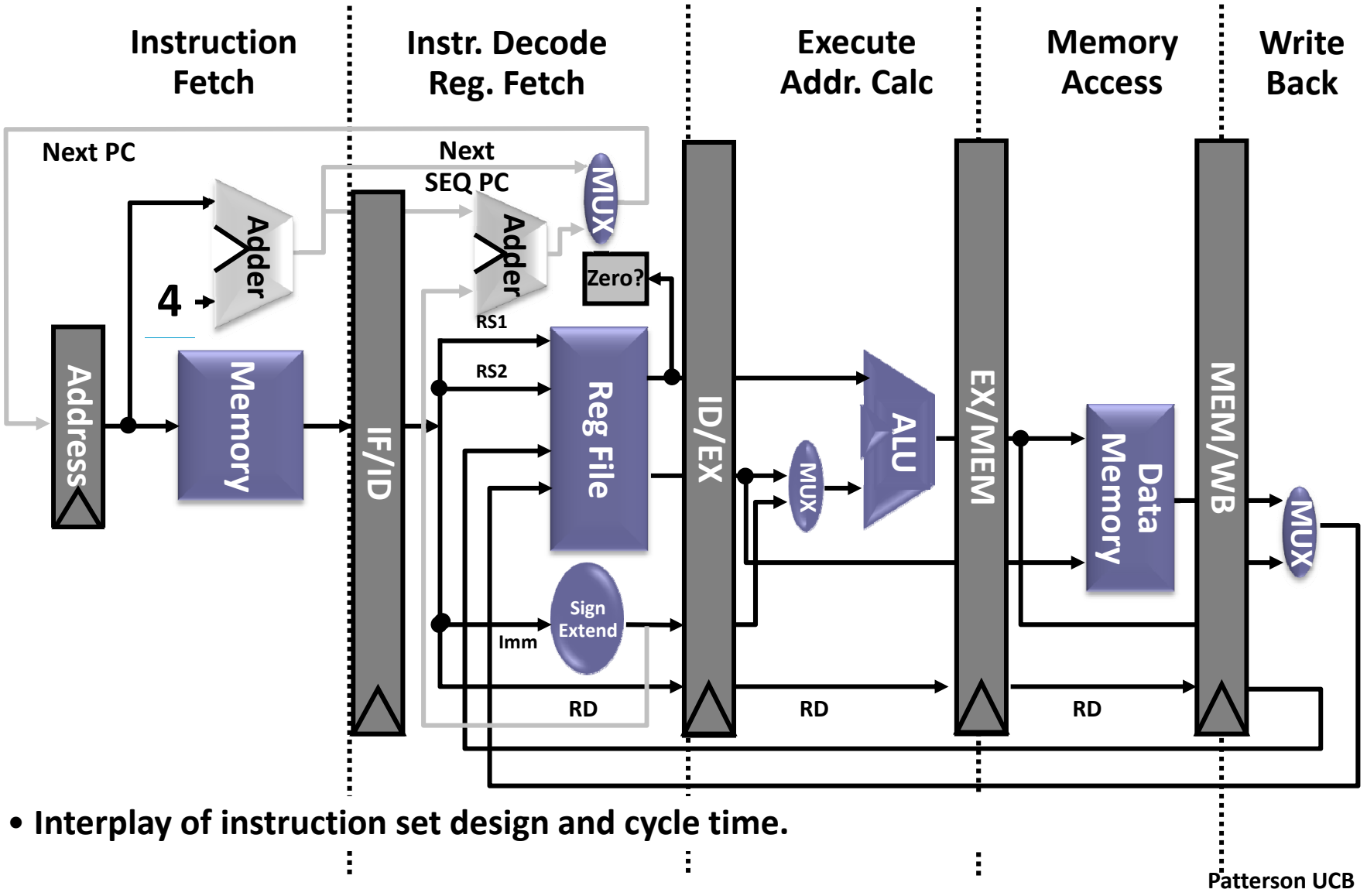
Patterson UCB

Branch Stall Impact

- If CPI = 1, 30% branch,
  Stall 3 cycles => new CPI = 1.9!
- Two part solution:
  - Determine branch taken (or not taken) sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or ≠ 0
- MIPS Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

Patterson UCB

Pipelined MIPS Datapath



- Interplay of instruction set design and cycle time.

Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- Execute successor instructions in sequence
- "Squash" instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- 47% MIPS branches not taken on average
- PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken

- 53% MIPS branches taken on average
- But haven't calculated branch target address in MIPS
  - MIPS still incurs 1 cycle branch penalty
  - Other machines: branch target known before outcome

Patterson UCB

Four Branch Hazard Alternatives (cont)

#4: Delayed Branch

- Define branch to take place *AFTER* a following instruction

```
branch instruction
    sequential successor1
    sequential successor2
    ........
    sequential successorn
branch target if taken
```

Branch delay of length *n*

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this technique

Patterson UCB

Delayed Branch

- **Where to get instructions to fill branch delay slot?**
  - Before branch instruction
  - From the target address: only valuable when branch taken
  - From fall through: only valuable when branch not taken
  - Canceling branches allow more slots to be filled
- **Compiler effectiveness for single branch delay slot:**
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - ~50% (60% x 80%) of slots usefully filled
- **Delayed Branch downside: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot**
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches

Patterson UCB

Evaluating Branch Alternatives

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

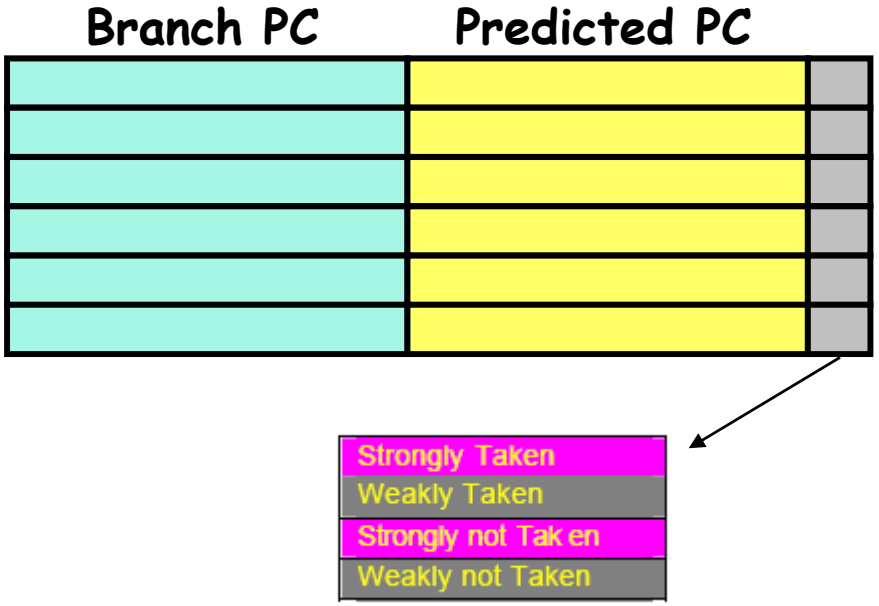
Assume 4% unconditional branch, 6% conditional branch- untaken, 10% conditional branch-taken

Scheduling scheme	*Branch penalty*	*CPI*	*speedup v. unpipelined*	*speedup v. stall*
Stall pipeline	3	1.60	3.1	1.0
Predict taken	1	1.20	4.2	1.33
Predict not taken	1	1.14	4.4	1.40
Delayed branch	0.5	1.10	4.5	1.45

Patterson UCB

ARM Branch Prediction Unit

- Dynamic branch predictor: a 64-entry, 4-state branch target address cache (BTAC) is maintained.
- Static branch prediction
  - If the branch instruction does not exist in the BTAC, a static branch prediction process takes over
  - Backward branch
    - Predict taken
  - Forward branch
    - Predict non-taken
- Return stack
  - Handle up to three procedure calls



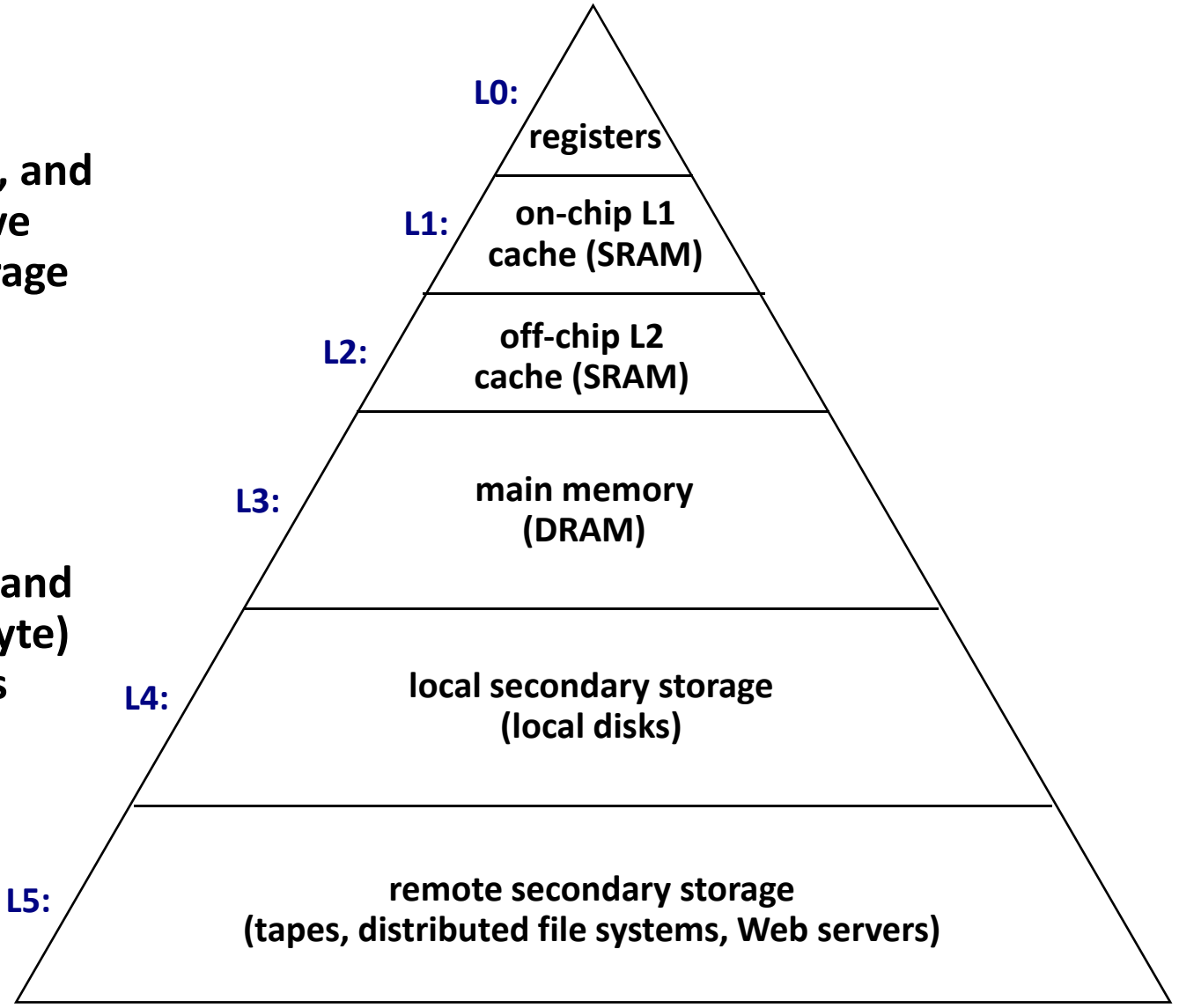
ARM, Inc.

Memory System Design

Memory System Design

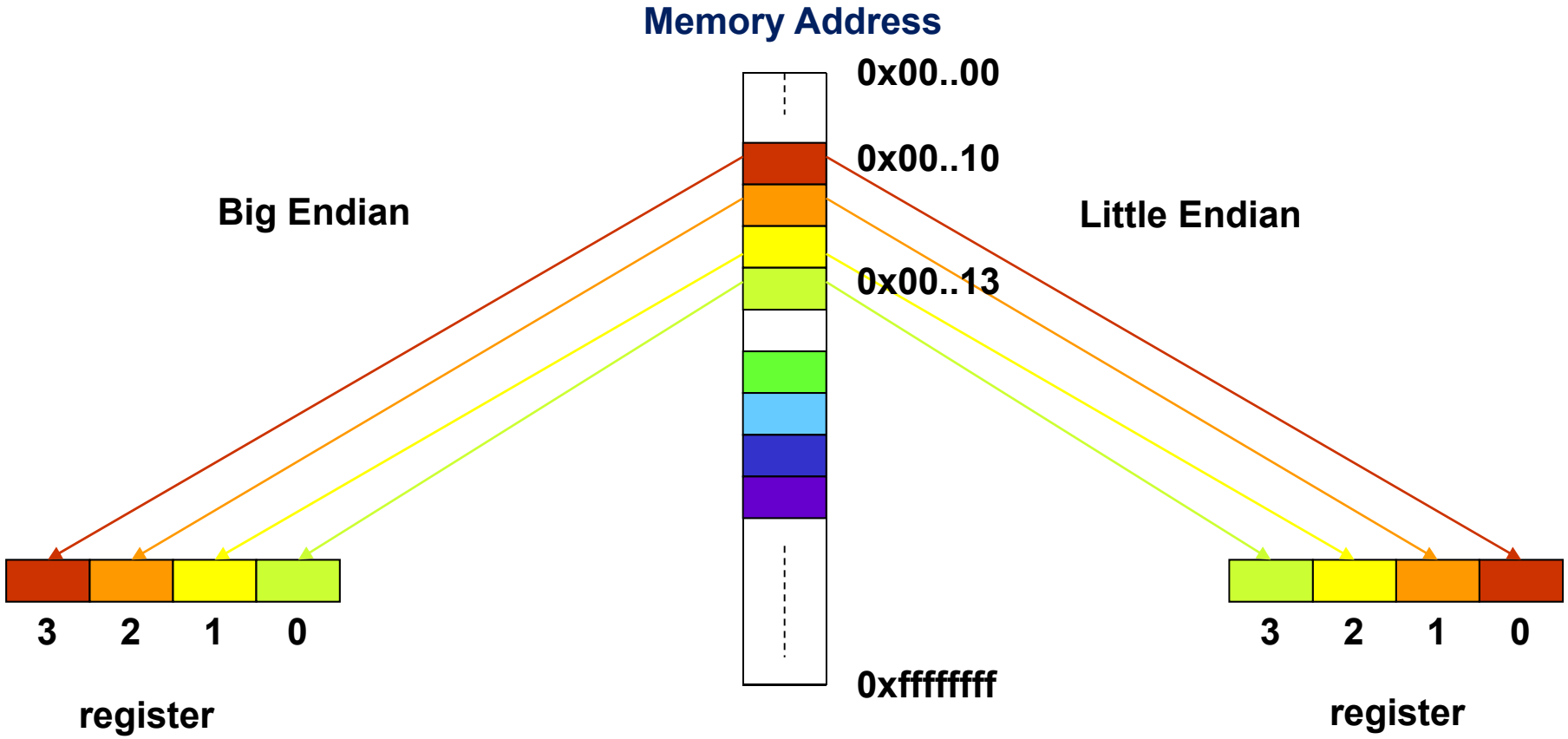
Smaller, faster, and more expensive (per byte) storage devices

Larger, slower, and cheaper (per byte) storage devices



Discussion on Memory Endianness

- Endianness: ordering of bytes within a larger object, e.g., word, i.e., how a large object is stored in memory
- ARM can be both a BIG Endian or a Little Endian machine



Why Memory Hierarchy?

- **Bandwidth:**

$$BW = \frac{1.0\,inst}{cycle} \times \left[\frac{1\,Ifetch}{inst} \times \frac{4\,B}{Ifetch} + \frac{0.4\,Dref}{inst} \times \frac{8\,B}{Dref} \right] \times \frac{3\,Gcycles}{\text{sec}}$$
$$= \frac{21.6\,GB}{\text{sec}}$$

- **Capacity:**
  - 1+GB for Windows PC to multiple TB
- **Cost:**
  - (TB x anything) adds up quickly
- **These requirements appear incompatible**

Why Memory Hierarchy?

- **Fast and small memories**
  - Enable quick access (fast cycle time)
  - Enable lots of bandwidth (1+ L/S/I-fetch/cycle)
- **Slower larger memories**
  - Capture larger share of memory
  - Still relatively fast
- **Slow huge memories**
  - Hold rarely-needed state
  - Needed for correctness

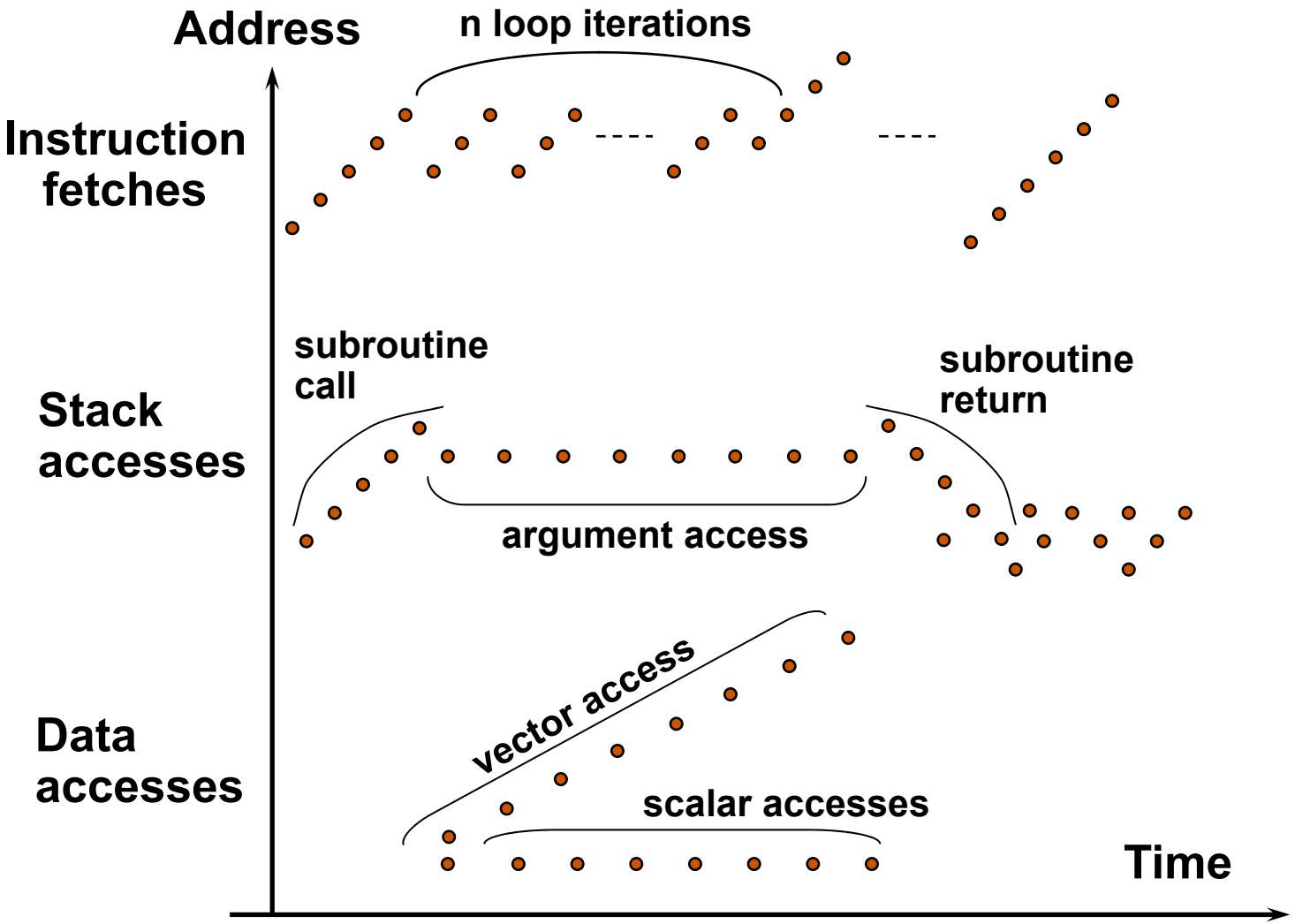
Combination provides the appearance of large, fast memory with cost of cheap, slow memory

Why Does a Hierarchy Work?

- **Locality of reference**
  - Temporal locality
    - Reference same memory location repeatedly
  - Spatial locality
    - Reference near neighbors around the same time

- **Empirically observed**
  - Significant!
  - Even small local storage (8KB) often satisfies >90% of references to multi-MB data set

Memory Reference Patterns



MIT/UCB

Four Key Issues

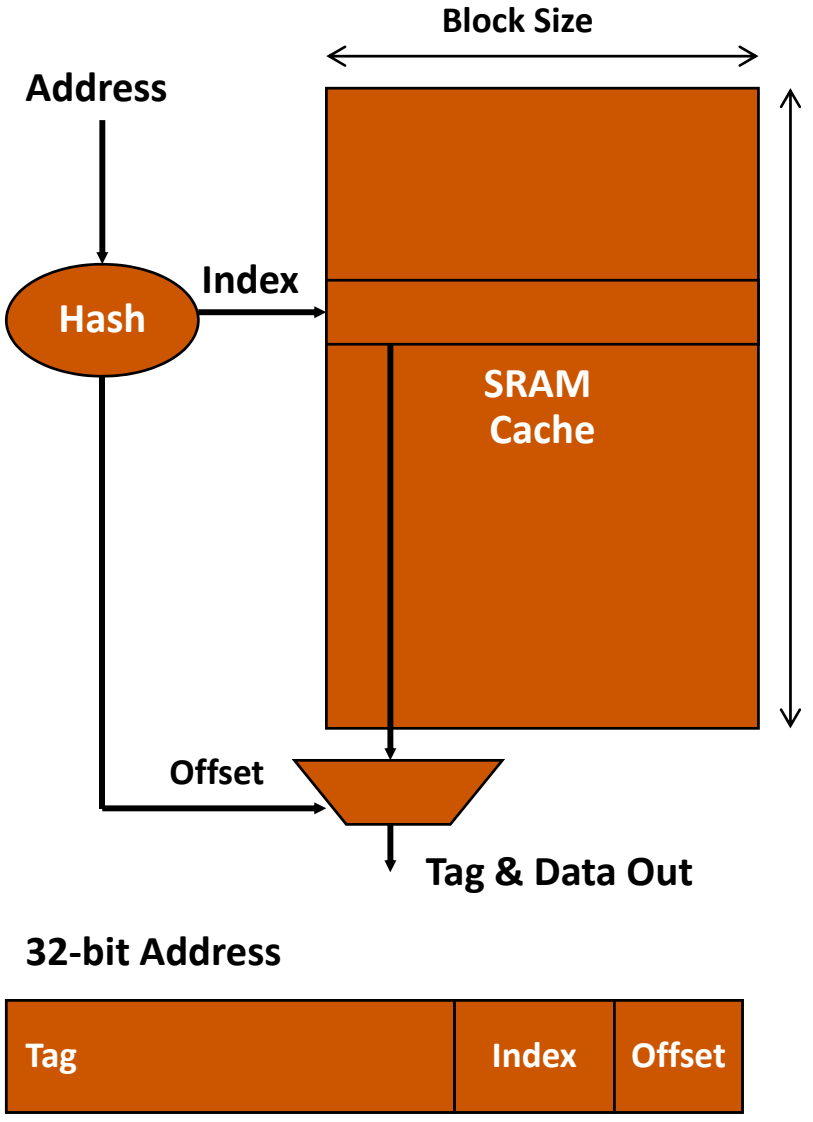
- These are:
  - Placement
    - Where can a block of memory go?
  - Identification
    - How do I find a block of memory?
  - Replacement
    - How do I make space for new blocks?
  - Write Policy
    - How do I propagate changes?
- Generally these are for caches
  - Usually SRAM
- Also apply to main memory, flash, disks

Placement

Memory Type	Placement	Comments
Registers	Anywhere; Int, FP, SPR	Compiler/programmer manages
Cache (SRAM)	Fixed in H/W	*Direct-mapped, set-associative, fully-associative*
DRAM	Anywhere	O/S manages
Disk	Anywhere	O/S manages

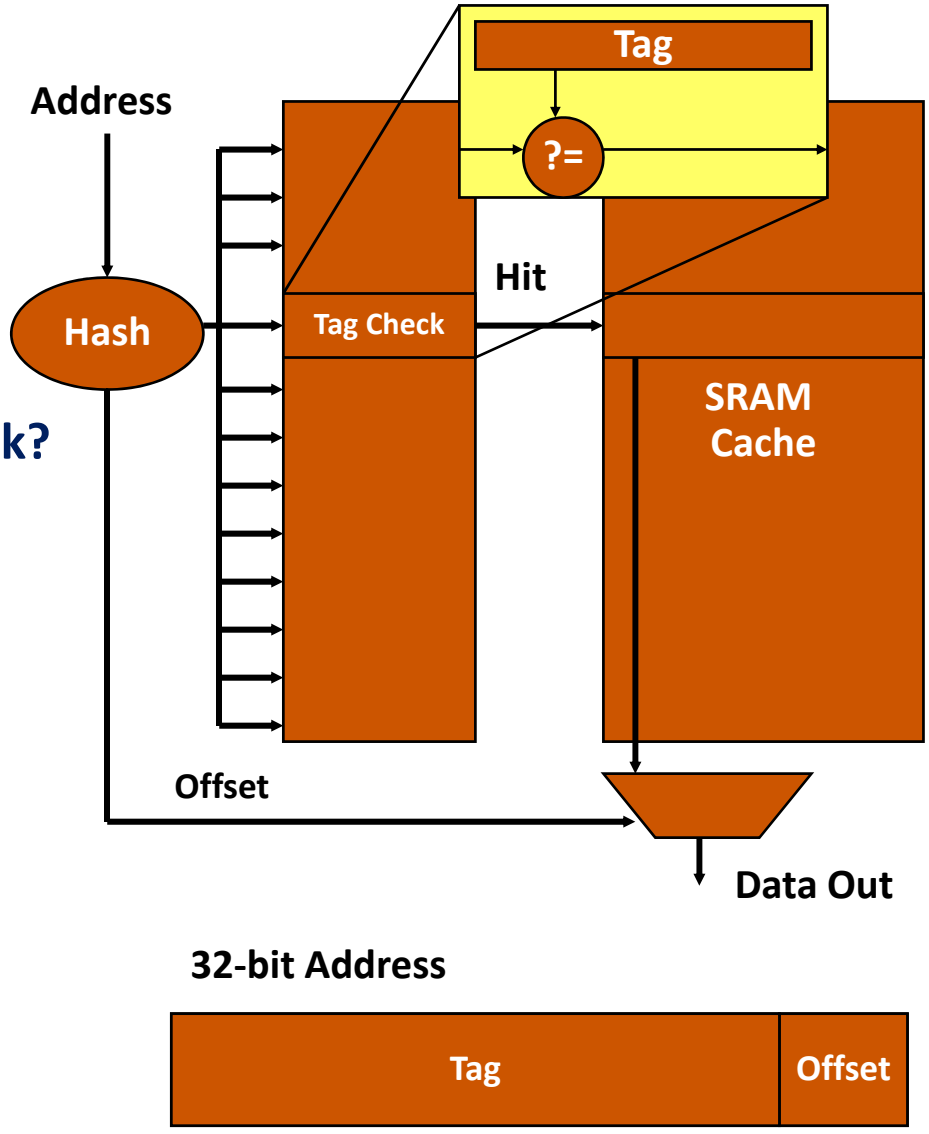
Placement

- **Address Range**
  - Exceeds cache capacity
- **Map address to finite capacity**
  - Called a hash
  - Usually just masks high-order bits
- ***Direct-mapped***
  - Block can only exist in one location
  - Hash collisions cause problems
  - Must check tag (identification)



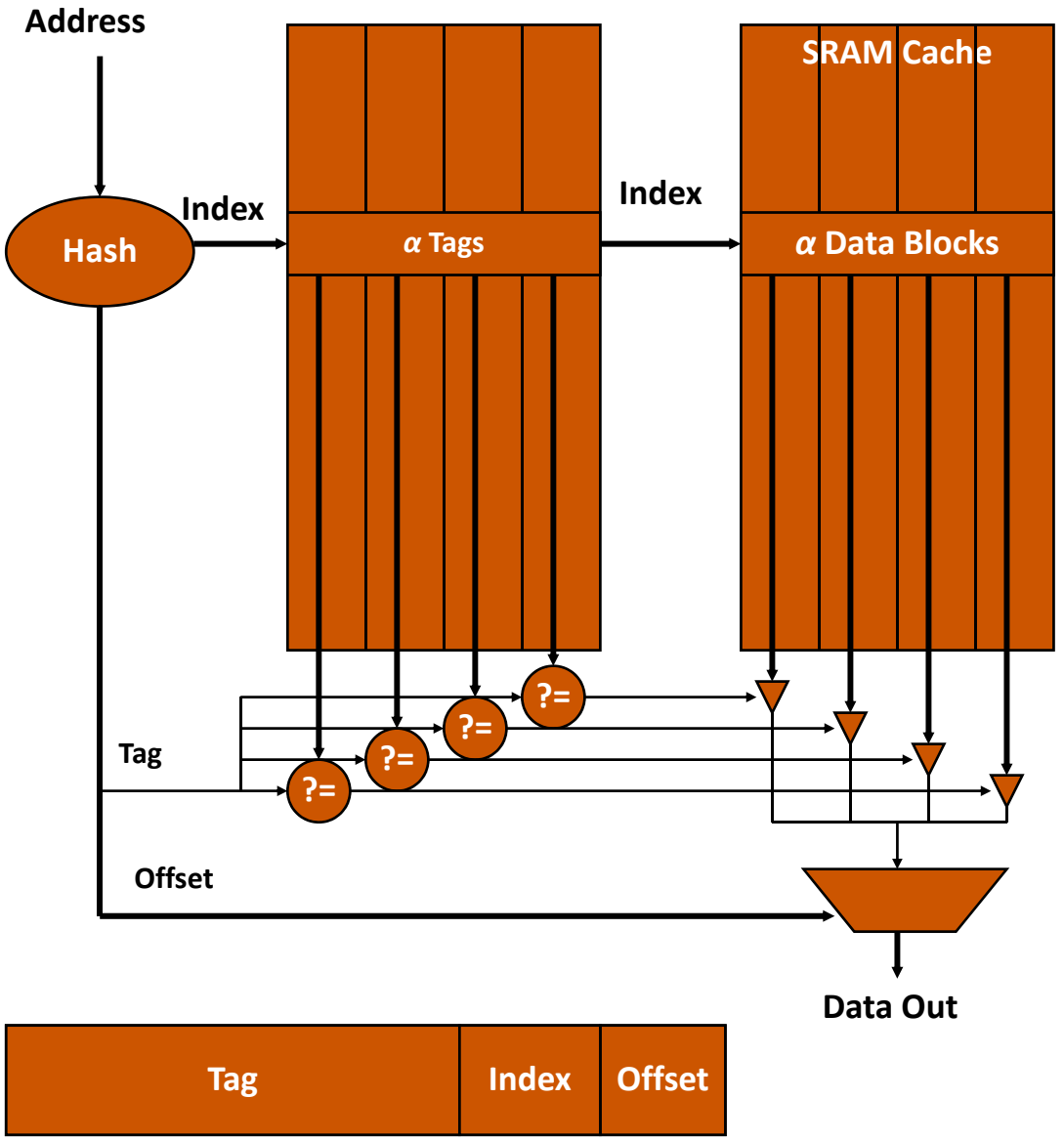
Identification

- Fully-associative
  - Block can exist anywhere
  - No more hash collisions
- Identification
  - How do I know I have the right block?
  - Called a tag check
    - Must store address tags
    - Compare against address
- Expensive!
  - Tag & comparator per block



Placement and Identification

- *Set-associative*
  - Block can be in *α* locations
  - Hash collisions:
    - Up to *α* still OK
- *Identification*
  - Still perform *tag check*
  - However, only *a* in parallel



Replacement

- Cache has finite size
  - What do we do when it is full?
- Analogy: desktop full?
  - Move books to bookshelf to make room
- Same idea:
  - Move blocks to next level of cache

Replacement

- How do we choose victim?
  - Verbs: Victimize, evict, replace, cast out
- Several policies are possible
  - FIFO (first-in-first-out)
  - LRU (least recently used)
  - NMRU (not most recently used)
  - Pseudo-random
- Pick victim within set where a = associativity
  - If a <= 2, LRU is cheap and easy (1 bit)
  - If a > 2, it gets harder
  - Pseudo-random works pretty well for caches

Write Policy

- Replication in memory hierarchy
  - 2 or more copies of same block
    - Main memory and/or disk
    - Caches
- What to do on a write?
  - Eventually, all copies must be changed
  - Write must propagate to all levels

Write Policy

- Easiest policy: write-through
- Every write propagates directly through hierarchy
  - Write in L1, L2, memory, disk (?!?)
- Why is this a bad idea?
  - Very high bandwidth requirement
  - Remember, large memories are slow
- Popular in real systems only to the L2
  - Every write updates L1 and L2
  - Beyond L2, use write-back policy

Write Policy

- Most widely used: write-back
- Maintain state of each line in a cache
  - Invalid – not present in the cache
  - Clean – present, but not written (unmodified)
  - Dirty – present and written (modified)
- Store state in tag array, next to address tag
  - Mark dirty bit on a write
- On eviction, check dirty bit
  - If set, write back dirty line to next level
  - Called a writeback or castout

Write Policy

- Complications of write-back policy
  - Stale copies lower in the hierarchy
  - Must always check higher level for dirty copies before accessing copy in a lower level
- Not a big problem in uniprocessors
  - In multiprocessors: the cache coherence problem
- I/O devices that use DMA (direct memory access) can cause problems even in uniprocessors
  - Called coherent I/O
  - Must check caches for dirty copies before reading main memory

Caches and Performance

- Caches
  - Enable design for common case: cache hit
    - Pipeline tailored to handle cache hits efficiently
    - Cache organization determines access latency, cycle time
  - Uncommon case: cache miss
    - Stall pipeline
    - Fetch from next level
      - Apply recursively if multiple levels
- What is performance impact?

Cache Misses and Performance

- **Miss penalty**
  - Detect miss: 1 or more cycles
  - Find victim (replace line): 1 or more cycles
    - Write back if dirty
  - Request line from next level: several cycles
  - Transfer line from next level: several cycles
    - (block size) / (bus width)
  - Fill line into data array, update tag array: 1+ cycles
  - Resume execution
- **In practice: 6 cycles to 100s of cycles**

Cache Miss Rate

- **Determined by:**
  - Program characteristics
    - Temporal locality
    - Spatial locality
  - Cache organization
    - Block size, associativity, number of sets
- **Measured:**
  - In hardware
  - Using simulation
  - Analytically

Cache Misses and Performance

- How does this affect performance?
- Performance = Time / Program

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

(code size) (CPI) (cycle time)

- Cache organization affects cycle time
  - Hit latency
- Cache misses affect CPI

Cache Misses and CPI

$$CPI = \frac{cycles}{inst} = \frac{cycles_{hit}}{inst} + \frac{cycles_{miss}}{inst}$$
$$= \frac{cycles_{hit}}{inst} + \frac{cycles}{miss} \times \frac{miss}{inst}$$
$$= \frac{cycles_{hit}}{inst} + Miss\_penalty \times Miss\_rate$$

- Cycles spent handling misses are strictly additive
- Miss_penalty is recursively defined at next level of cache hierarchy as weighted sum of hit latency and miss latency

Cache Misses and CPI

$$CPI = \frac{cycles_{hit}}{inst} + \sum_{l=1}^{n} P_l \times MPI_l$$

- Pl is miss penalty at each of n levels of cache
- MPI_l is miss rate per instruction at each of n levels of cache
- Miss rate specification:
  - Per instruction: easy to incorporate in CPI
  - Per reference: must convert to per instruction
    - Local: misses per local reference
    - Global: misses per ifetch or load or store

Cache Performance Example

- **Assume following:**
  - L1 instruction cache with 98% per instruction hit rate
  - L1 data cache with 96% per instruction hit rate
  - Shared L2 cache with 40% local miss rate
  - L1 miss penalty of 8 cycles
  - L2 miss penalty of:
    - 10 cycles latency to request word from memory
    - 2 cycles per 16B bus transfer, 4x16B = 64B block transferred
    - Hence 8 cycles transfer plus 1 cycle to fill L2
    - Total penalty 10+8+1 = 19 cycles

Cache Performance Example

$$CPI = \frac{cycles_{hit}}{inst} + \sum_{l=1}^{n} P_l \times MPI_l$$

$$CPI = 1.15 + \frac{8cycles}{miss} \times \left(\frac{0.02miss}{inst} + \frac{0.04miss}{inst} \right)$$

$$+ \frac{19cycles}{miss} \times \frac{0.40miss}{ref} \times \frac{0.06ref}{inst}$$

$$= 1.15 + 0.48 + \frac{19cycles}{miss} \times \frac{0.024miss}{inst}$$

$$= 1.15 + 0.48 + 0.456 = 2.086$$

Cache Misses and Performance

- **CPI equation**
  - Only holds for misses that cannot be overlapped with other activity
  - Store misses often overlapped
    - Place store in store queue
    - Wait for miss to complete
    - Perform store
    - Allow subsequent instructions to continue in parallel
  - Modern out-of-order processors also do this for loads
    - Cache performance modeling requires detailed modeling of entire processor core

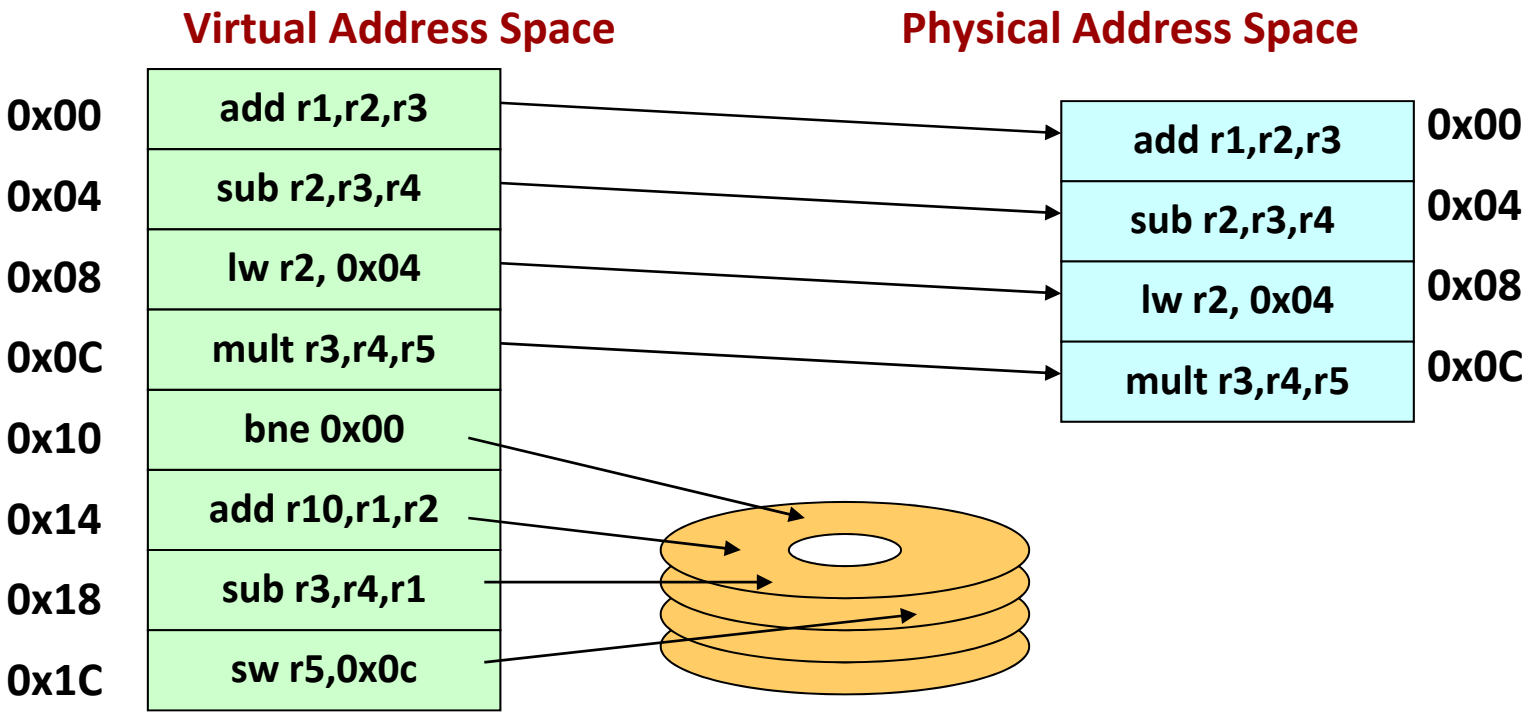
Virtual Memory

- **What is virtual memory?**
  - Technique that allows execution of a program that may not completely reside in memory (RAM)
- **Allows the computer to "fake"' a program into believing that its memory space is larger than physical RAM**
- **Why is VM important?**
  - Cheap - no longer have to buy lots of RAM
  - Removes burden of memory resource management from the programmer
  - Other benefits ...

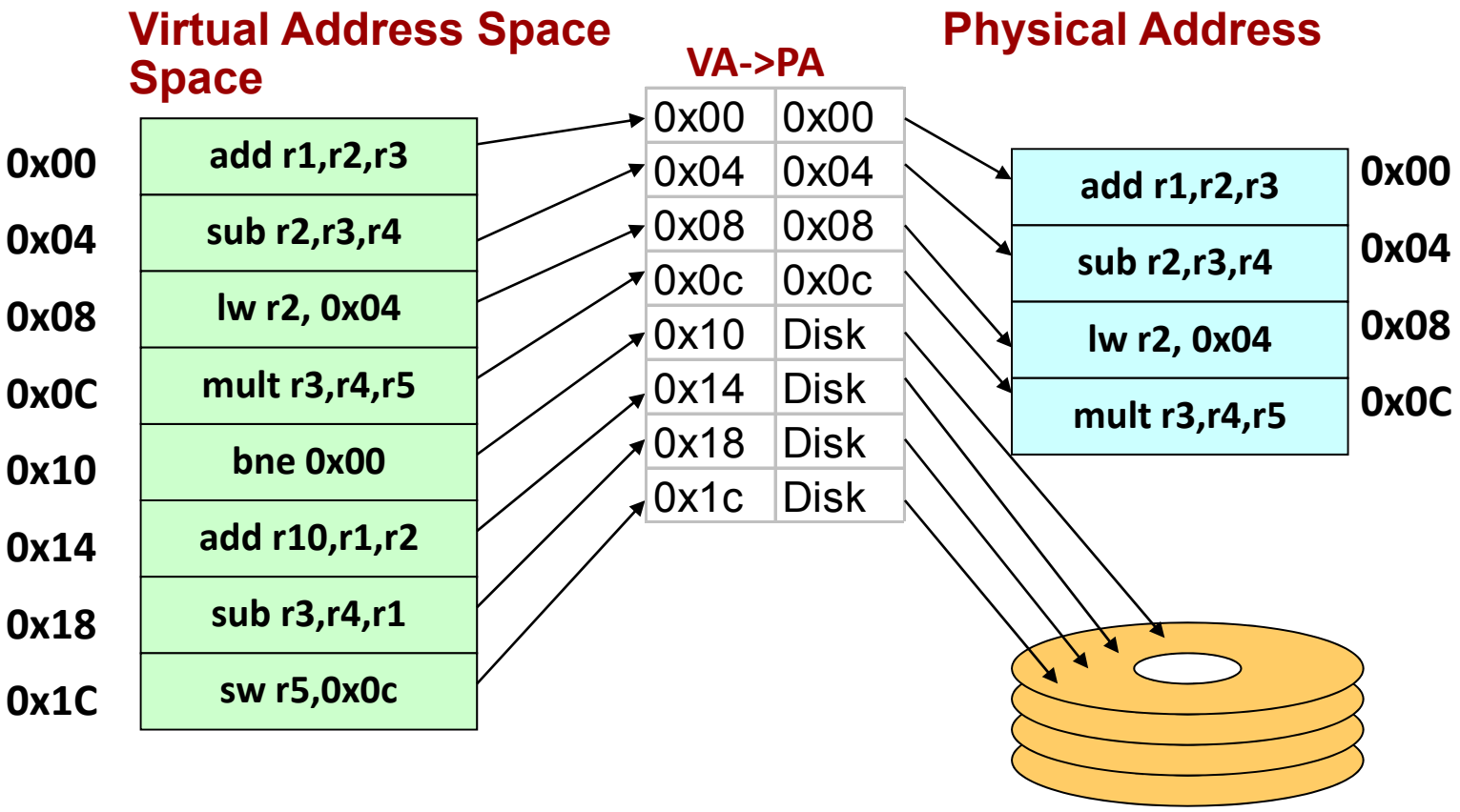
How Does VM Work ?

- Two memory "spaces"
  - Virtual memory space - what the program "sees"
  - Physical memory space - what the program runs in (size of RAM)
- On program startup
  - OS copies program into RAM
  - If there is not enough RAM, OS stops copying program and starts it running with only a portion of the program loaded in RAM
  - When the program touches a part of the program not in physical memory (RAM), OS catches the memory abort (called a page fault) and copies that part of the program from disk into RAM
  - In order to copy some of the program from disk to RAM, OS must evict parts of the program already in RAM
    - OS copies the evicted parts of the program back to disk

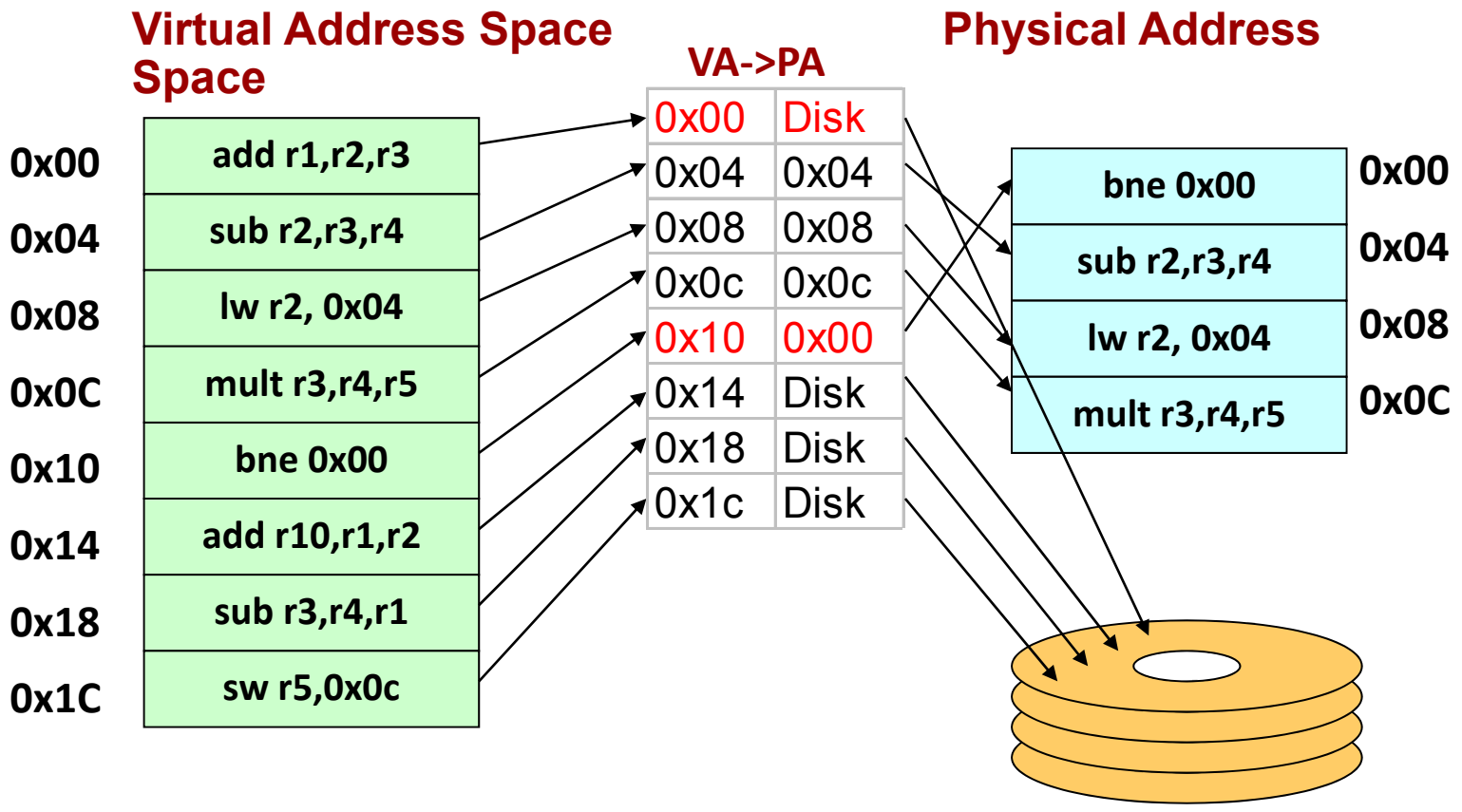
Example: Virtual and Physical Address Spaces



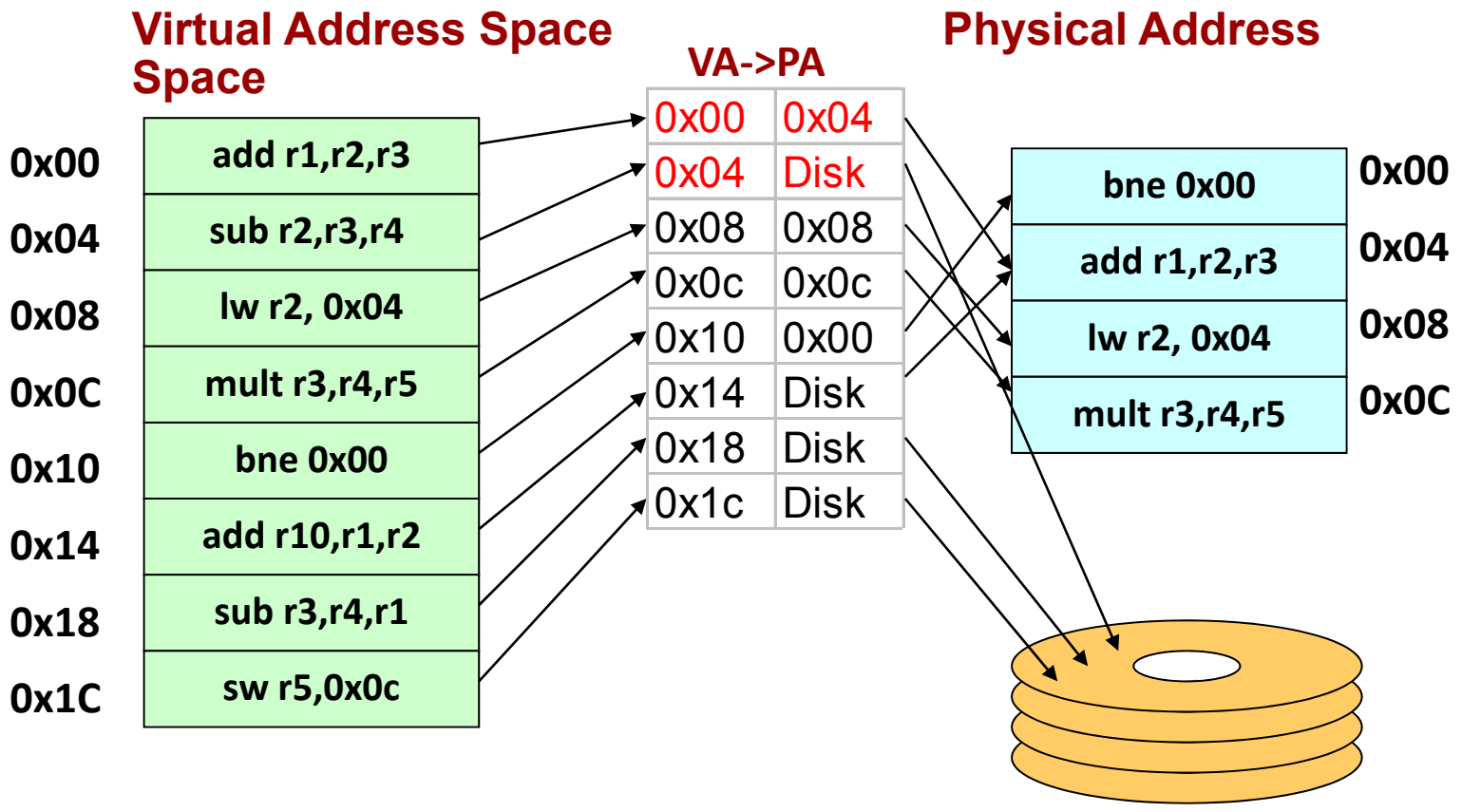
Example (cont): Need VA-to-PA mappings



Example (cont): After handling a page fault

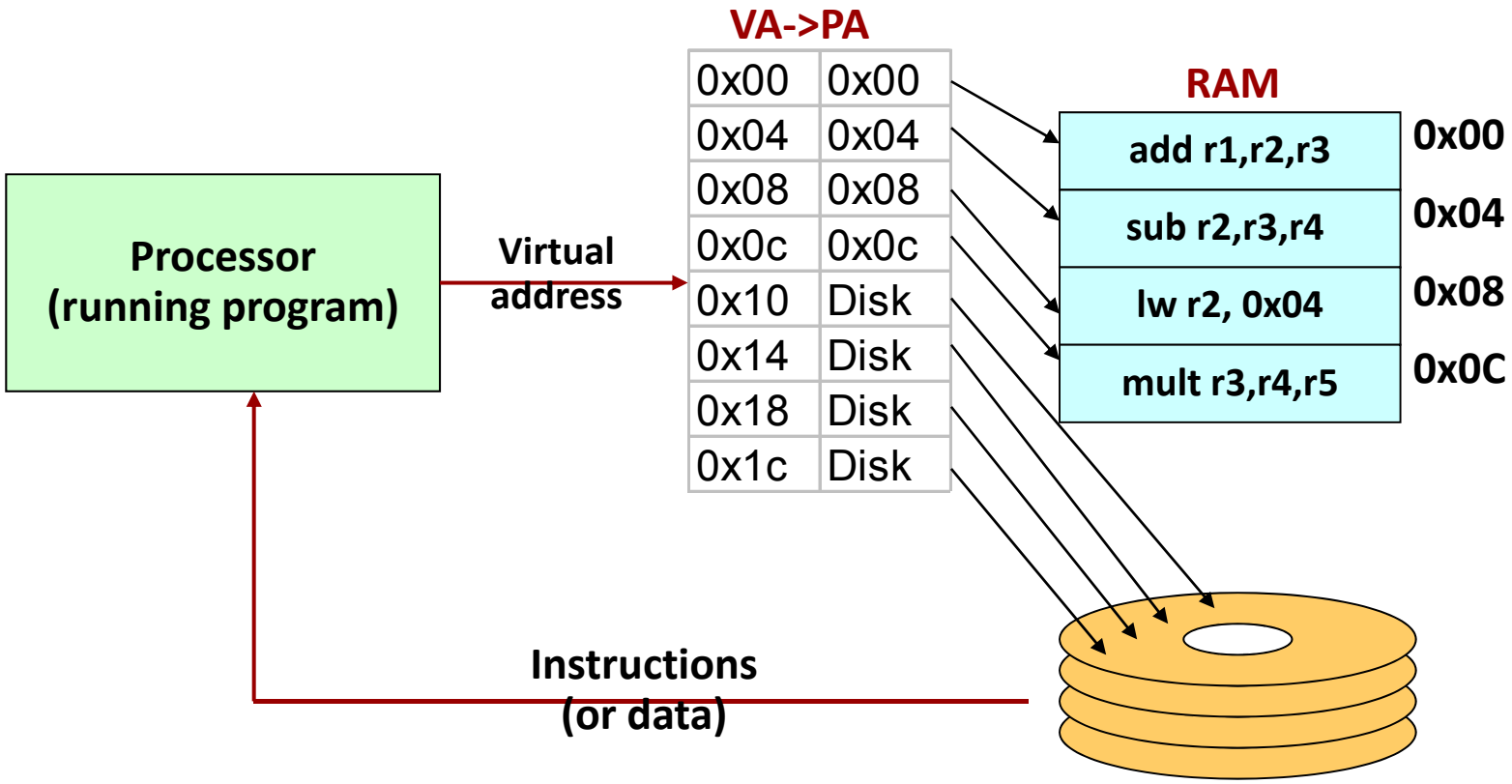


Example (cont): After a second page fault



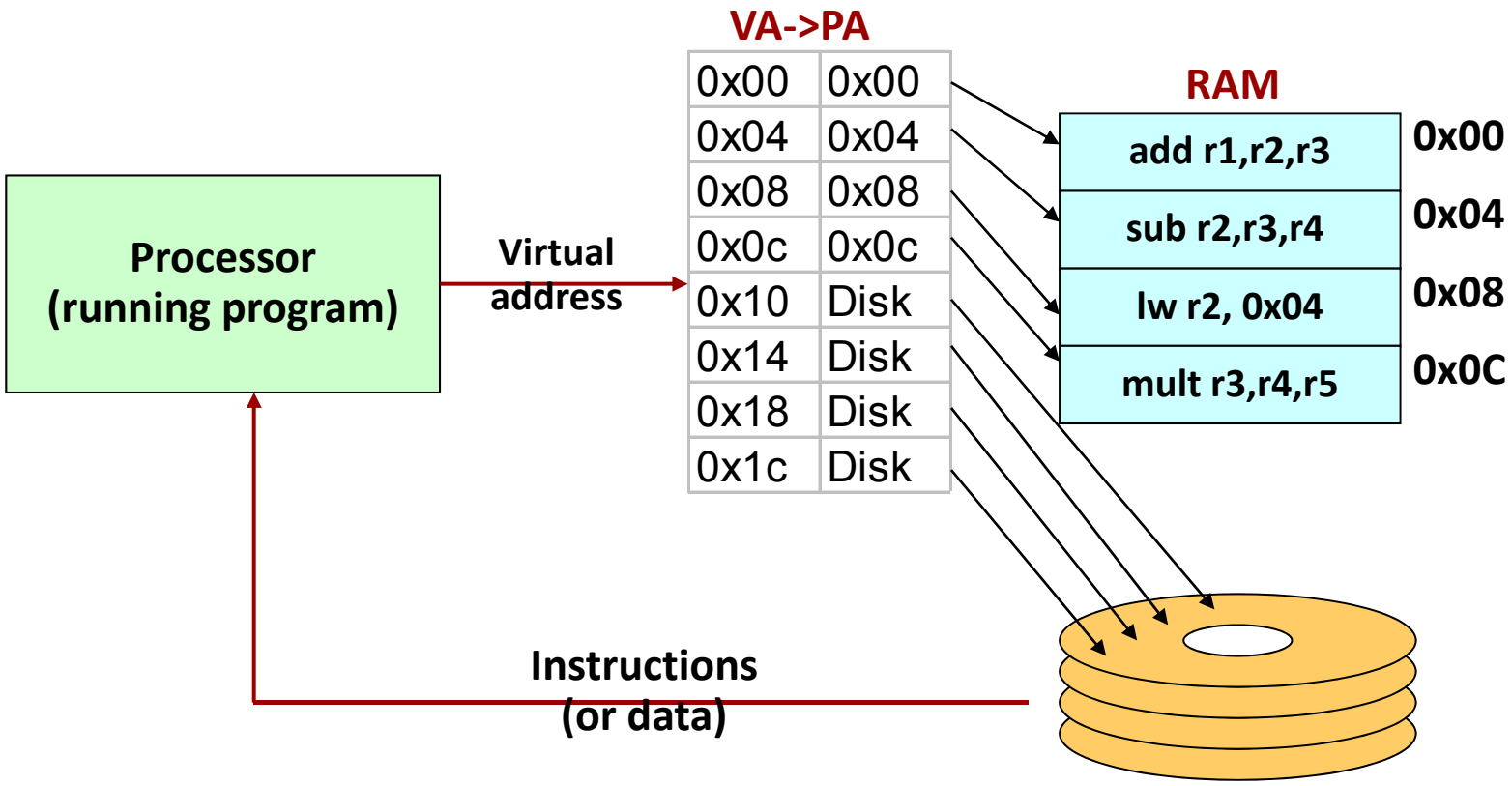
Basic VM Algorithm

- Program asks for virtual address
- Computer translates virtual address (VA) to physical address (PA)
- Computer reads PA from RAM, returning it to program



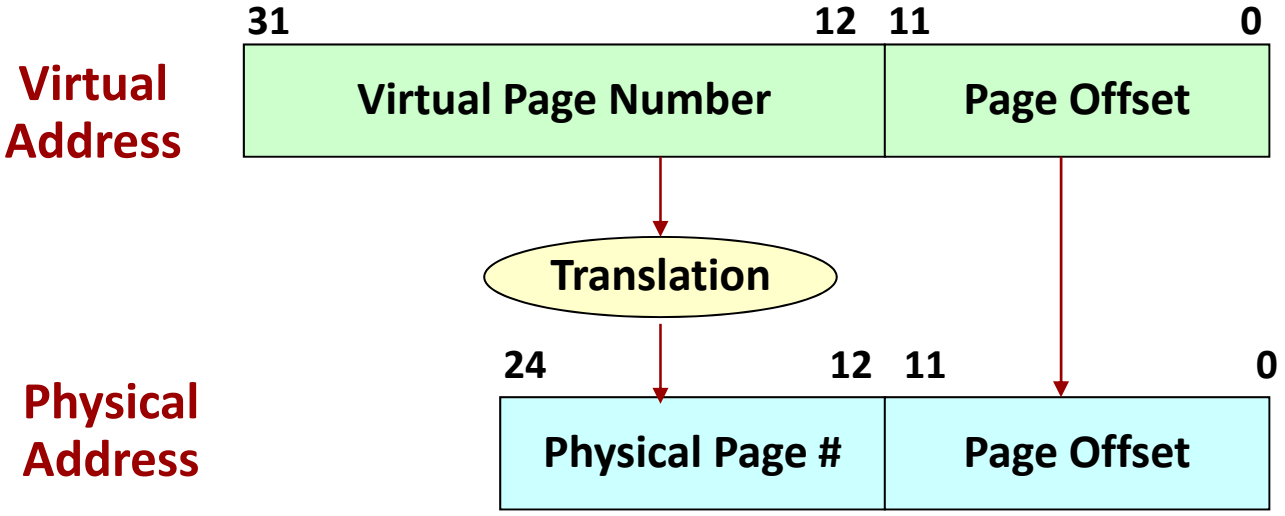
Page Tables

- Table which holds VA -> PA translations is called the page table
- In our current scheme, each word is translated from a virtual address to a physical address
  - How big is the page table?

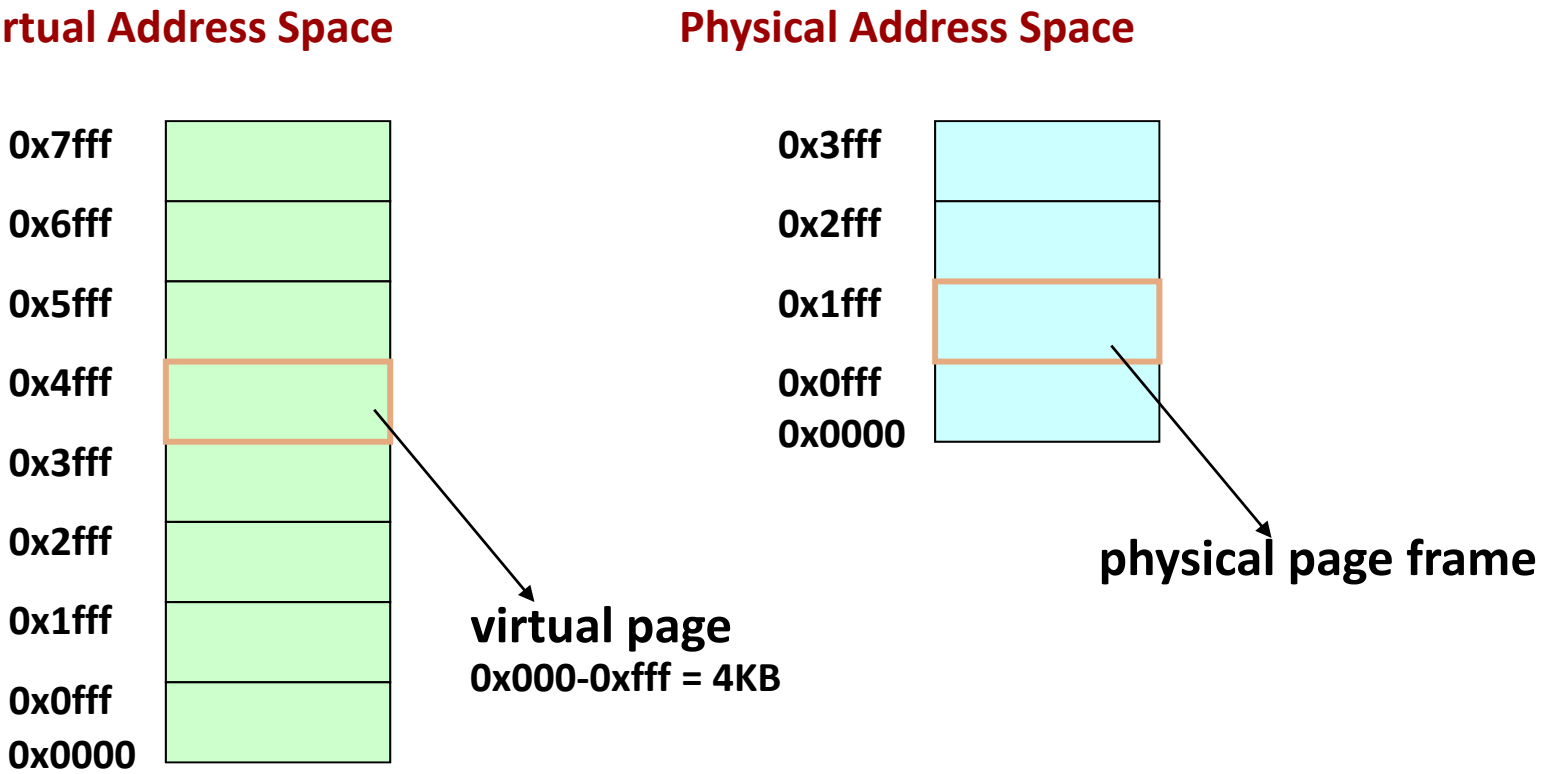


Real Page Tables

- Instead of the fine-grained VM where any virtual word can map to any RAM word location, partition memory into chunks called pages
  - Typical page size today is 4 or 8 KBytes
- This reduces the number of VA-> PA translation entries
  - Only one translation per page
  - For a 4 KByte page, that's one VA-> PA translation for every 1,024 words
- *Within* a page, the virtual address == physical address



Virtual Pages & Physical Page Frames

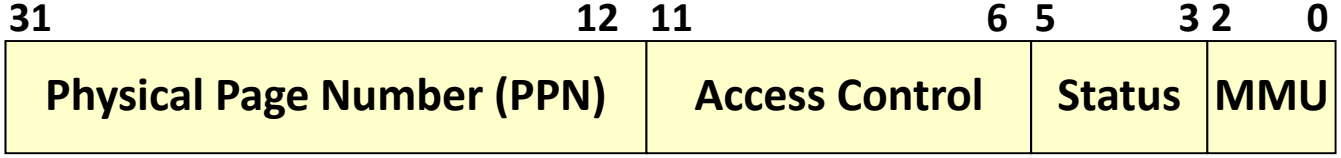


Page Frames

- Every address within a virtual page maps to the same location within a physical page frame
  - In other words, bottom log 2(page size in bytes) is not translated

Page Table Entries

- A real page table entry (PTE) contains a number of fields
  - Physical page number
  - Access control bits (e.g., writeable bit)
  - Status bits (e.g., accessed and dirty bits)
  - MMU control bits (e.g., cacheable and bufferable bits)
- Why is the virtual page number not in the page table entry?



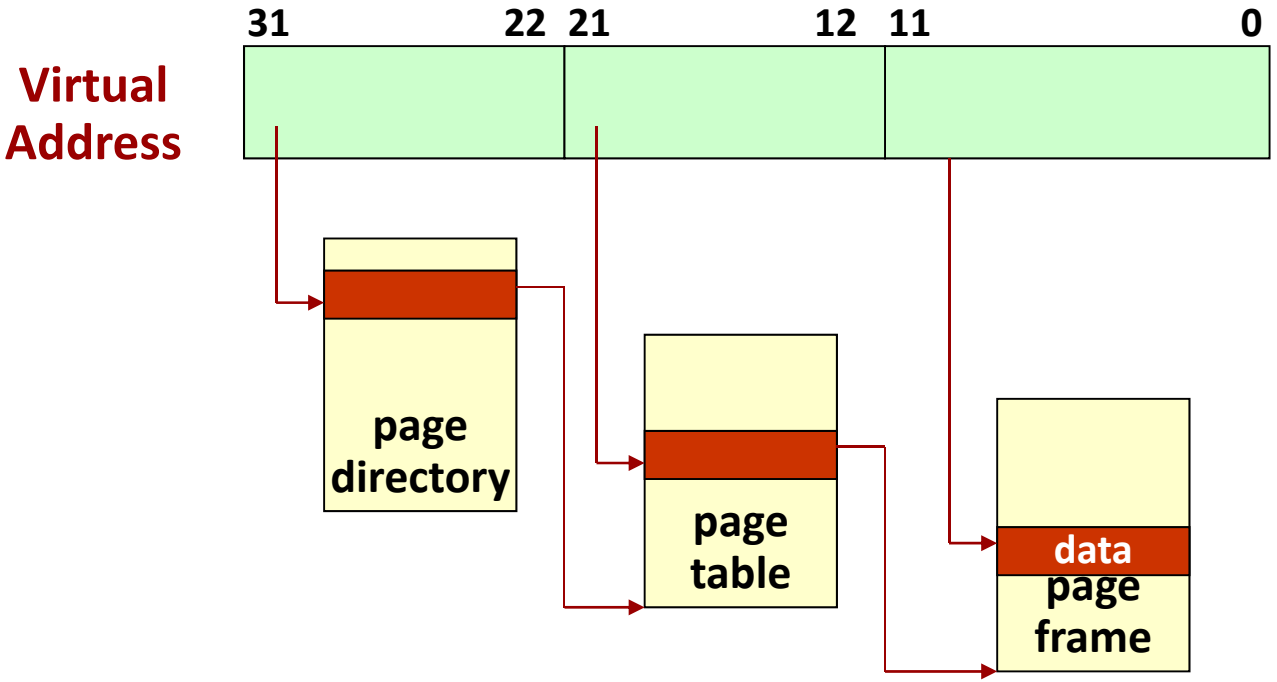
Interactions with Instruction/Data Cache

- Page table entry has bits to determine if region is
  - cacheable vs . uncacheable
  - write-through vs. write-back
- Why control caching?
  - Some regions, such as for memory-mapped I/O devices, should not be cached. Why?
    - Because cache doesn't know if I/O device register has changed value

Interactions with Write Buffer

- Page table entry has bits to determine if region
  - Allows write buffer to buffer writes
- Why would one not want to buffer writes?
  - Possible problems with memory-mapped I/O devices that require a write to complete before subsequent commands (instructions) are issued

Two-level Page Tables

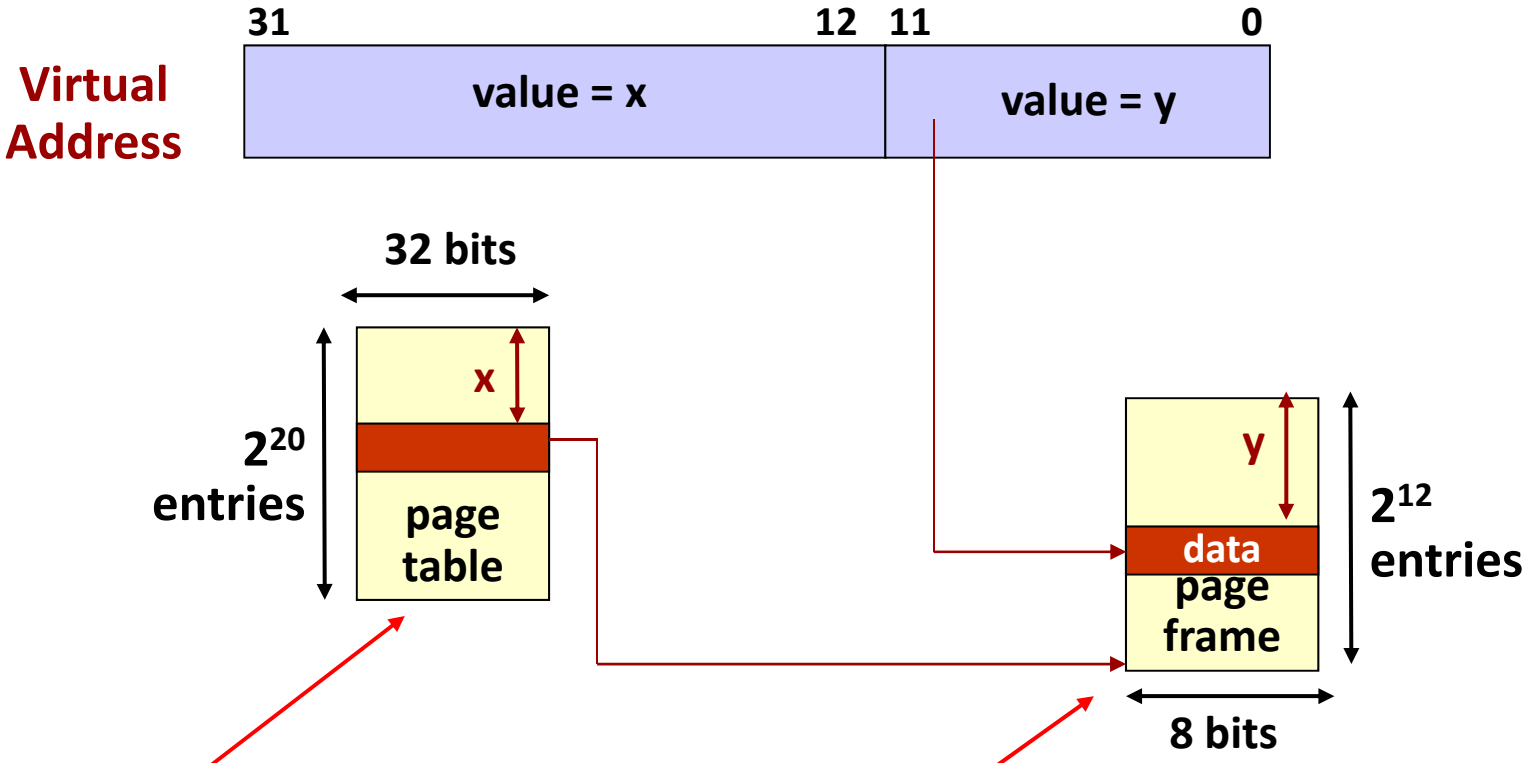


ARM MMU

- Complex VM and protection mechanisms
- Presents 4 GB address space
- Memory granularity: 3 options supported
  - 1MB sections
  - Large pages (64 KBytes) - access control within a large page on 16 KBytes
  - Small pages (4 KBytes) - access control within a large page on 1 Kbytes
- Puts processor in Abort Mode when virtual address not mapped or permission check fails
- Change pointer to page tables (called the translation table base, in ARM jargon) to change virtual address space
  - useful for context switching of processes

ARM, Inc.

Example: Single-Level Page Table



ARM, Inc.

Single-Level Page Table

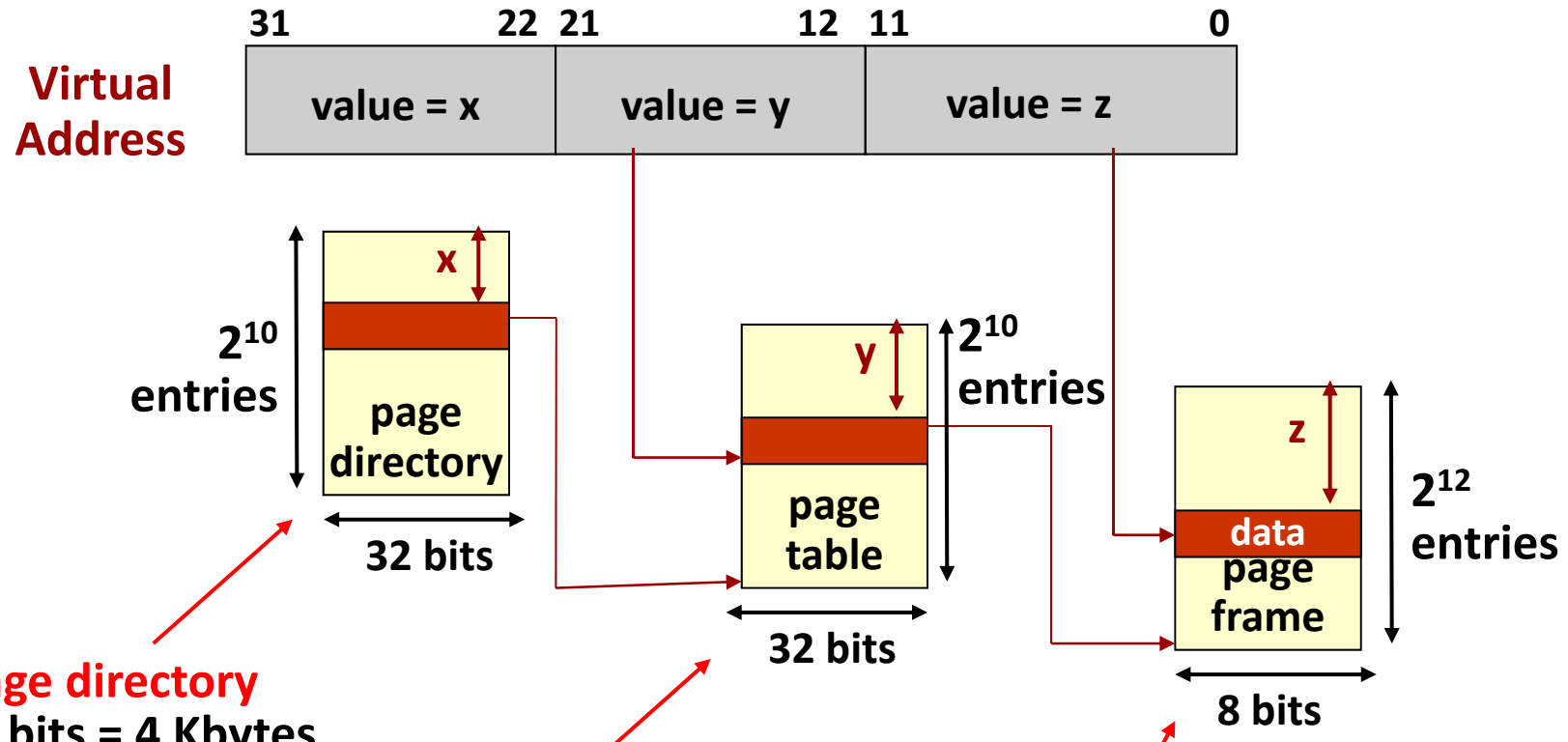
- **Assumptions**
  - 32-bit virtual addresses
  - 4 Kbyte page size = 2^{12} bytes
  - 32-bit address space

- **How many virtual page numbers?**
  - $2^{32} / 2^{12} = 2^{20}$ = 1,048,576 virtual page numbers = number of entries in the page table

- **If each page table entry occupies 4 bytes, how much memory is needed to store the page table?**
  - 2^{20} entries * 4 bytes = 2^{22} bytes = 4 Mbytes

ARM, Inc.

Example: Two-level Page Table



Size of page directory
= 2^{10} * 32 bits = 4 Kbytes

Size of page table
= 2^{10} * 32 bits = 4 Kbytes

Size of page
= 2^{12} * 8 bits = 4 Kbytes

ARM, Inc.

Two-Level Page Table

- **Assumptions**
  - 2^{10} entries in page directory (= max number of page tables)
  - 2^{10} entries in page table
  - 32 bits allocated for each page directory entry
  - 32 bits allocated for each page table entry

- **How much memory is needed?**
  - Page table size = 2^{10} entries * 32 bits = 2^{12} bytes = 4 Kbytes
  - Page directory size = 2^{10} entries * 32 bits = 2^{12} bytes = 4 Kbytes

ARM, Inc.

Two-Level Page Table

- **Small (typical) system**
  - One page table might be enough
    - Page directory size + Page table size = 8 Kbytes of memory would suffice for virtual memory management
  - How much *physical* memory could *this one page table* handle?
    - Number of page tables * Number of page table entries * Page size = $1 * 2^{10} * 2^{12}$ bytes = 4 Mbytes

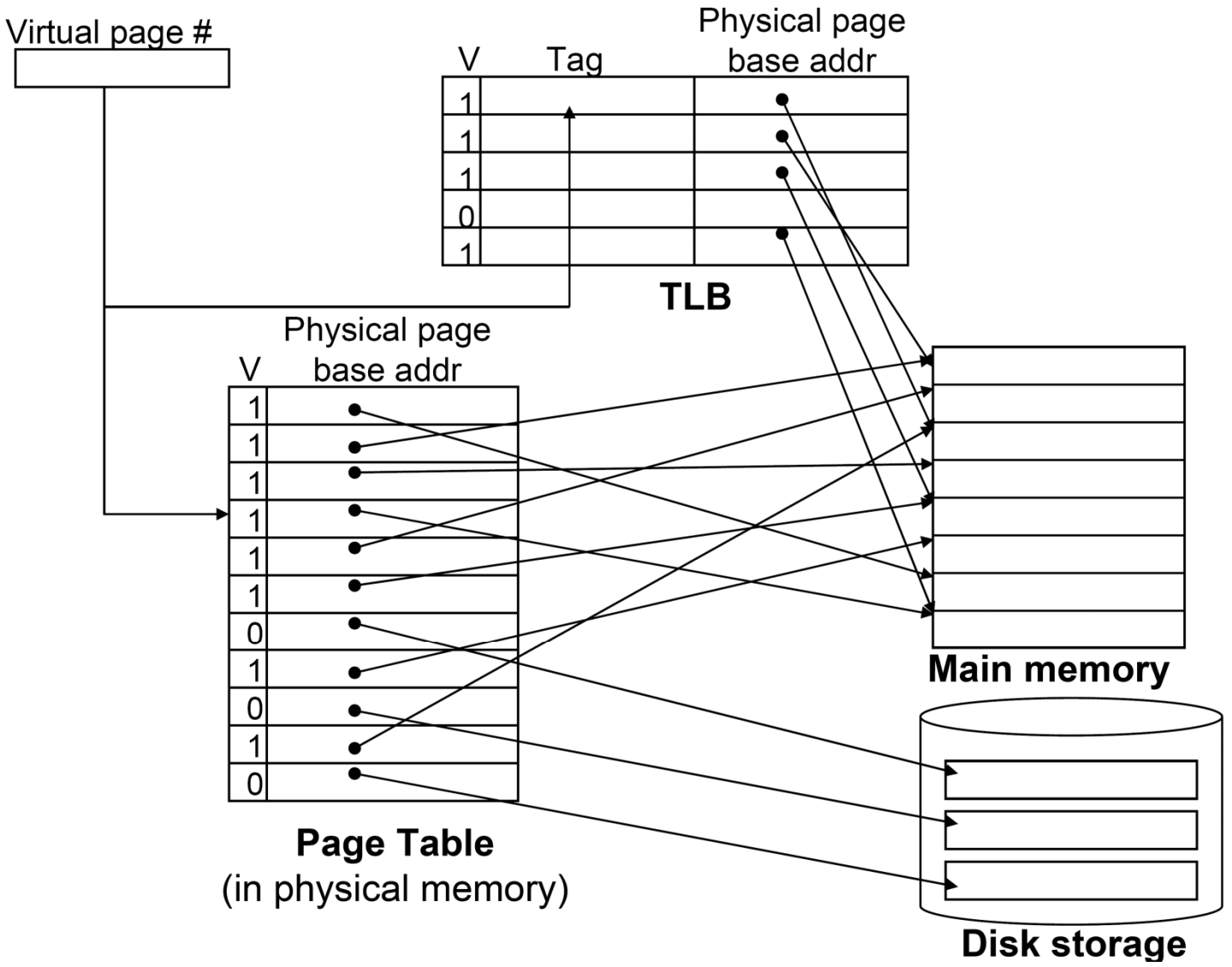
- **Large system**
  - You might need the maximum number of page tables
    - Max number of page tables * Page table size = 2^{10} directory entries * 2^{12} bytes = 2^{22} bytes = 4 Mbytes of memory would be needed for virtual memory management
  - How much *physical* memory could *these 2^{10} page tables* handle?
    - Number of page tables * Number of page table entries * Page size = $2^{10} * 2^{10} * 2^{12}$ bytes = 4 Gbytes

ARM, Inc.

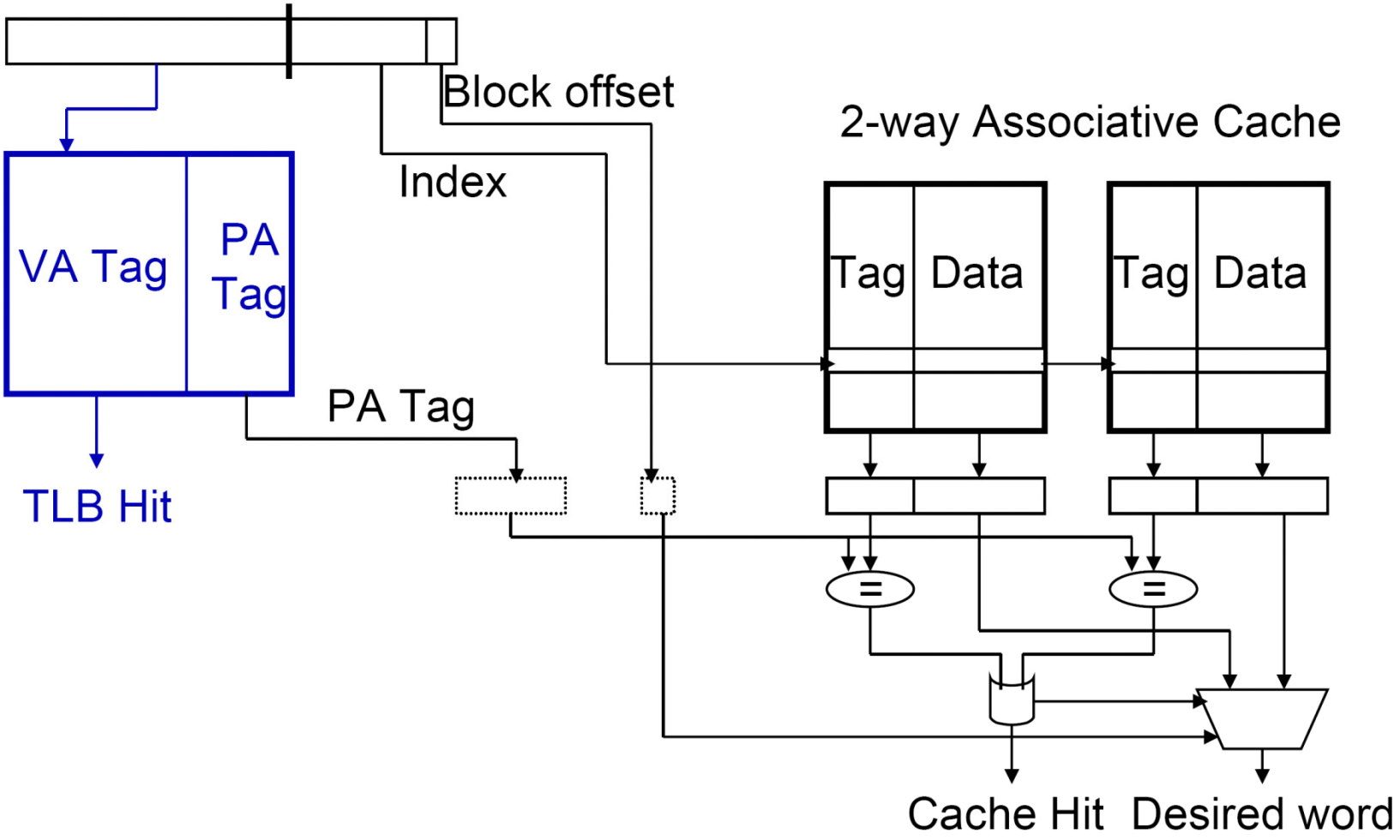
Translation Lookaside Buffer (TLB)

- Each virtual memory reference can cause two physical memory accesses
  - One to fetch the page table
  - One to fetch the data
- To overcome this problem a high-speed cache is set up for page table entries called the TLB - Translation Lookaside Buffer
- Contains page table entries that have been most recently used
  - Functions same way as a memory cache
- Given a virtual address, processor examines the TLB
  - If page table entry is present (a hit), the frame number is retrieved and the real address is formed
  - If page table entry is not found in the TLB (a miss), the page number is used to index the process page table

TLB Block Diagram



TLB + 2-Way Set Associative Cache Architecture



Backup

ALPHA 21264 Tournament Branch Predictor

The following is a summary of the predictor design:

Local history table	1024 entries, indexed by PC; each entry is a 10-bit history of a specific branch's set of previous outcomes
Local prediction table	1024 entries, indexed by the 10-bit history from the local history table, each entry is a 3-bit saturating counter
Global history register	A single 12-bit register that encodes the last 12 branch outcomes
Global prediction table	4096 entries, indexed by the global history register, each entry is a 2-bit saturating counter
Chooser prediction table	4096 entries, indexed by the global history register, each entry is a 2-bit saturating counter

The local history table maintains history entries for individual branches. Each 10-bit history (which encodes the last 10 outcomes for this branch) is used to index into a local prediction table, which maintains entries that have 3-bit saturating counters. When a 3-bit saturating counter reaches 1002 or above, a "taken" prediction is made.

The global history register maintains a single 12-bit history (which encodes the outcomes for the previous 12 branches) used to index into the global prediction table, which has 2-bit saturating counters per entry. When a 2-bit saturating counter reaches 102 or above, a "taken" prediction is made.

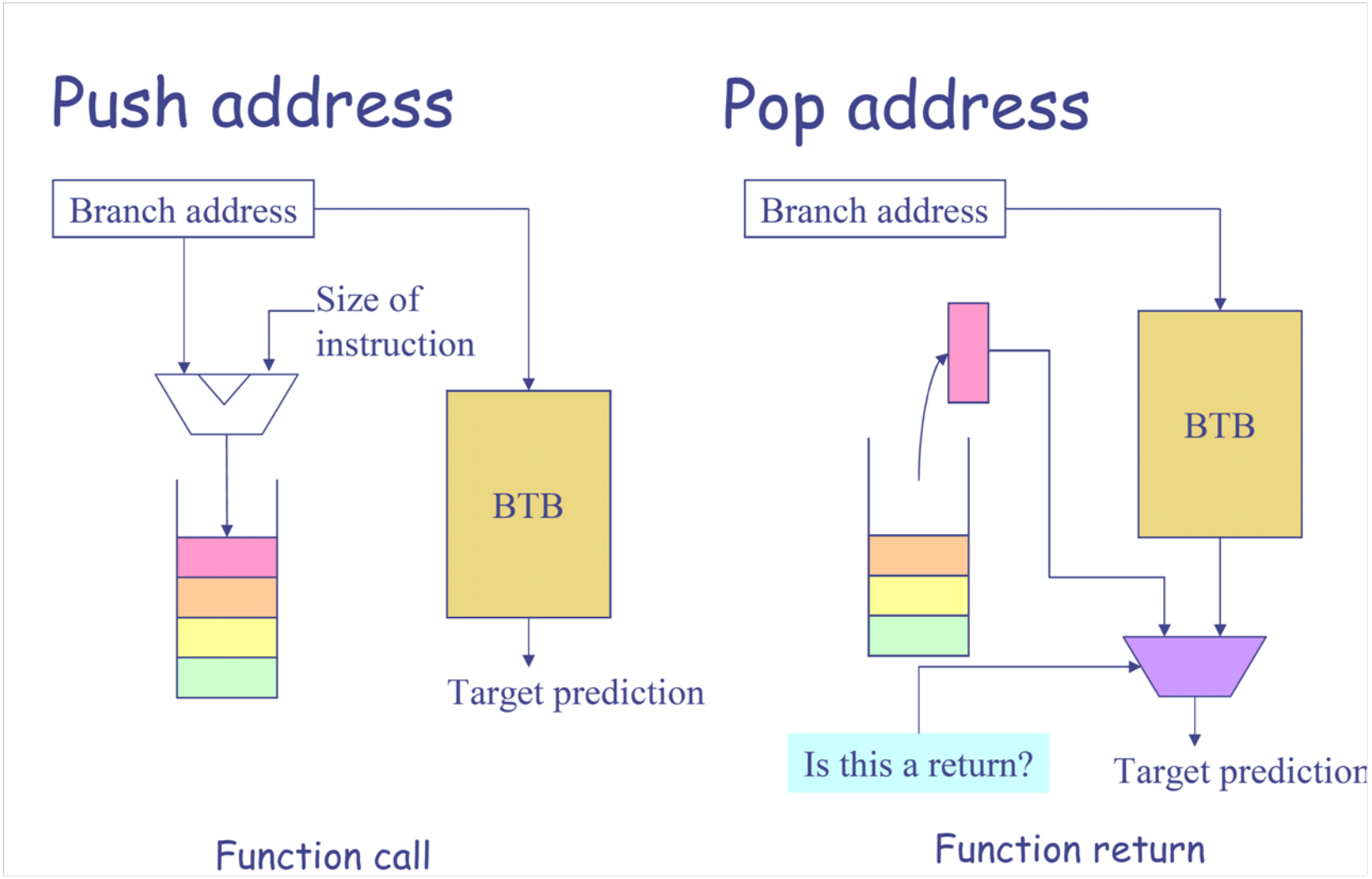
The local and global predictions are compared against one another, and if they agree, the prediction is made. However, if they disagree, they should consult the chooser prediction table, which is also indexed by the global history register. If the saturating counter is 102 or above, the global predictor's decision should be used; otherwise, the local predictor's decision is picked. The chooser prediction table is updated depending on the outcome of disagreeing predictors. Depending on whether a predictor chose correctly, the saturating counter is incremented or decremented.

Kessler, R. E., "The Alpha 21264 Microprocessor", IEEE Micro, March/April 1999, pp. 24-36 (available at ieeexplore.ieee.org).

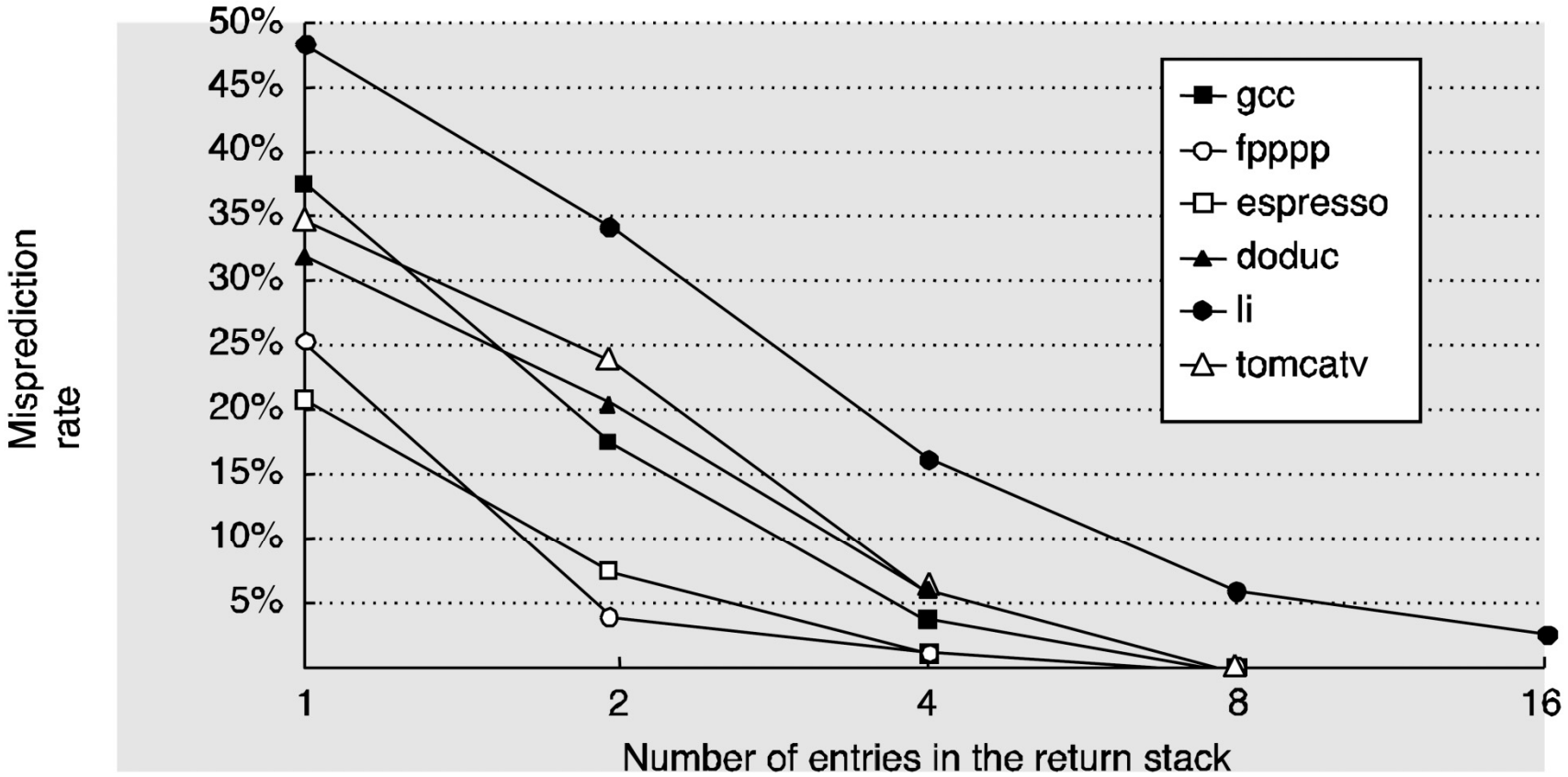
Return Addresses Prediction

- Register indirect branch hard to predict address
  - Many callers, one callee
  - Jump to multiple return addresses from a single address (no PC-target correlation)
- SPEC89 85% such branches for procedure return
- Since stack discipline for procedures, save return address in small buffer that acts like a stack: 8 to 16 entries has small miss rate

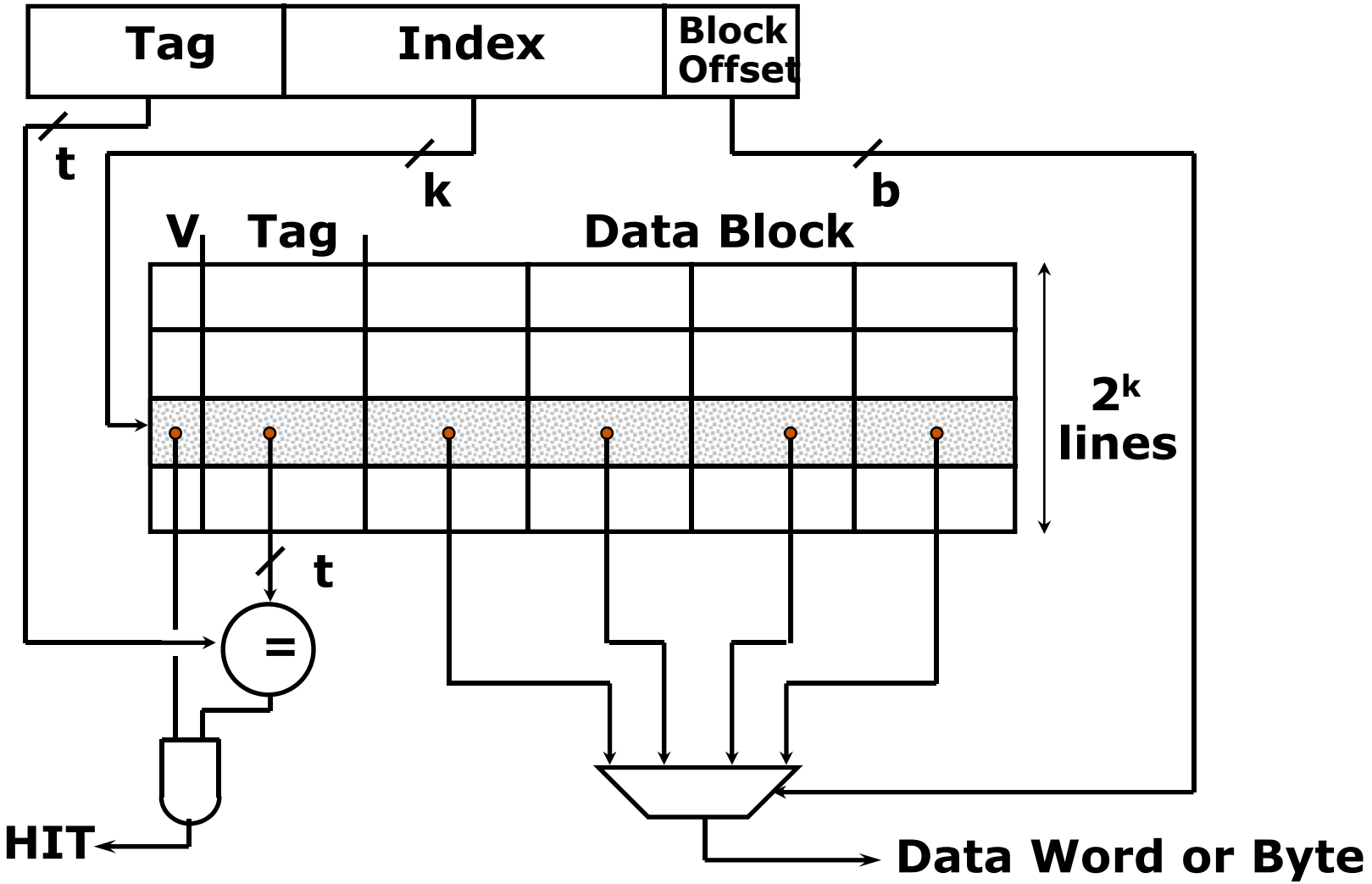
Return Address Stack Design



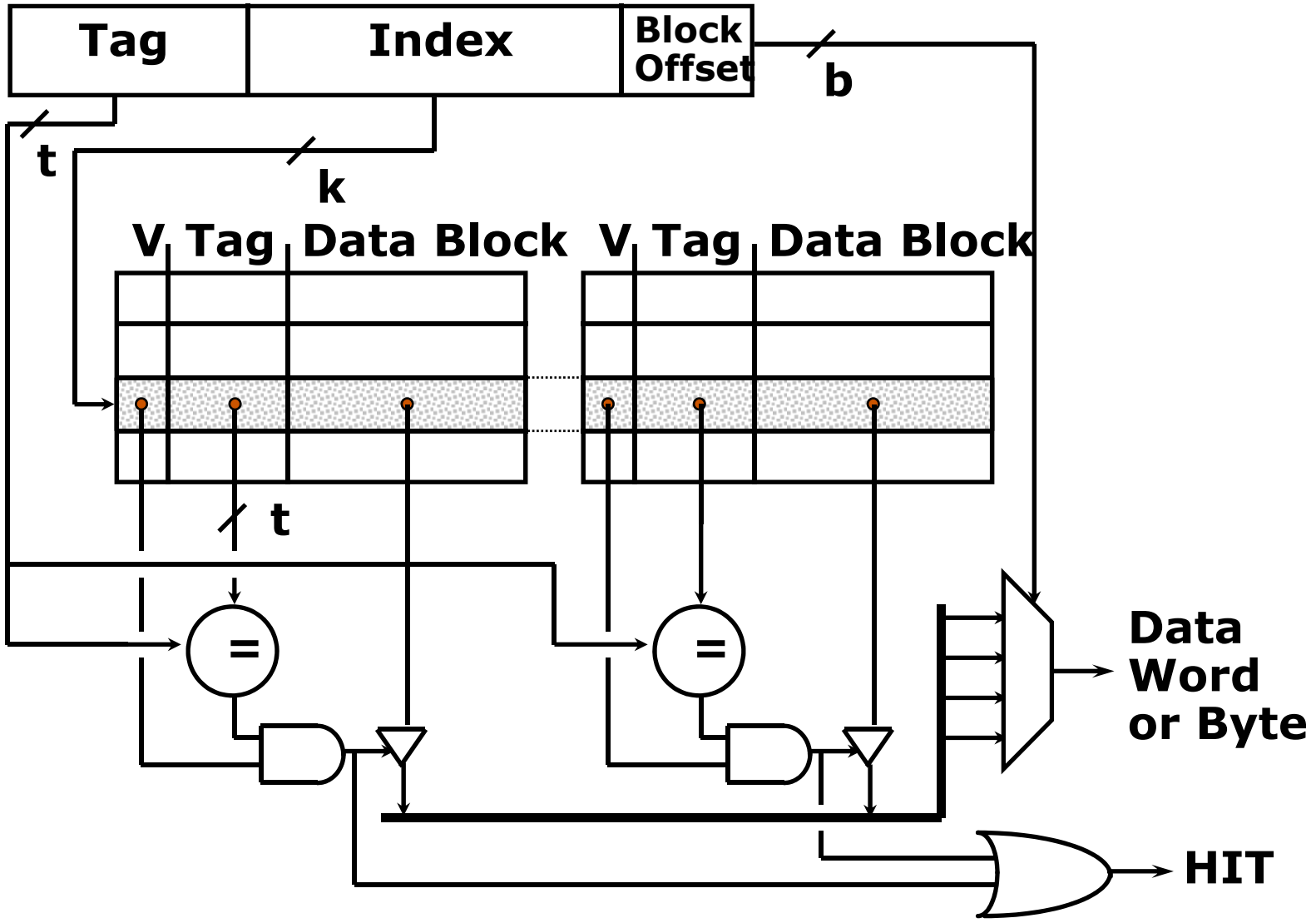
Accuracy of Return Address Predictor



Direct-Mapped Cache



2-Way Set-Associative Cache



Fully Associative Cache