

Empty Book Template

Ryan Hou

Invalid Date

Table of contents

Preface	4
Resources	5
1 Introduction	6
1.1 Perspective	6
1.2 Eight Great Ideas	6
1.3 High Level Ideas	6
2 Instruction Set Architecture	7
2.1 (Simplified) System Organization	8
2.2 ISA Design Space	10
3 Assembly	11
3.1 Basics	11
3.1.1 Instruction Encoding	11
3.1.2 Properties	11
3.2 Special Purpose Registers	11
3.3 Memory	12
3.4 Big Endian vs Little Endian	13
3.5 Memory Layout	14
3.5.1 Memory Layout of Variable	14
3.5.2 Structure Alignment	14
3.6 Control Flow	18
3.7 C to Assembly	18
3.8 Functions	18
3.8.1 Call and Return	18
7 Memory Hierarchy & Caching	22
7.1 Cache Basics	23
7.1.1 Basic Cache Design	24
7.2 Cache Operation	27
7.2.1 AMAT	27
7.2.2 Overheads	27
7.3 Tracking LRU	28
7.3.1 Pseudo LRU	28

7.4	Cache Blocks	28
7.4.1	Row vs Column Major	29
7.5	Stores: Write-Through vs Write-Back	30
7.6	Cache Mapping	32
7.6.1	Fully-Associative Cache	32
7.6.2	Direct Mapped Cache	33
7.6.3	Set-Associative Cache	34
7.6.4	Cache Associativity Summary	36
7.7	Cache Misses	38
7.7.1	Cache Parameters	39
7.8	Cache Summary	43
7.9	Examples	45
8	Virtual Memory	46
8.0.1	Page Faults	48
8.1	Hierarchical Page Tables	48
8.2	Translation Look-Aside Buffer (TLB)	49
8.3	Virtually-Addressed Caches	49
9	Summary	50
	References	51

Preface

My notes on ???.

Resources

Some relevant resources:

- [Resource Name](#)

Textbooks:

- [Book 1](#)

1 Introduction

1.1 Perspective

i Note 1: Definition - Some definition

Term is defined as blah blah blah...

1.2 Eight Great Ideas

Eight Great Ideas:

- Design for Moore's Law
- Use abstraction to simplify design
- Make the common case fast
- Performance via pipelining
- Performance via prediction
- Performance via parallelism
- Hierarchy of memories
- Dependability via redundancy

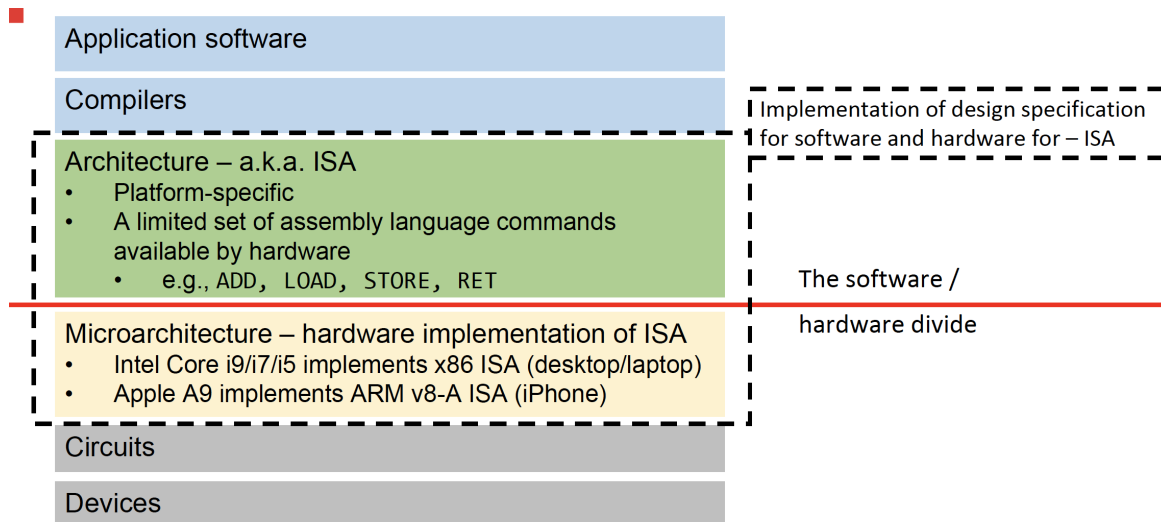
1.3 High Level Ideas

2 Instruction Set Architecture

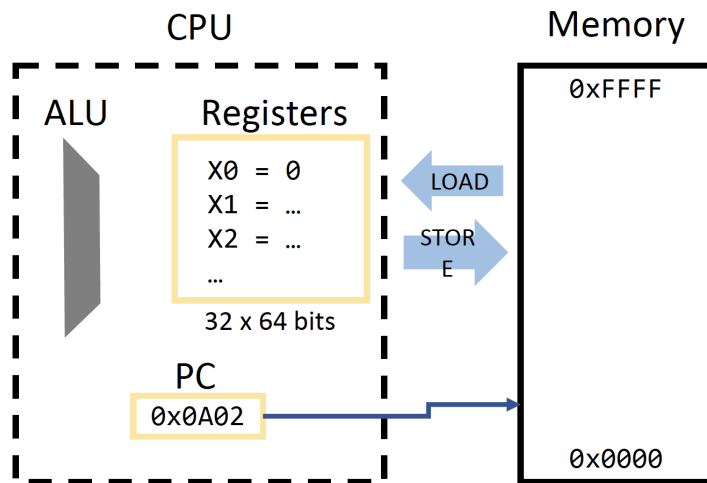
Instruction Set Architecture (ISA)

- An abstract interface between the hardware and the lowest-level software that encompasses all the information necessary to write a machine language program that will run correctly, including instructions, registers, memory accesses, I/O, and so on
- Includes anything programmers need to know to make a binary program work correctly
- Defines interface between hardware and software

its essentially a legal document that states the instructions, machine code that works on the hardware



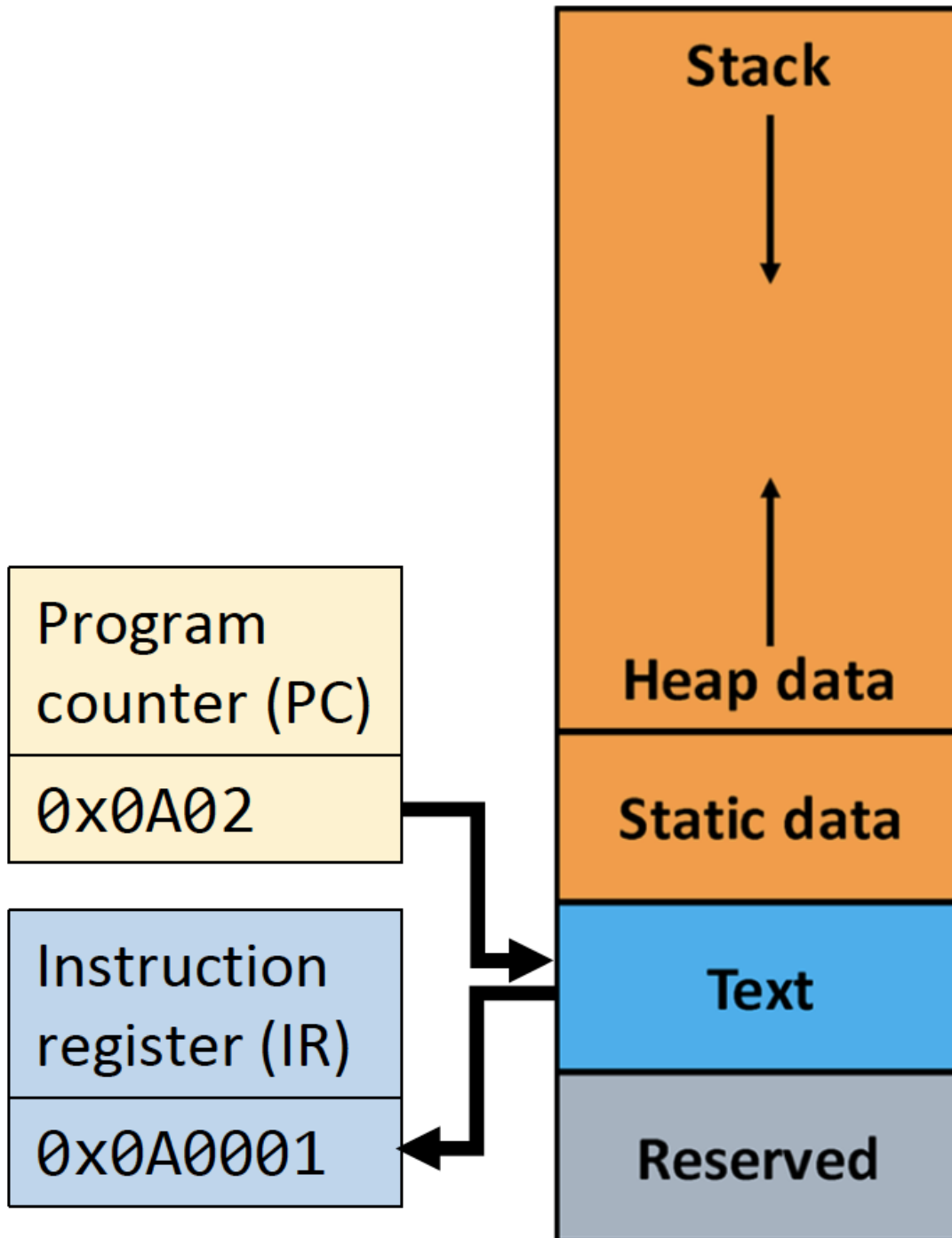
2.1 (Simplified) System Organization



CPU – Central Processing Unit

ALU – Arithmetic Logic Unit, executes instructions

PC – Program Counter, holds address (in memory) of next instruction



- Von Neumann Architecture

- Data and instructions are stored in the same memory
- Programs (instructions) can be viewed as data - simplifies storage
- Data can be viewed as instructions - complicates computer security

Stack - Grows down in memory Heap Data - dynamically allocated data, Grows up in memory
Text - Reserved

2.2 ISA Design Space

1. What instructions to include?
 1. add, mult, divide, branch, load/store, etc.
2. What storage locations?
 1. how many registers, how much memory, other “architected” storage?
3. How should instructions be formatted?
 1. 0, 1, 2, or more operands? Immediate operands?
4. How to encode instructions?
 1. CISC vs RISC
5. What instructions can access memory?
 - RISC (Reduced Instruction Set Computer)
 - All instructions are same length (eg ARM, LC2K) smaller set of simpler instructions
 - the microarchitecture can be simple so we can pack more into them. Is simple, compact
 - CISC (Complex Instruction Set Computer)
 - Instructions can vary in size (Digital Computer’s VAX, x86) large set of simple and complex instructions
 - ARM: only loads and stores can access memory (called a “**load-store architecture**”)
 - Intel x86 is a “register-memory architecture”, that is, other instructions beyond load/store can access memory

3 Assembly

3.1 Basics

3.1.1 Instruction Encoding

3.1.2 Properties

3.2 Special Purpose Registers

Return Address:

- Example: ARM register X30, aka Link Register (LR)
- Holds the return address or link address of a subroutine

Stack Pointer

- Example: ARM register X28-SP, or x86 ESP
- Holds the memory address of the stack
 - Points to the bottom of the stack

Frame Pointer

- Example: ARM register X29-FP
- Holds the memory address of the start of the stack frame

Program Counter (PC)

- Cannot be accessed directly in most architectures
- Accessed indirectly through jump insts

The above registers store memory addresses

0 Value Register

- ARM register X31-XZR
- no storage, reading always return 0
- lots of uses - ex: mov->add

- Its a source of 0s
 - We can throw a value away by writing to it
 - We can get a 0 by reading it
 - We can use fewer opcodes if we have a 0 value register
 - * Eg you can make a move instruction by adding 0

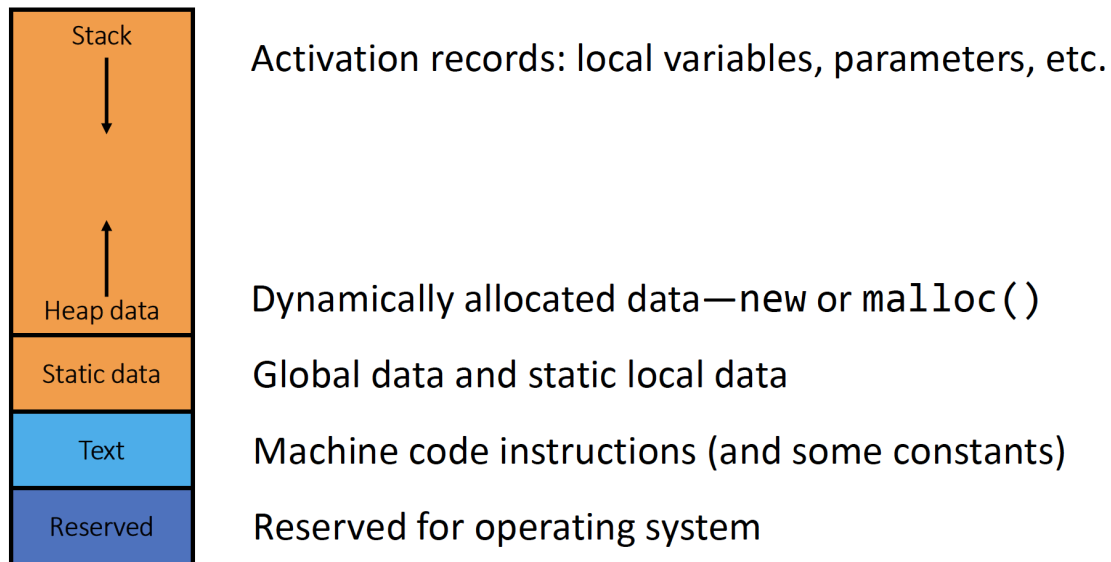
Status Register

- Example: ARM SPSR, or x86 EFLAGS
- Status bit set by various instructions
 - Compare, add (overflow and carry), etc
- Used by other instructions like conditional branches

3.3 Memory

Machine with n -bit address can reference memory locations $[0, 2^n - 1]$

Assembly instructions have multiple ways to access memory (ie addressing)



Addressing Modes

- Direct addressing - mem address is in the instruction
- Register indirect - mem addr is stored in a reg
- Base + Displacement - reg. indirect + an imm. value

- PC-relative - base + displacement using the PC (essentially PC + imm. value)

A **word** is a collection of bytes

- Exact size of a word depends on the architecture
- In ARM a word is 4 bytes

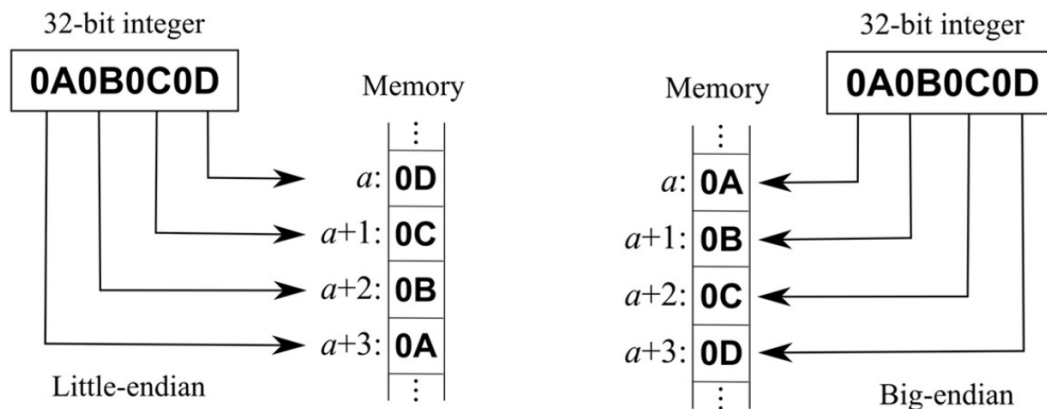
ARM and most modern ISAs are **byte addressable**

- I.e. each addr. refers to a particular byte in memory

3.4 Big Endian vs Little Endian

Endian-ness: ordering of bytes within a word

- Little - increasing numeric significance with increasing memory addresses
- Big - opposite, most significant byte first
- Internet is big indian, x86 is little endian, ARM can switch
- No advantage to either one, just arbitrary



How to tell the endian-ness of a system?

- Method 1: Using simple C program to check byte order:

```

unsigned int x = 1;
char *c = (char*)&x;

if (*c)
    printf("Little Endian\n");
else
    printf("Big Endian\n");

```

- The integer x is assigned 1, which in memory is represented as 0x00000001.
- We then take a pointer to the first byte (`char* c = (char*)&x`).
- If the first byte (`*c`) is 1, the system is Little Endian (least significant byte stored first).
- If the first byte is 0, the system is Big Endian (most significant byte stored first).

3.5 Memory Layout

3.5.1 Memory Layout of Variable

- Most modern ISAs require that data be **aligned**
 - **alignment:** An N-byte variable must start at an address A, such that $A \% N = 0$
 - * i.e. address must be divisible by the size of the variables
- **“Golden” Rule:** Address of a variable is aligned based on the size of the variable
 - char is byte aligned (any address is fine)
 - * 1 byte, anything mod 1 is fine
 - short is half-word (H) aligned (LSB of address must be 0)
 - int is word (W) aligned (2 LSBs of address must be 0)
- Why? This simplifies hardware needs for loads and stores
 - Otherwise multiple memory accesses are needed to access one variable

3.5.2 Structure Alignment

- Each field in a struct is laid out in the order it is declared using the Golden Rule for alignment
- Identify largest (primitive) field
 - Starting address of overall struct is aligned based on the largest field (any primitive variable)
 - * Primitives are mostly 1,2,4,8 bytes


- Size of overall struct is a multiple of the largest fields
- Why? So that we could have an array of structs
 - * Guarantees that each instance of struct is aligned the same way
- Note: Notice how arrays are itself a struct, thus arrays are aligned on its data type rather than the overall size of the array

Why care about data layout?

- Compilers don't reorder variables in memory to avoid padding
- Programmer is expected to manage data layout for your program and structs
- Tip: Powers of 2s for arrays
 - Can use shifting to access faster

3.5.2.1 Examples

Pointers are 8 bytes in ARM and x86



Datatype	size (bytes)
short	2
char	1
int	4
double	8
address	8

Problem: What boundary should this struct be aligned to?
What is the total size of the struct?

```
C-code
struct {
    char w;
    int x[3];
    char y;
    short z;
} s;
```

Largest field is 4 bytes (int), therefore:

- struct size will be multiple of 4
- struct starting address is word aligned, since a word is 4 bytes

Assume struct starts at address 1000, what is the data layout of the struct?

```
char w -> 1000
// padding 1001-1003
x[0] -> 1004-1007
x[1] -> 1008-1011
x[2] -> 1012-1015
char y -> 1016
// padding 1017
short z -> 1018-1019
```

start: 1000

end: 1019

Total size = 20 bytes

Datatype	size (bytes)
short	2
char	1
int	4
double	8

Why padding?
"Golden" rule –
Address of a variable is aligned based on the size of the variable

Problem: Calculate the total amount of memory needed for the declarations.
Assume data memory starts at address 100

C-code	C-code	Bytes	Start	End	Notes
<pre>short a[100]; char b; int c; double d; short e; struct { char f; int g[1]; char h; } x;</pre>	short a[100];	200	100	299	
	char b;	1	300	300	
		3	301	303	padding
	int c;	4	304	307	
		4	308	311	padding
	double d;	8	312	319	
	short e;	2	320	321	
		2	322	323	padding
	struct {	12	324	335	largest field: 4 bytes
	char f;	1	324	324	
		3	325	327	padding
	int g[1];	4	328	331	
	char h;	1	332	332	
		3	333	335	padding
	} x;	12	324	335	

Datatype	size (bytes)
short	2
char	1
int	4
double	8

An N-byte variable must start at an address A, such that
 $(A \% N) == 0$

Total size: 236 bytes

19

Problem: Calculate the total amount of memory needed for the declarations.
Assume data memory starts at address **200**

C-code	C-code	Bytes	Start	End	Notes
<pre> int a; struct { double b; char c; int d; } x; char *f; short g[20]; </pre>	int a;	4	200	203	
		4	204	207	padding
	struct {	16	208	223	largest field: 8 bytes
	double b;	8	208	215	
	char c;	1	216	216	
		3	217	219	padding
	int d;	4	220	223	
	} x;	16	208	223	
	char *f;	8	224	231	
	short g[20];	40	232	271	
	TOTAL:	72	200	271	

3.6 Control Flow

3.7 C to Assembly

3.8 Functions

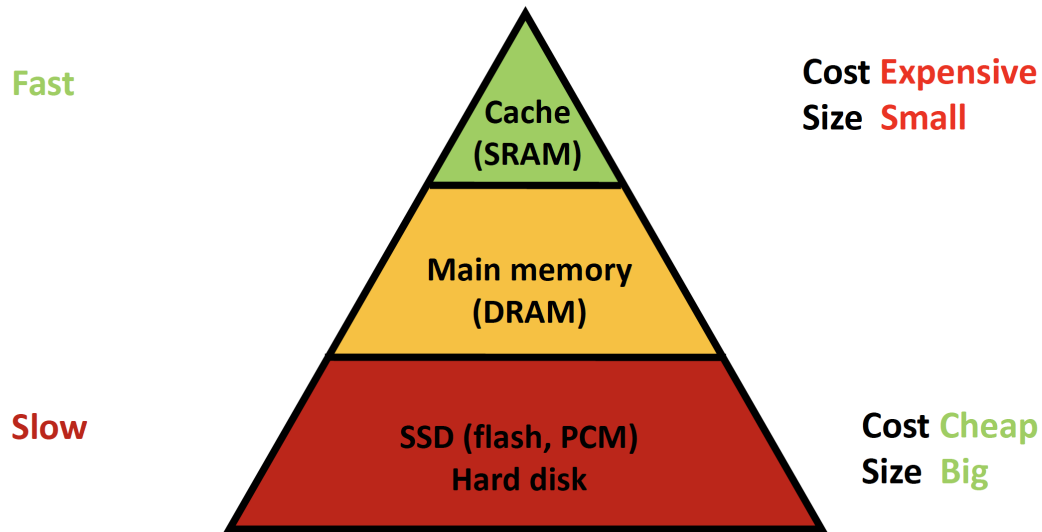
3.8.1 Call and Return

4

5

6

7 Memory Hierarchy & Caching



SRAM

Area: 6T per bit (used on-chip within processor) Speed: Typical Size: Cost: Volatile ()

DRAM

Disks

Flash

Cache

Few KBs to MBs of SRAM (within processor – on-chip cache)

Fast

Small, so cheap

Serves most loads and stores, provided program has good locality

Main Memory

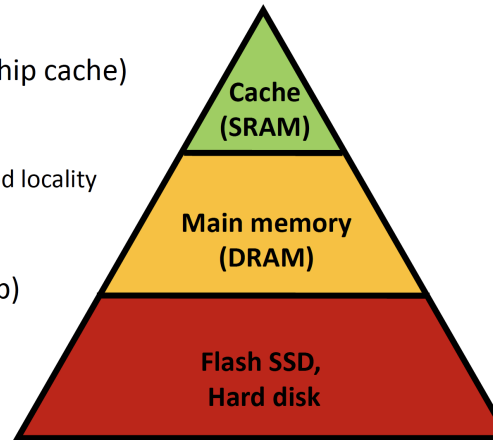
Tens of GBs of DRAM (outside processor – off-chip)

Cheaper than SRAM, faster than flash/disk

“Swap space”

Few TBs OF flash and/or disk

Cheap, Big, Non-volatile.



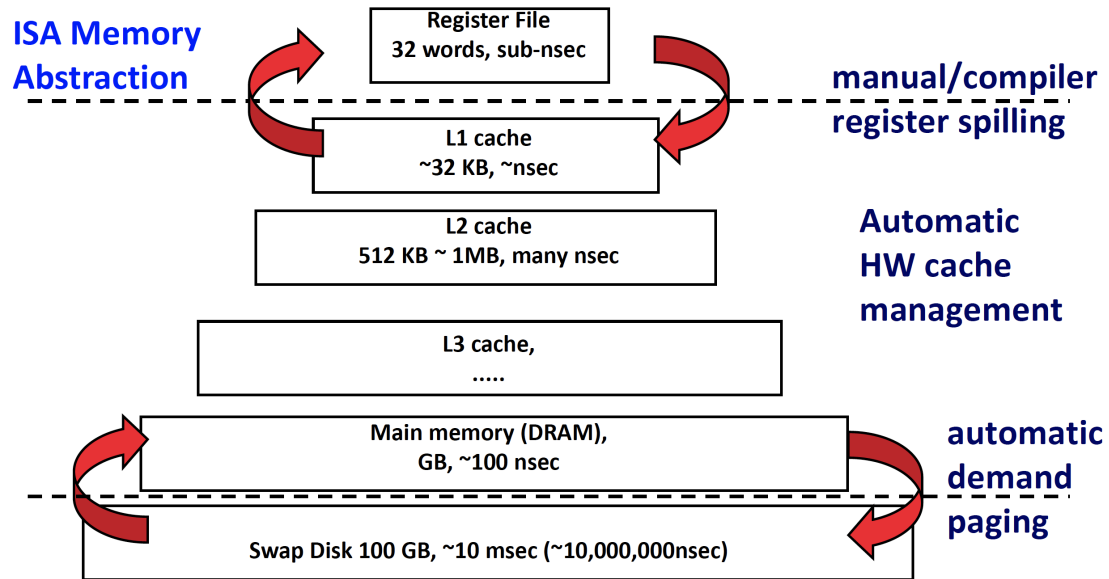
Virtual Memory provides an illusion that an ISA’s entire address space is available (since cost of having all addresses be available in a 64 bit system is too high)

7.1 Cache Basics

Cache commonly refers to SRAM used on-chip within the processor. Used to store data that is **most likely** to be referenced by a program

Key cache performance metric: AMAT - Average Memory Access Time

Cache consumes most of a CPU’s die area



7.1.1 Basic Cache Design

addr	data
addr	data

TAG BLOCK

- **TAG** (CAM) holds the memory address
- **BLOCK** (SRAM) holds the memory data

Accessing the cache: compare reference address and tag

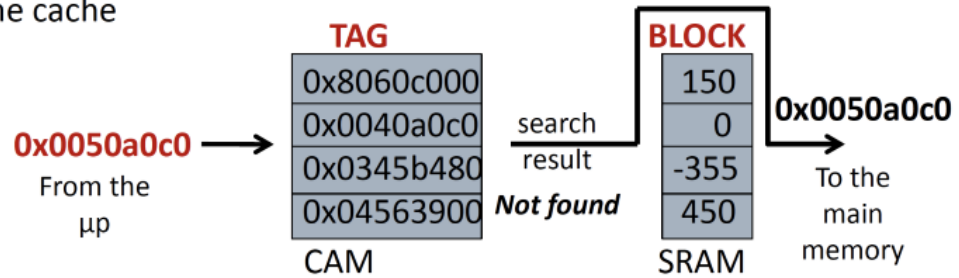
- Match? Get data from cache block -> Cache HIT
- No match? Get data from main memory -> Cache MISS
- This is implemented with CAMs to store tags

A cache consists of multiple tag/block pairs, each pair is a **cache line**

A **hit** in the cache



A **miss** in the cache

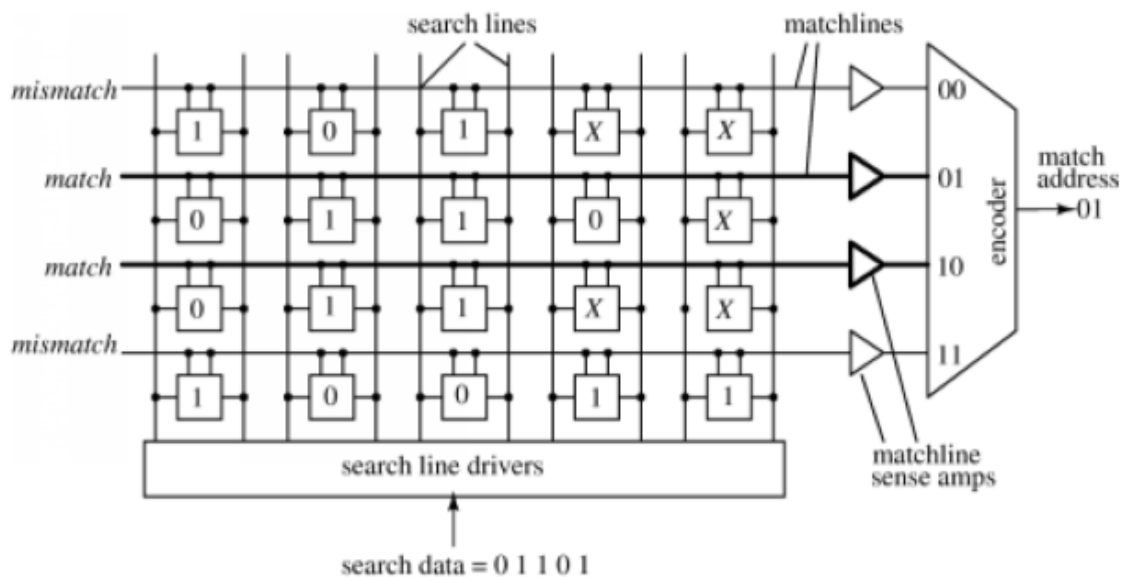


Cache Block refers to the data block

Cache Line refers to the entire line (including valid, tag array, data array)

7.1.1.1 Operations of a CAM

- ❑ **Search:** the primary way to access a CAM
 - Send data to CAM memory
 - Return “found” or “not found”; “hit” or “miss”
 - If found, return location of where it was found or associated value
- ❑ **Write:**
 - Send data for CAM to remember
 - Where should it be stored if CAM is full?
 - Replacement policy
 - Replace oldest data in the CAM
 - Replace least recently searched data



7.2 Cache Operation

On a cache miss, we fetch data from main memory and allocate a cache line and store the fetched data in it

Different policies exist to pick the victim of data replacement

We mainly exploit two localities:

1. Temporal Locality

1. More recently accessed memory locations are more likely to be re-accessed than other locations
2. Data in least recently referenced (or Least Recently Used **LRU**) cache line should be evicted to make room

2. Spatial Locality

1. Memory addresses close to gether are more likely to be accessed than far apart addresses

7.2.1 AMAT

$AMAT = \text{cache latency} \times \text{hit rate} + \text{memory latency} \times \text{miss rate}$

- Every memory access goes through the cache
 - Cache misses go through main memory
 - *this eqn doesn't include disk latency, thus changes if we include disk

To improve AMAT:

1. Reduce latency of cache/main memory/disk AND/OR
2. Increase hit rate of cache and main memory

7.2.2 Overheads

Cost of tags often drives the design of real caches, since these functionalities use significant area

Overheads: tag bits, valid bits, dirty bits, etc per cache line

7.3 Tracking LRU

Naive Method:

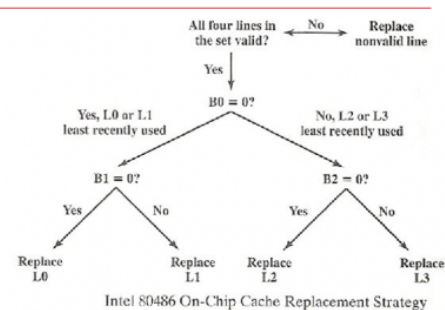
- Maintain LRU_rank per cache line
- Set LRU_rank of newly accessed block to 0, increment all others by 1
- Line with highest LRU_rank is victim on eviction
- Area overhead: $\log(\# \text{ cache lines})$ per cache line
- However this is extremely slow and inefficient for hardware since it requires linearly modifying every cache line

LRU with least area overhead:

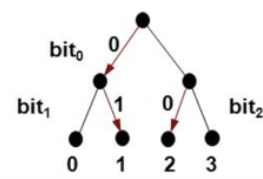
- We know that the number of permutations for N cache lines is $n!$
 - Thus, we need $\log(n!)$ bits (in total) for N cache lines

7.3.1 Pseudo LRU

- ❖ Partial LRU order to reduce complexity
- ❖ Need $N - 1$ bits to maintain LRU order of N cache lines
 - ❖ Rank based algorithm from last lecture needs $N * \log N$ bits
- ❖ Find LRU for replacement \rightarrow Decode the $\log N$ bits as a binary tree.
 - ❖ Each bit represents one branch point in a binary decision tree
 - ❖ Bit is "0" means go right to find LRU
 - ❖ Bit is "1" means go left to find LRU
- ❖ Update LRU rank on access
 - ❖ Flip bits along the path to the cache line accessed



Example



MRU: line 1, LRU: line 3

24

7.4 Cache Blocks

To reduce overhead of each cache line, we could also increase cache block size

Memory Address / Block Size = Tag (which is essentially a Block Address)

Block Offset = Address % Block Size

Block Size = 2^{block_offset}

sizeof(block_offset) = log2(block_size)

Tag size = address_size - block_offset_size

However, larger block size isn't always better due to (slower access since larger data needs to be fetched, and wasteful fetches exceeding spatial locality)

7.4.1 Row vs Column Major

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

could be stored in two possible ways:

Row major is
a common choice

Address	Row-major order	Column-major order
0	a_{11}	a_{11}
1	a_{12}	a_{21}
2	a_{13}	a_{12}
3	a_{21}	a_{22}
4	a_{22}	a_{13}
5	a_{23}	a_{23}

Row major is more common. Column major used for more GPU and HPC tasks. Row/Col major is important due to spatial locality in memory going through 2D/3D arrays.

7.5 Stores: Write-Through vs Write-Back

Where should you write the result of a store to an address X?

If address X is in the cache

Write to the cache.

Should we also write to memory?

(yes - **write-through policy**)

(no - **write-back policy** – write only when modified cache block is evicted)

If address X is not in the cache

Allocate address in the cache?

yes - **allocate-on-write policy**

no - **directly write to memory, no allocate-on-write policy**

Cache does not write all stores to memory immediately - Keep track of most recent updated copy and only update main memory when data is evicted from the cache - Keep a **dirty bit** that tracks whether a cache line has been modified and needs to writeback to memory - Such that we don't need to write back other unmodified lines to memory on eviction

Each cache miss reads a block (2 bytes in our example) from main memory

Total reads from main memory: 8 bytes

Each eviction of a dirty line writes a whole block (2 bytes) to main memory

Total writes to main memory: 4 bytes

(1 write-back during simulation, 1 write-back at the end of simulation)

	Memory Reads	Memory Writes
Write Through	Block size * misses	Store size * Num of stores
Write Back	Block size * misses	Block size * Num of dirty block replacements

Write-allocate vs. no-write-allocate caches

Policy that decides what to do with a cache-miss on a store instruction.

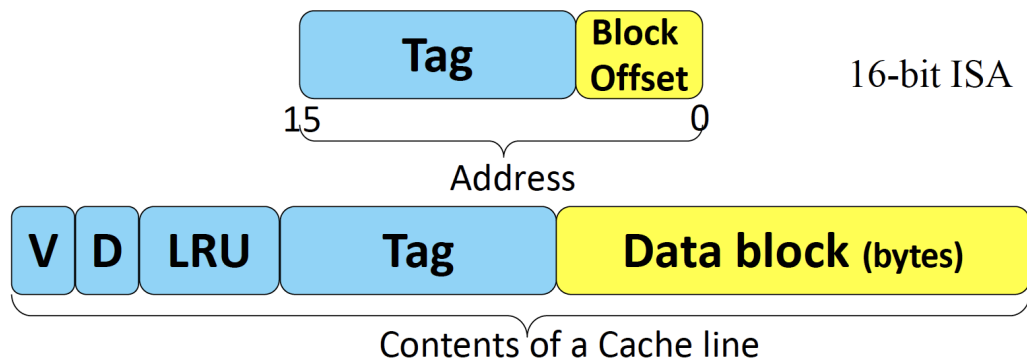
Write-allocate: First bring data from memory into the cache, then write

No-write-allocate: do not bring data in the cache, just write directly to the memory, not to the cache

Store w No Allocate	Write-Back	Write-Through
Hit?	Write Cache	Write to Cache + Memory
Miss?	Write to Memory	Write to Memory
Replace block?	If evicted block is dirty, write to Memory	Do Nothing

Store w Allocate	Write-Back	Write-Through
Hit?	Write Cache	Write to Cache + Memory
Miss?	Read from Memory to Cache, Allocate to LRU block Write to Cache	Read from Memory to Cache, Allocate to LRU block Write to Cache + Memory
Replace block?	If evicted block is dirty, write to Memory	Do Nothing

7.6 Cache Mapping



Cache blocks:

- Captures spatial locality (increase cache hit rate)
- Reduces tag overhead (number and size of tags)

Need not store block offset in the cache line

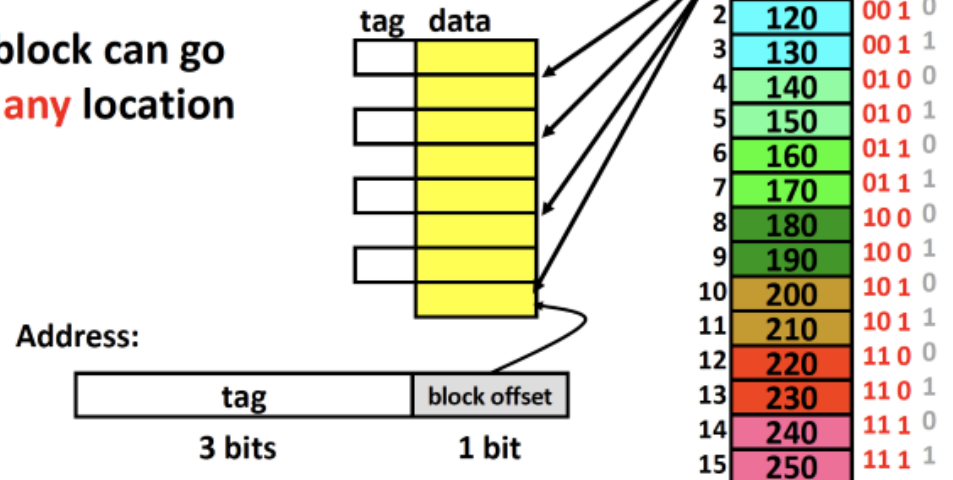
Determine byte to be read/written from the address directly

7.6.1 Fully-Associative Cache

- A memory location can go to any free cache line
- Check the tag of every cache line to determine match
 - Slow due to parallel tag searches over entire cache

Fully-associative caches

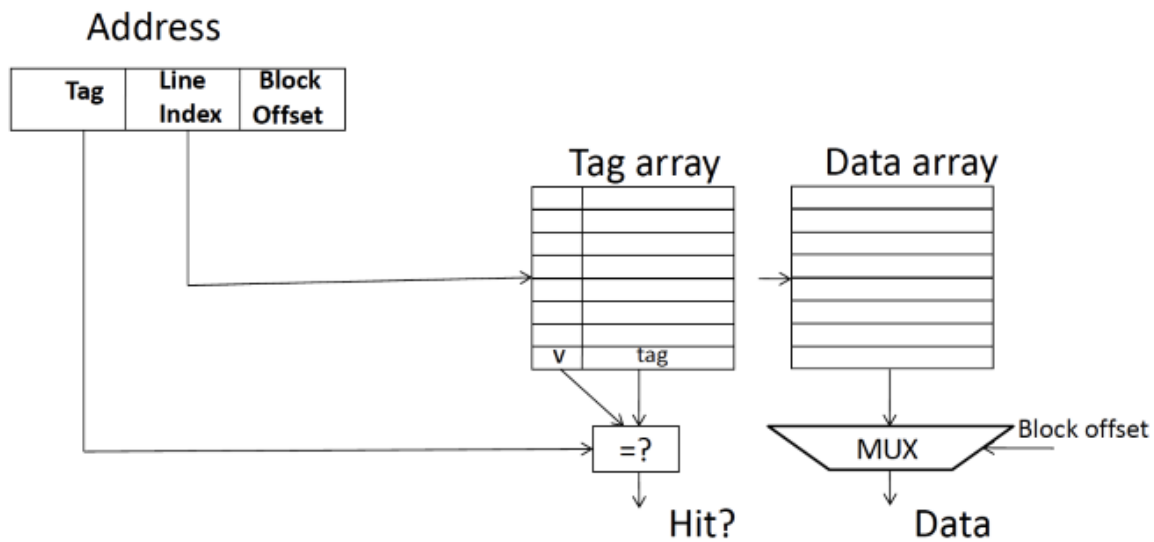
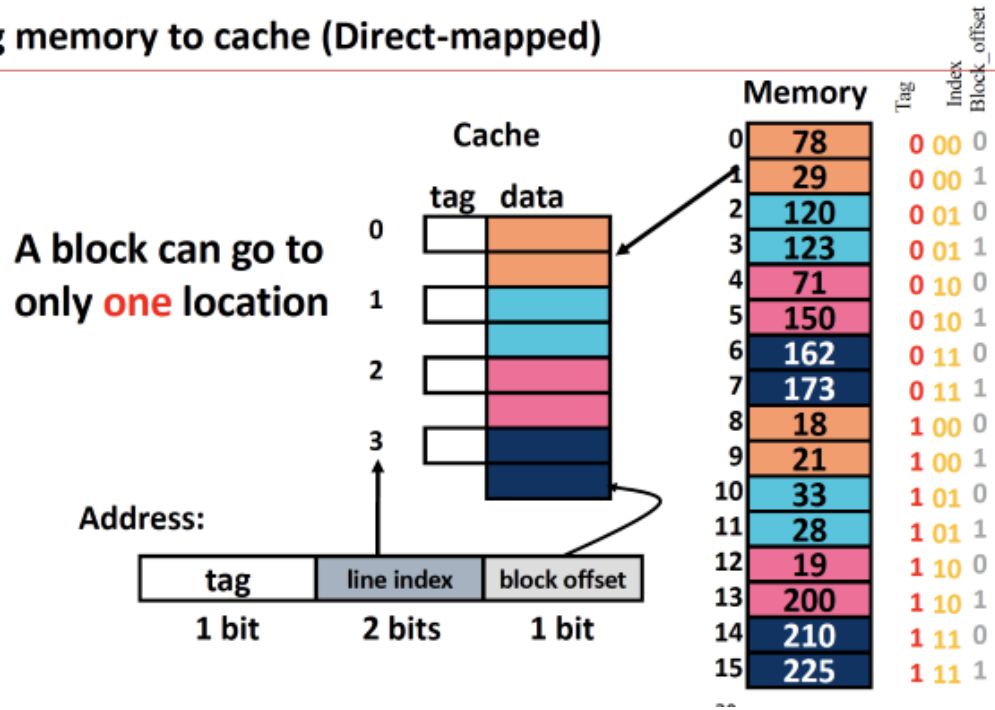
A block can go to **any** location



7.6.2 Direct Mapped Cache

- Directly map blocks of memory to one single location it can be held in
- Partition memory into as many regions as there are cache lines
- Each memory region maps to a single cache line in which data can be placed
- Eliminates requirement for parallel tag lookups
 - Only need to check a single tag - the one associated with the region the reference is located in
- Associativity is 1

Mapping memory to cache (Direct-mapped)

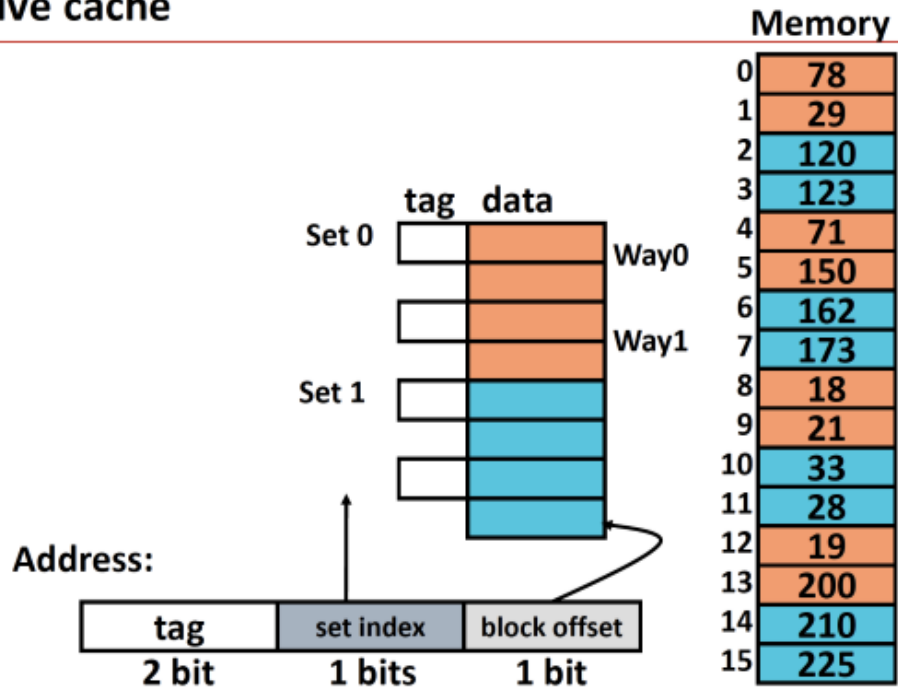


7.6.3 Set-Associative Cache

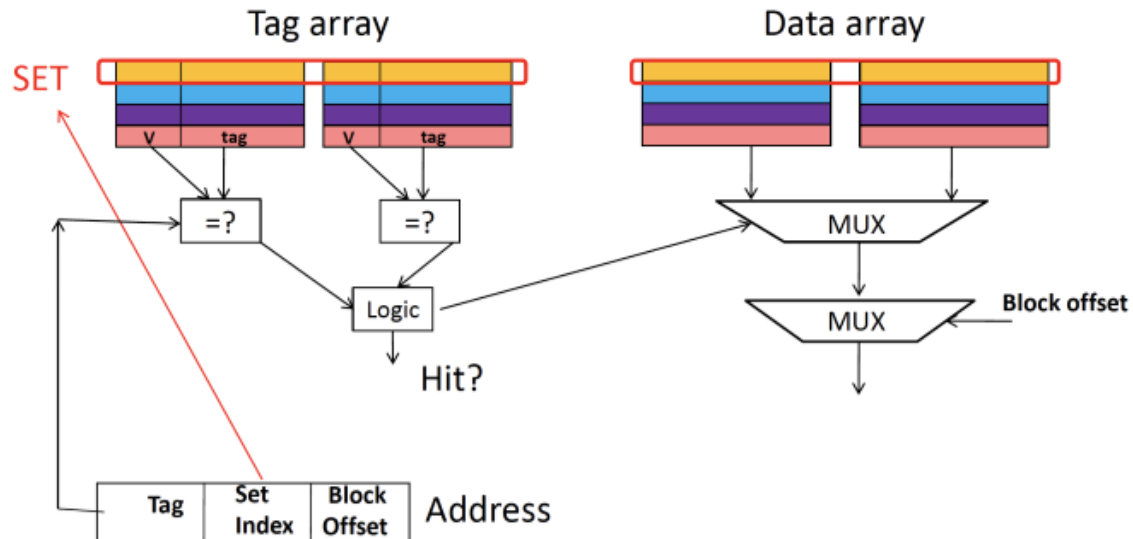
- Partition memory into regions but with fewer partitions than direct mapped
- Associate a region to a set of cache lines

- Check tags for all lines in a set to determine a HIT
- Treat each line in a set like a small fully associative cache
 - LRU (or LRU-like policy generally used)

Set-associative cache



- **Ways** are the blocks within a set
- Each cache line within a set is a **way**
 - E.g. if a set is 2-way associative, it has 2 cache lines within a set
 - * i.e. it has to perform 2 tag comparisons per access
- **Sets** are essentially the number of partitions you split your entire cache into smaller fully-associative “sub” caches



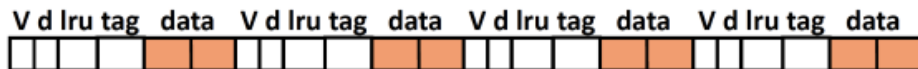
7.6.4 Cache Associativity Summary

Cache Organization Comparison

Cache size = 8 bytes (for all caches)
 Block size = 2 bytes
 #blocks = 4

Fully associative

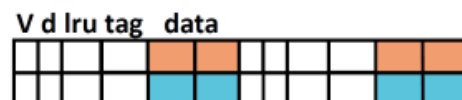
blocks per set = all blocks = 4 in this example;
 so, also correct to view this cache as 4-way associative



Direct mapped: (#blocks per set = 1)



2-way associative (#blocks per set = 2)



Block

$$\#blocks = \text{cache size} / \text{block_size}$$

$$\#cache\ lines = \#blocks$$

$$\text{block_offset_size} = \log_2(\#block_size)$$

Set

$$\#sets = \#lines / \#ways = \#lines / (\text{lines per set})$$

Direct-mapped: $\#sets = \#lines / 1$

2-way associative: $\#sets = \#lines / 2$

n-way associative: $\#sets = \#lines / n$

fully-associative: $\#sets = 1$ (all lines are in 1 set)

$$\text{set_index_size} = \log_2(\#sets)$$

$$\text{Tag size} = \text{address size} - \text{set_index_size} - \text{block_offset_size}$$

7.7 Cache Misses

Compulsory miss

First reference to any block will always miss

Also sometimes called a “**cold start**” miss

Capacity miss

Cache is too small to hold all the data

Would have had a hit with an infinite cache

Conflict miss

Would have had a hit with a fully associative cache

Can we classify a cache miss into one of the following?

Compulsory miss

Capacity miss

Conflict miss

Yes! Simulate three different caches

Simulate with a cache of unlimited size (cache size = memory size)

– Any misses must be **compulsory misses**

Simulate again with a fully associative cache of the intended size

- Any new misses must be **capacity misses**

Simulate a third time, with the actual intended cache

- Any new misses must be **conflict misses**

Compulsory misses

First reference to an address

No way to completely avoid these

Reduce by **increasing block size (spatial locality)**

This reduces the total number of blocks

Capacity misses

Would have a hit with a large enough cache

Reduce by **building a bigger cache**

Conflict misses

Would have had a hit with a fully associative cache

Cache does not have enough associativity

Reduce by **increasing associativity**

7.7.1 Cache Parameters

Key parameters affecting miss rate:

- Cache Size
- Block Size
- Associativity
- Replacement Policy

7.7.1.1 Cache Size

Cache size in the total data (not including tag) capacity

bigger can exploit temporal locality better

not ALWAYS better

Too large a cache adversely affects hit & miss latency

smaller is faster => bigger is slower

access time may degrade critical path

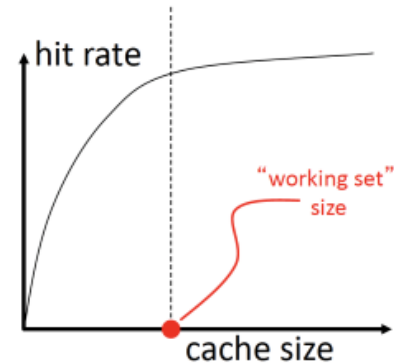
Too small a cache

doesn't exploit temporal locality well

useful data replaced often

Working set: the whole set of data
executing application references

Within a time interval



7.7.1.2 Block / Line Size

Block size is the data that is associated with an address tag

Sub-blocking: A block divided into multiple pieces (each with V bit)

Can improve “write” performance

Too small blocks

don't exploit spatial locality well

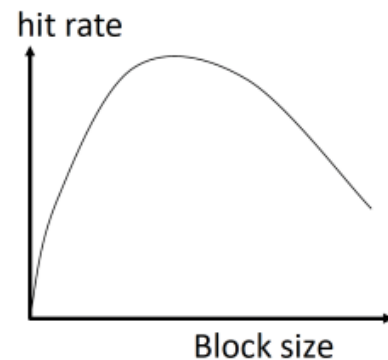
have larger tag overhead

Too large blocks

too few total # of blocks

likely-useless data transferred

Extra bandwidth/energy consumed



7.7.1.3 Associativity

How many blocks map to the same set (same set index)?

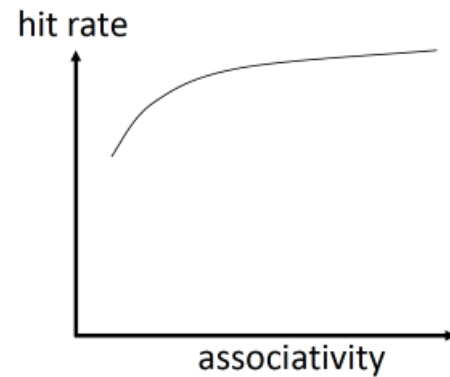
Larger associativity

- lower miss rate, less variation among programs
- diminishing returns

Smaller associativity

- lower cost
- faster hit time
- Especially important for L1 caches

Power of 2 associativity?



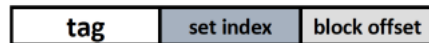
7.8 Cache Summary

Programmer's View

Address size: defined by ISA

e.g., 64-bit ISA has 64-bit addresses

Cache design defines how address is split into the following (hidden from the programmers/compiler):



Byte vs Word Addressable

Byte addressable ISA: each address refers to a **byte** in memory

Word addressable ISA: each address refers to a **word** in memory.
(cannot address individual bytes in a word)

Common practice:

In 64-bit ISA, a "word" is 64-bits (8 bytes)

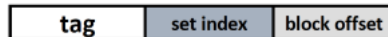
In 32-bit ISA, a "word" is 32-bits (4 bytes)

This difference impacts how we determine block offset size:

Example: Block-offset for a 256-byte block:

In 64-bit **byte** addressable ISA: a 256-byte block has **256 bytes**. So block_offset_size is 8 bits

In 64-bit **word** addressable ISA: a 256-byte block has **32 words**. So block_offset_size is 5 bits



7

Cache Organization: Equations

Block

$$\#blocks = \text{cache size} / \text{block_size}$$

$$\#cache\ lines = \#blocks$$

$$\text{block_offset_size} = \log_2(\#blocksize)$$

Set

$$\#sets = \#lines / \#ways = \#lines / (\text{lines per set})$$

Direct-mapped: $\#sets = \#lines / 1$

2-way associative: $\#sets = \#lines / 2$

n-way associative: $\#sets = \#lines / n$

fully-associative: $\#sets = 1$ (all lines are in 1 set)

$$\text{set_index_size} = \log_2(\#sets)$$

$$\text{Tag size} = \text{address size} - \text{set_index_size} - \text{block_offset_size}$$

Calculating Average Memory Access Time (AMAT)

If we **do not** assume memory latency includes time to cache access and determine hit/miss:

$$\text{AMAT} = \text{cache latency} + \text{memory latency} \times \text{miss rate}$$

If we assume memory latency includes time to cache access and determine hit/miss:

$$\text{AMAT} = \text{cache latency} \times \text{hit rate} + \text{memory latency} \times \text{miss rate}$$

3Cs Problem

Simulate three different caches:

Infinite cache	(same block size)
Fully associative cache	(same block size and cache size)
Given cache	

How to classify a miss in the given cache?

Is it also a miss in the infinite cache?

Yes, then **Compulsory miss**

No. Is it a miss in the fully associative cache?

Yes, then **capacity miss**

No, then **conflict miss**

Simulating Caches

Figure out fields in the address:

tag, set index, block-offset

Determine (tag, set-index) of given sequence of addresses.

Sequence of (tag, set-index) is sufficient to simulate a cache. Data is not necessary.

Keep track of LRU rank of cache lines within each cache set for set-associative caches. (replacement policies other than LRU exist)

7.9 Examples

Consider the following cache:

32-bit byte addressable ISA

Cache size : 64KB (kilo-bytes) = $64 * 8$ kilo-bits = 512 Kilo-bits

Cache block size : 64 Bytes

Write-allocate, write-back, *fully associative*

Recall: 1 kilobyte = 1024 bytes (NOT 1000 bytes!)

What is the cache area overhead (tags, valid, dirty, LRU)? Assume $\log(\text{\#blocks})$ bits for LRU.

Tag	= 32 (Address) – 6 (block offset)	= 26 bits
#blocks	= 64K / 64	= 1024
LRU bits	= $\log(\text{\#blocks})$	= 10 bits
Overhead per block	= 26(Tag) + 1(V) + 1(D) + 10 (LRU)	= 38 bits
Total overhead for cache	= 38 bits * #blocks = 38 * 1024	= 38912 bits

8 Virtual Memory

Problems when running concurrent programs:

- How do we divide memory between programs?
- How do we isolate programs and prevent them from overwriting/accessing each others memory?
- How do we handle the limited physical capacity of memory?

Virtual Memory is introduced to provide a layer of indirection, letting each program think they have access to the entire address space, but is an illusion maintained by both the OS and the hardware

- It is an **operating system functionality** that enables multiple concurrently running programs to share physical memory and swap disk space
 - TLB (Translation Look-Aside Buffer) is a special cache that helps implement this efficiently

Virtual addresses are used by loads and stores in a program

Physical addresses are used by the hardware to identify storage

Primary Capabilities provided:

- Capacity
 - An illusion of full address space capacity
- Security
 - Isolation
 - * Isolates programs from each other
 - Permissions
 - * Manage access permissions to data/code

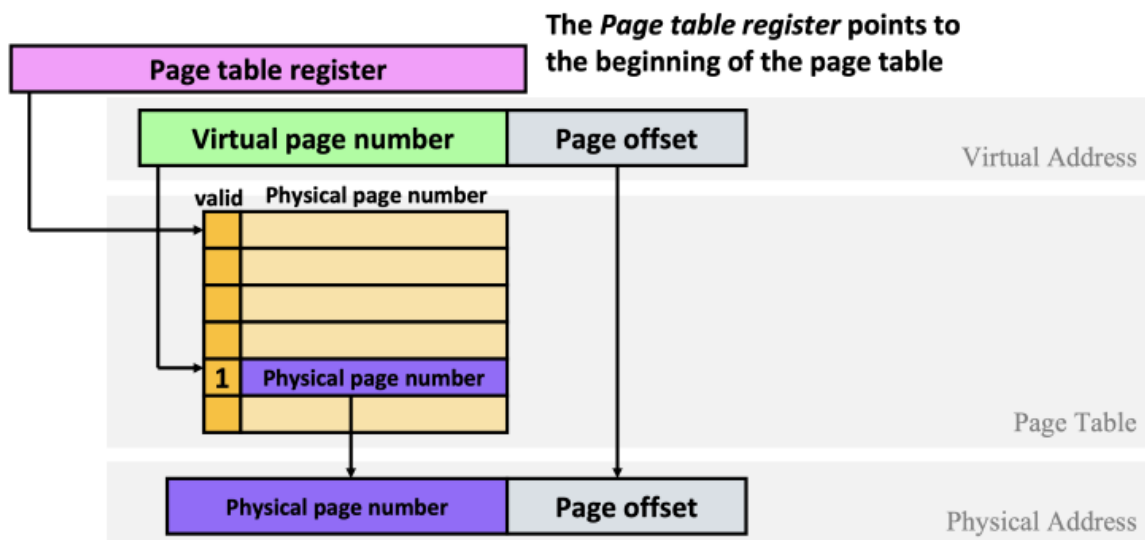
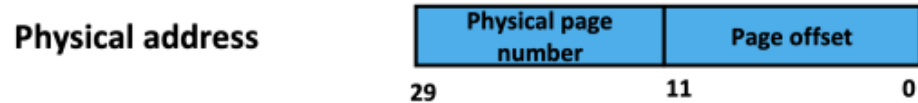
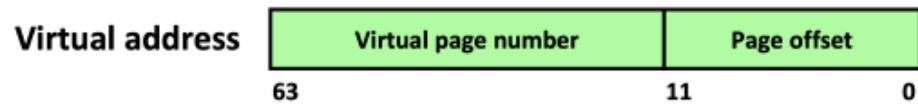
To efficiently create a mapping between virtual and physical addresses, we map entire chunks of memory addresses, called **pages**

Page Tables store this translation, and is managed by the OS and stored in memory.

- Size of a virtual page = size of physical page

A virtual address consists of

- A virtual page number
- A page offset (low order bits of the address)

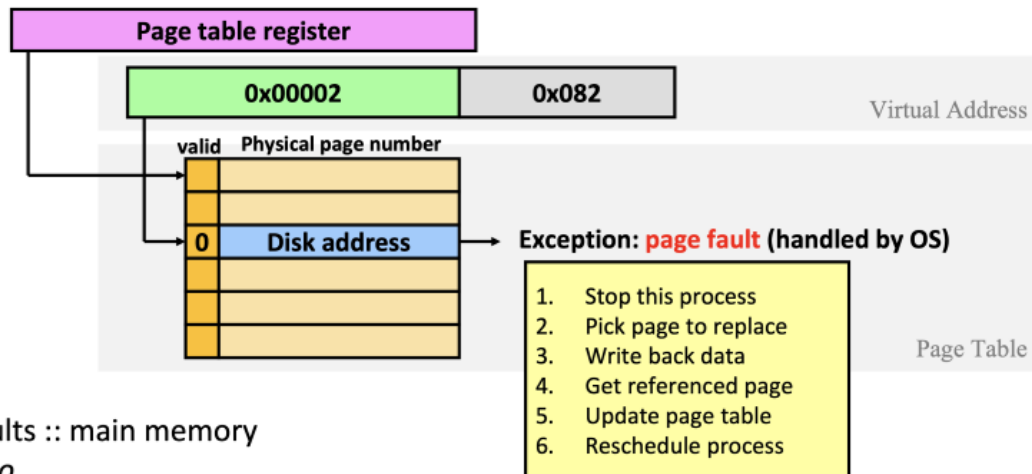


- Each process has its own **page table** to perform **address translation**
- The page table is maintained by the **operating system** and is stored in memory
 - OS knows the physical address of a program's page table
 - No address translation is required by the OS for accessing the page tables
- Address translation is needed for every load/store:
 - Would be expensive! Processor provides special support (TLB) to speed this up
 - A good example of hardware-software co-design!

Page Table (OS Data Structure)	
valid	Physical page number

8.0.1 Page Faults

When main memory is exhausted, the disk is used as “extra” space. This space is called **swap partition** (in linux based systems)



Page faults :: main memory
 similar to
 Cache miss :: processor caches

28

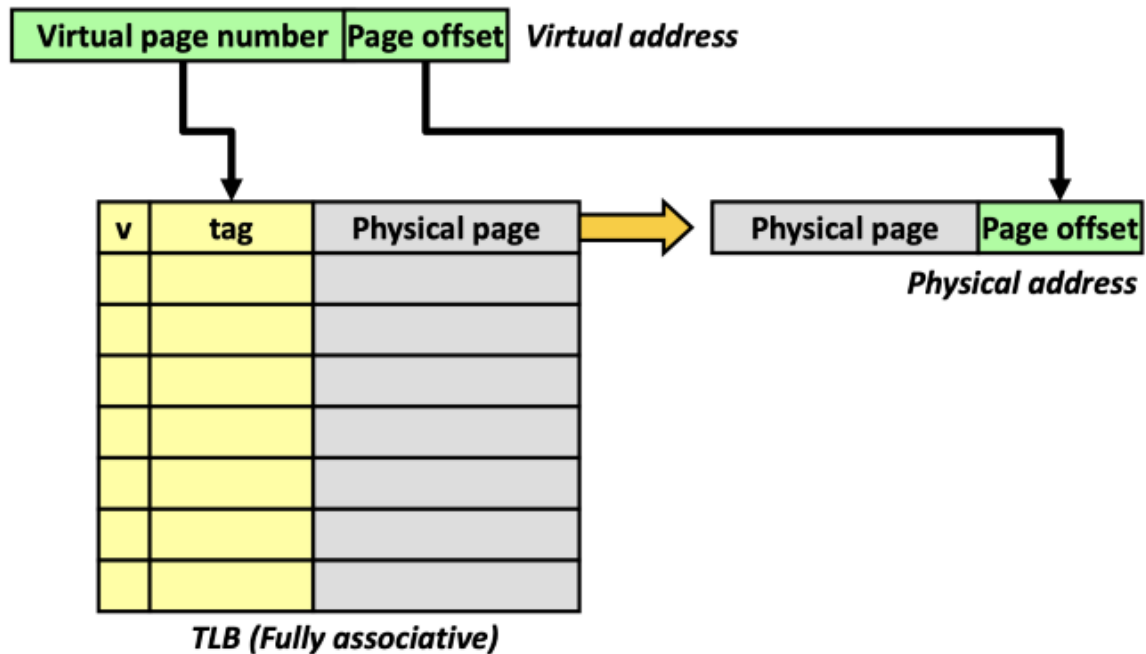
8.1 Hierarchical Page Tables

to be done...

8.2 Translation Look-Aside Buffer (TLB)

A special cache for page-tables to speed up address translation (reduce main memory access to page tables)

Commonly 16-512 entries and typically low miss rate (<1%)



8.3 Virtually-Addressed Caches

See more in Computer Architecture notes

9 Summary

In summary...

References