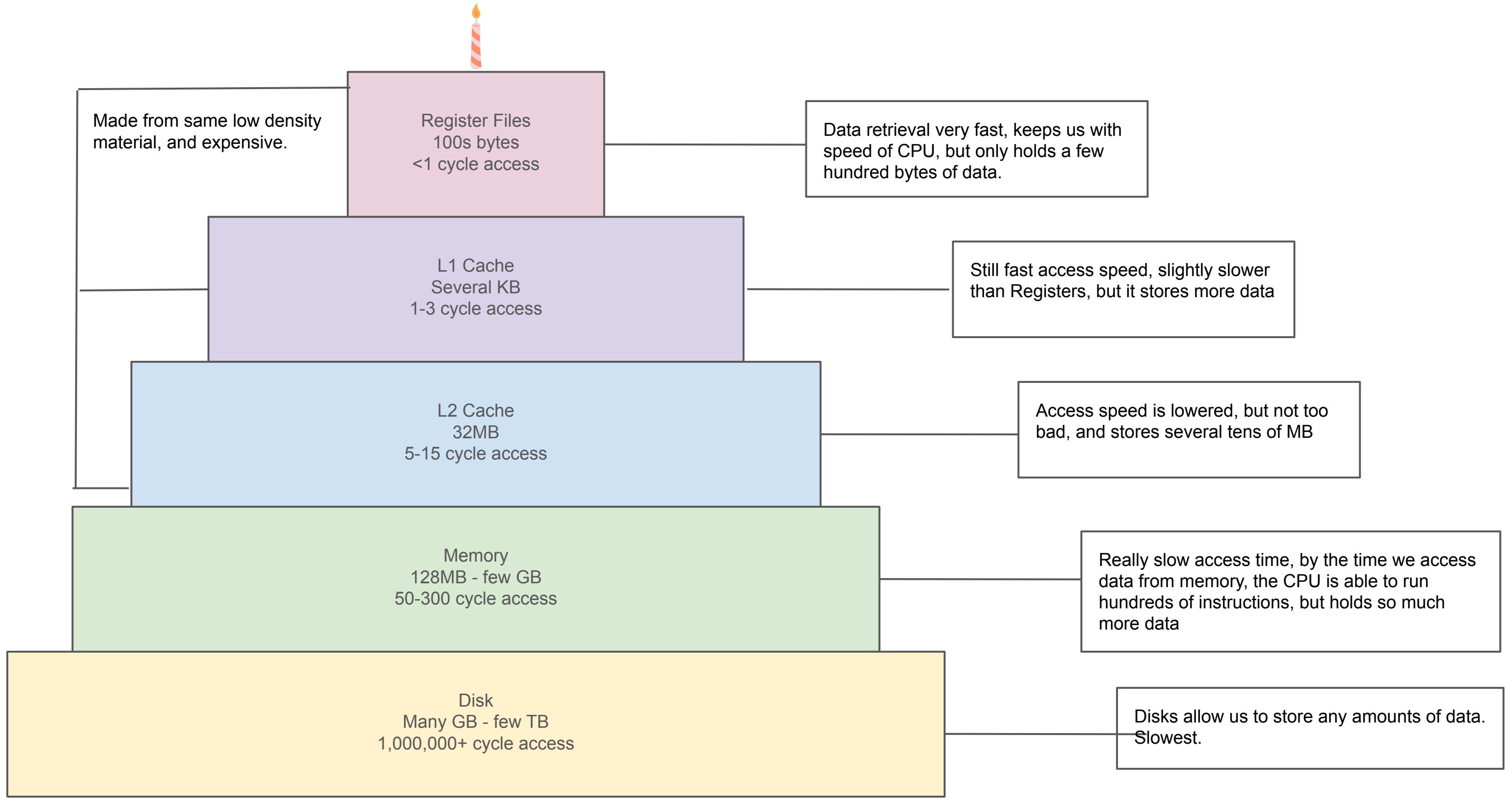

Made from same low density material, and expensive.
Register Files
100s bytes
<1 cycle access
Data retrieval very fast, keeps us with speed of CPU, but only holds a few hundred bytes of data.
L1 Cache
Several KB
1-3 cycle access
Still fast access speed, slightly slower than Registers, but it stores more data
L2 Cache
32MB
5-15 cycle access
Access speed is lowered, but not too bad, and stores several tens of MB
Memory
128MB - few GB
50-300 cycle access
Really slow access time, by the time we access data from memory, the CPU is able to run hundreds of instructions, but holds so much more data
Disk
Many GB - few TB
1,000,000+ cycle access
Disks allow us to store any amounts of data. Slowest.

The larger a memory device is, the speed of the clock needs to be reduced in order to ensure every memory cell receives the signal at the same time. Hence, speed on a larger device would be slower than the speed on a smaller device.

Temporal locality: A recently accessed memory has a higher probability that in the near future the same data will be accessed again.
Spatial locality: If we are accessing a data item, it is highly likely that the data items stored around the currently accessed item, will be accessed in the near future.

- So when loading list[1], we might also want to load list[2], list[3] … etc
- Hence, we will pre-fetch this data item list into memory so it's available in cache

CACHE LOOK-UP

In a cache with 4 slots, each slot holds one item. When accessing data items, insert them into an empty cache slot. If no slots are available, evict the data item in the slot where the data item has not been accessed for the longest time.

PROBLEM: Searching for a data item must be done in linear time complexity, hence it is not very efficient.

→ **SOLUTION**: Direct-Mapped Caches

DIRECT-MAPPED CACHES

Now an address can go to just one predefined slot in the cache using slot = address % CacheSize. So addresses that have a remainder of 1 go into slot 1, remainders of 2 into slot 2 etc.

PROBLEM 1: Searching for a data item requires a division operation, which consumes a lot of clock cycles

PROBLEM 2: Cache Thrashing: two items go in and out all the time even though there is plenty of space available.

→ **SOLUTION 1**: Replace division operation with a bitwise AND operation, provided the denominator is a power of 2.

→ **SOLUTION 2**: Fully-Associative Cache

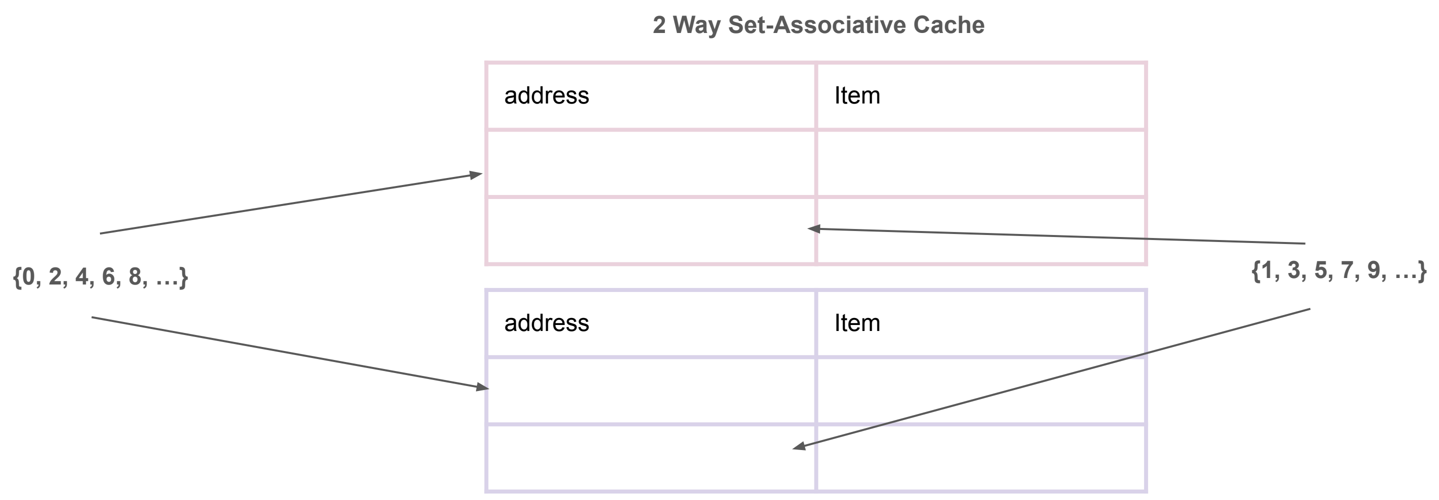
FULLY-ASSOCIATIVE CACHE

Fully associative mapping is that it solves the conflict miss problem, thereby increasing the hit rate. Instead of one cache box, we have two cache boxes, which is the original split into two smaller cache lines.

PROBLEM: Inefficient data lookup operations

→ **SOLUTION**: Set-Associative Cache

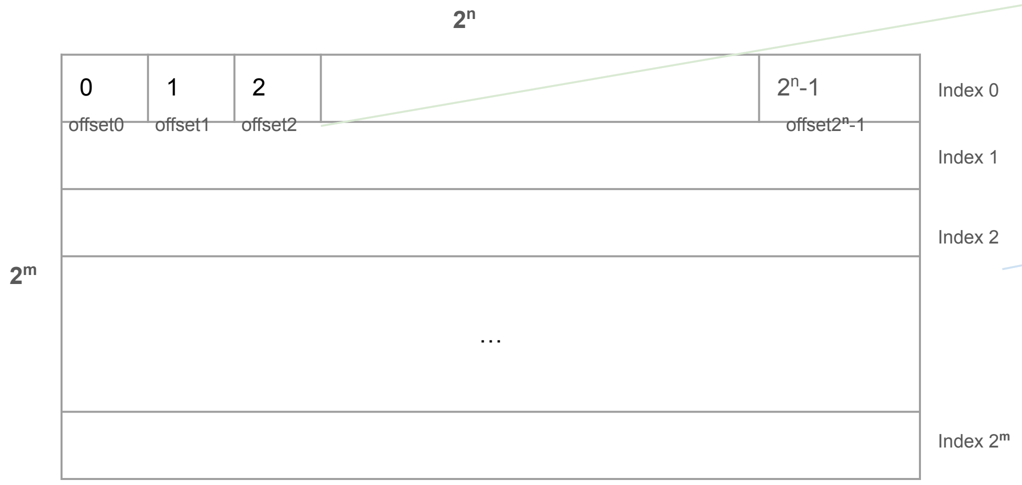
Set-Associative Cache



We can't fix all thrashing problem with a set-associative cache. For example, in a 2 way set-associative cache, the sequence {1,3,5,1,3,5} will cause a thrashing problem.

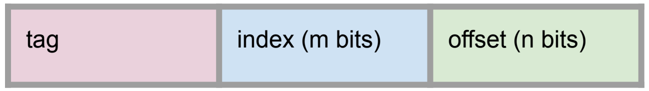
Cache Calculations

Cache is organised as a collection of lines.
Each line consists of one or several bytes.
A cache with 2 lines where each line has four words and each word has four bytes, how many bytes are there in the cache?
4 * 4 * 2 = 32
In general, if there are 2^m cache lines and each line has 2^n bytes, there are 2^{m+n} bytes in the cache.



The offset determines position in a cache line
OFFSET = log(bytes in a line)

The index determines the cache line
INDEX = log(bytes of cache size)



The tag field distinguishes different memory locations that are mapped to the same cache location.

The position of a byte in a cache line is given as an offset value. For a line with 2^n bytes, there are 2^n offset values.

Each cache line is given an index. For 2^m line, there are 2^m index values.

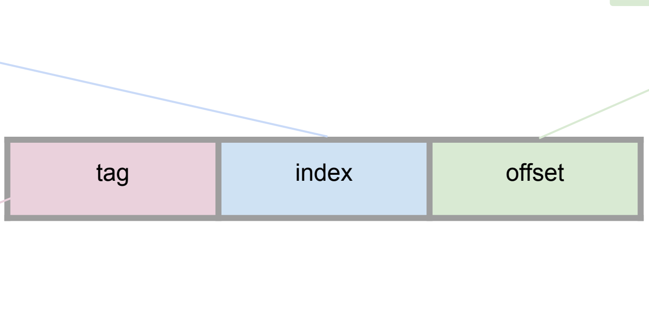
For a cache with 2 cache lines and 4 words per cache line…

We have 2 lines.
$2 = 2^1$

So, 1 bits is used for the index field.

Each word holds 4 bytes
So 4 words = 16 bytes

$16 = 2^4$
So, 4 bits are used for the offset field.



The remaining bits are used for the tag field.

1 bits for index, 4 bits for offset.

Each line has 4 words, 4 bits each, so each line has 16 bits in total.

16 - 5 = 11 bits for the tag field.

For a direct mapped cache of size 32 and each slot holds one byte, where should the data items at memory addresses 0x0 and 0x20 be?

tag field = 27 bits	index = 5 bits	offset = 0 bits

0x0 = 00000000
Last five bits = x0000, hence x0 will be mapped to line 0

x20 = 0010 0000
Last five bits = x0000, hence x20 will be mapped to line 0

Hence, why we need a tag value.

If a cache line consists of 16 bytes and we access the byte at address 0x1231, what is the range of the addresses of the bytes that are loaded into the cache? Assume only one line of the cache is loaded at a time.

Offset = 4
Because the cache line holds 16 bytes, so log(16) = 4

Minimum offset value = 0
Maximum offset value = #15 (0xF)

So, x1230 is the first address with offset 0, x1231 is the second address with offset 1, and so x123F would be the last address with offset F.

Hence, the range of address would be from x1230 to x123F

If a cache line consists of 32 bytes and we access the byte at memory address 0x4567, what is the range of the addresses of the bytes that are loaded into the cache? Assume only one line of the cache is loaded at a time.

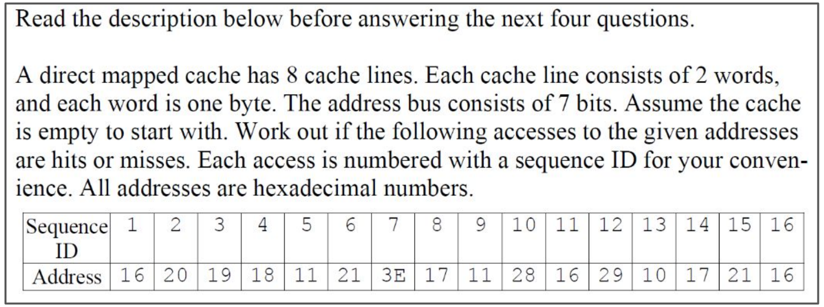
Offset = 5
x4567 = 0100 0101 0110 0111

minimum = 0100 0101 0110 0000 = x4560
maximum = 0100 0101 0111 1111 = x457F

A direct mapped cache has 8 cache lines. Each cache line consists of 2 words, and each word is one byte. The address bus consists of 7 bits. Which one of the following statements is correct?

a. The index field consists of 1 bit.
b. The index field consists of 2 bits.
c. The tag field of the cache consists of 2 bits.
d. The tag field of the cache consists of 3 bits

index = 3
offset = 1
tag = 7 - 4 = 3



Read the description below before answering the next four questions.

A direct mapped cache has 8 cache lines. Each cache line consists of 2 words, and each word is one byte. The address bus consists of 7 bits. Assume the cache is empty to start with. Work out if the following accesses to the given addresses are hits or misses. Each access is numbered with a sequence ID for your convenience. All addresses are hexadecimal numbers.

Sequence ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Address	16	20	19	18	11	21	3E	17	11	28	16	29	10	17	21	16

Each cache line has 2 words, 1 byte each, so each cache line consists of 2 bytes.

The address is 7 bits.

Each cache line has 2 bytes, so 2^1 = 2, so offset field has 1 bit.

We have 8 lines, so 2^3 = 8, so we have 3 bit for index field.

Remaining bits makes up tag field: 7 - 4 = 3

	TAG	WORD[0]	WORD[1]
0	010	x20	x21
1	010	x28	x29
2			
3	001	x16	x17
4	001	x18	x19
5			
6			
7	011	x3E	x3F

Address 1,2,3,4
x16 = 0001 011**0** (miss)
x20 = 0010 000**0** (miss)
x19 = 0001 100**1** (miss)
x18 = 0001 100**0** (hit)

Address 5,6,7
x11 = 0001 000**1** (miss)
x21 = 0010 000**1** (miss)
x3E = 0011 111**0** (miss)
x17 = 0001 011**1** (hit)

Address 8, 9, 10, 11, 12
x11 = 0001 0001 (miss)
x28 = 0010 1000 (miss)
x16 = 0001 0110 (hit)
x29 = 0010 1001 (hit)

Address 13, 14, 15, 16
x10 = 0001 0000 (hit)
x17 = 0001 0111 (hit)
x21 = 0010 0001 (miss)

HOW TO WRITE DATA

CPU → CACHE → MEMORY

IF DATA IS ALREADY IN THE CACHE

Write-Through - Write go to main memory and cache
When we write data item we update item in cache as well as in main memory (Completes only after memory write has completed, which takes a long time)

- keeps data in cache and main memory consistent
- takes a long time

- Reads entire block (cache line) from memory on a cache miss
- Writes only the updated item to memory for a store
- evictions do not need to write to memory

Write-Back
When writing, we only update data item in cache not main memory

- data in cache and memory is inconsistent (Upon evicting, we have to write data in cache back to main memory)
- Only one copy of data

- When we perform updates, we only update data in cache, so value stored in cache will not be the same as data stored in memory, upon eviction of an updated data item, we have to send this to main memory to ensure the update is reflected in main memory.

IF DATA IS NOT IN THE CACHE

Write-Allocate
First fetch data into cache, then compute write computation

No Write-Allocate
Send data straight into memory, don't write into cache

Write-Back Meta-Data

V	D	TAG	Byte 1	Byte 2	…Byte N

V = 1 means the line has valid data
D = 1 means the bytes are newer than main memory

When allocating line:
- Set V=1, D=0, fill in Tag and Data

When writing line:
- Set D=1

When evicting line:
- If D = 0; just set V=0
- If D = 1; write-back Data, then set D=0, V=0

Write-Through vs Write-Back

	WRITE-THROUGH **(keeps data in cache and main memory consistent but takes a long time)**	**WRITE-BACK** **(only update data item in cache not main memory)**
PROS	In a cache miss, write-through is generally better than write-through.	In a cache hit, write-back is faster than write-through on writes
CONS	In a cache hit, write-through is slower than write-back on writes In a cache miss, it needs to load a block of data. Write-through needs to write one data item to memory.	In a cache miss, write-back is slower than write-through if a dirty line has to be evicted. In a cache miss, it needs to load a block of data. Write-back needs to write a block of data to memory (if dirty)

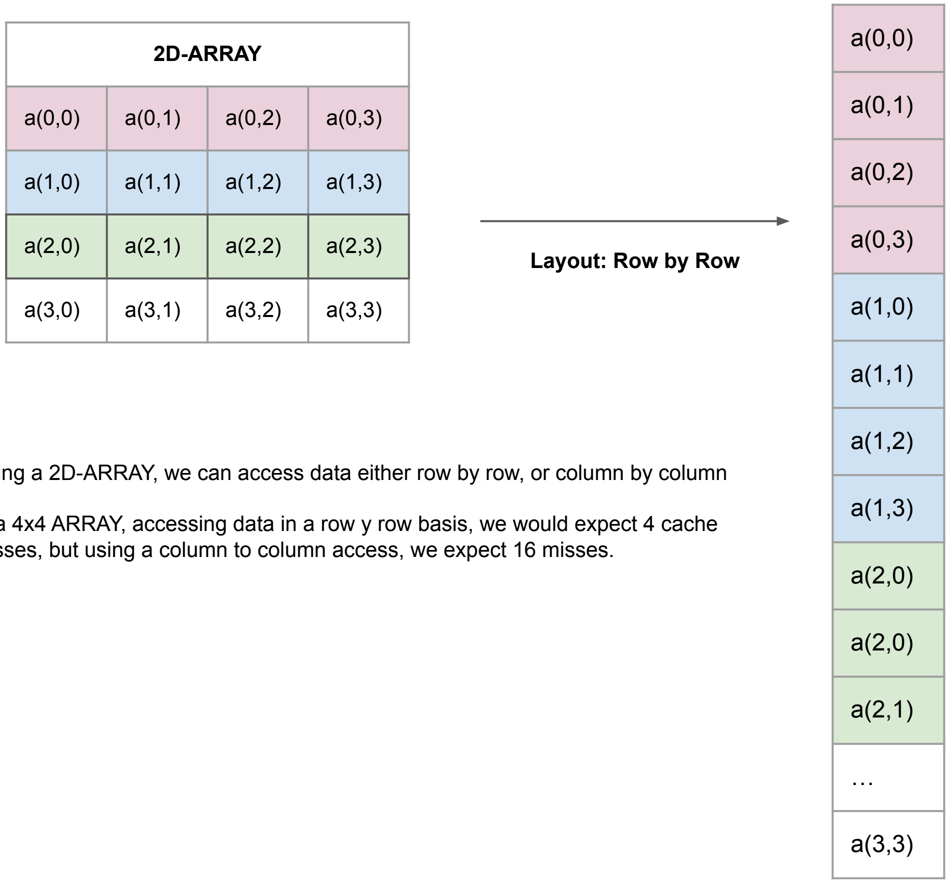
Write-Through
On a Cache Hit:
- Data is written to both the cache and the main memory.
- Ensures data consistency between cache and main memory, but it is slower because it requires two writes.

On a Cache Miss:
- Data is fetched from the main memory and written to both the cache and the main memory.
- Can be slower because it involves fetching data from the main memory and updating both places.

Write-Back
On a Cache Hit:
- Data is written only to the cache, and a "dirty bit" is set to indicate that the cache has the most recent data.
- Faster because it writes data only once (to the cache), reducing the write operations to the main memory.

On a Cache Miss:
- If the cache line to be replaced is dirty (modified), it is written back to the main memory first. Then, the new data is fetched from the main memory and updated only in the cache.
- Can be more complex as it requires checking the dirty bit and potentially performing additional write operations to the main memory.

HOW A 2D-ARRAY IS STORED IN MEMORY

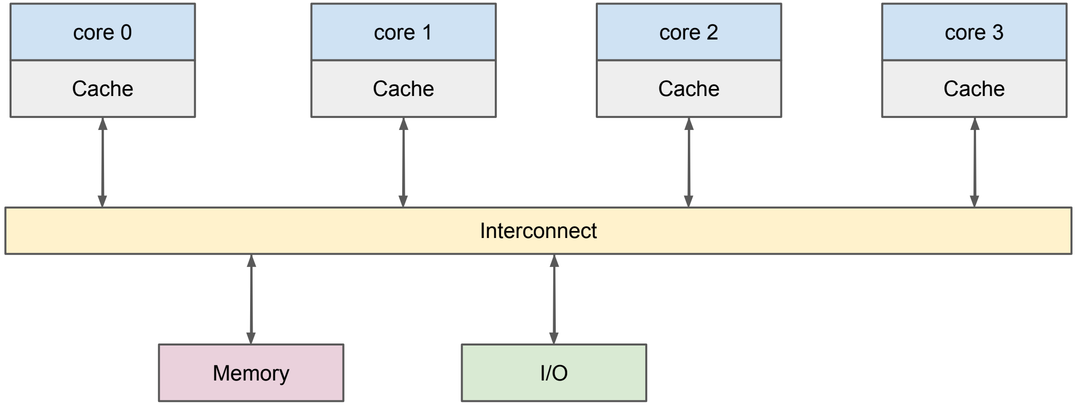

2D-ARRAY
a(0,0) a(0,1) a(0,2) a(0,3)
a(1,0) a(1,1) a(1,2) a(1,3)
a(2,0) a(2,1) a(2,2) a(2,3)
a(3,0) a(3,1) a(3,2) a(3,3)
Layout: Row by Row
a(0,0)
a(0,1)
a(0,2)
a(0,3)
a(1,0)
a(1,1)
a(1,2)
a(1,3)
a(2,0)
a(2,0)
a(2,1)
…
a(3,3)

Using a 2D-ARRAY, we can access data either row by row, or column by column

In a 4x4 ARRAY, accessing data in a row y row basis, we would expect 4 cache misses, but using a column to column access, we expect 16 misses.

VIRTUAL MEMORY

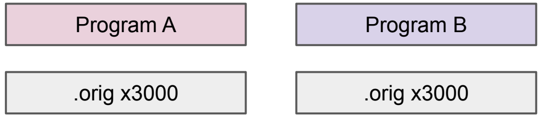
HOW CAN THE SINGLE MAIN MEMORY BE SHARED BY MULTIPLE PROCESSES?



Shared Memory Multiprocessors (SMP)
Each core can run a program independently as they have their own ALU and control logic.
Each core runs a program alone, but on a shared interconnect to share the same memory module.

Now, multiple program running at the same time, it might be assumed that they have access to all available memory. Now the issue is, what happens when another program is executed concurrently on another processor?

The addresses would conflict.



Both programs would try to access the same address in memory, meaning one program is likely going to overwrite the other.

SOLUTION 1: Microsoft and Google can sit together and decide on which memory addresses their programs can start at.

- Not a very viable option.

SOLUTION 2: Map a virtual address (generated by CPU) to a Physical Address (in memory) automatically.

- Each programmer thinks they can access any memory location (Can start program anywhere)
- When program loaded in system, the programs are mapped to different regions in memory (A virtual address is given, program can convert its address into the virtual address)

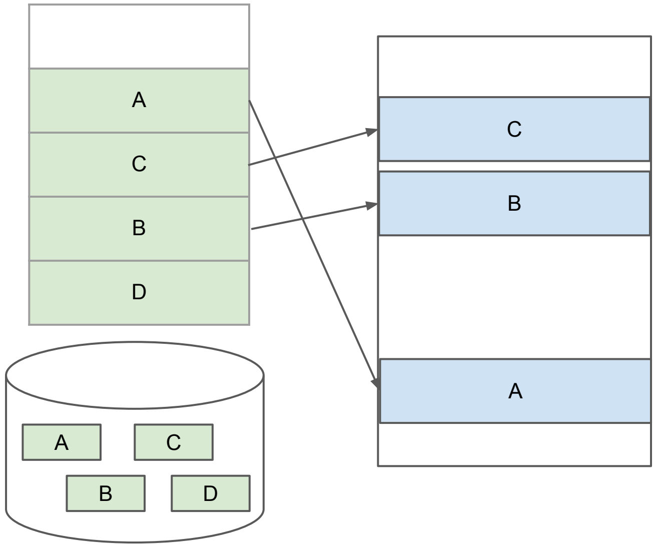
PROBLEM 2: NOT ENOUGH MEMORY

SOLUTION: Only load the instructions and data that are immediately needed into the memory

In virtual memory, it's just an abstraction, meaning we can technically store any amounts of data into Virtual Memory.
It appears to exist as main memory, but really, it corresponds to storage for information, without regard to its exact physical location. Therefore, VM supports multi-tasking, the ability to run more than one process at a time.

- Each process has its own virtual memory space.
- A process is an instance of the execution of a program
- Programmers can code as if they own all of memory
- Program/CPU can access any address from 0 to 2^N -1

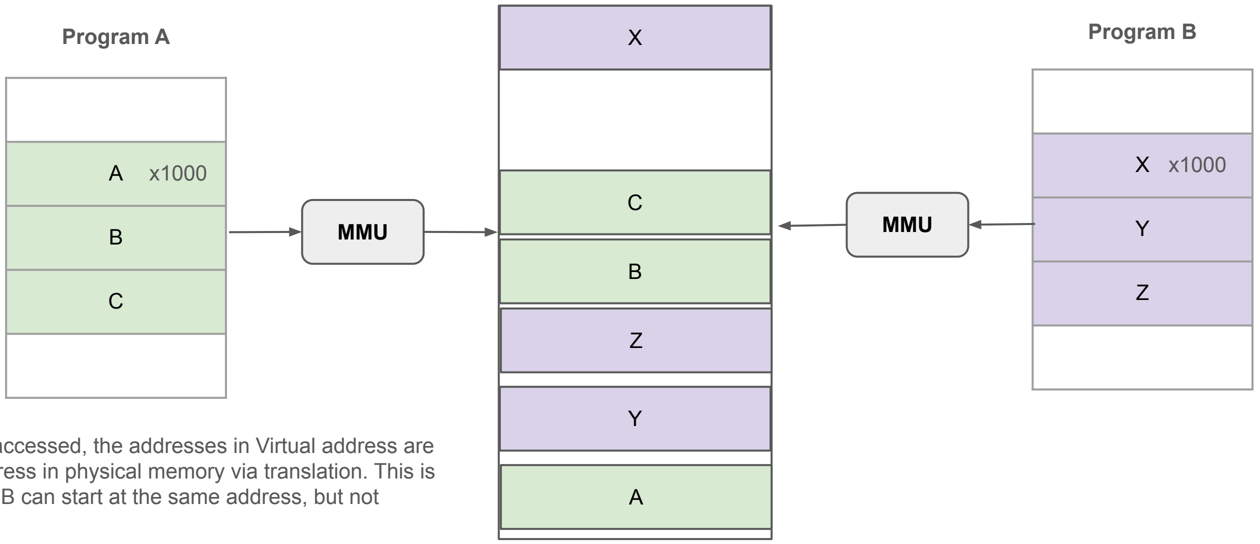
Physical memory refers to the actual physical chip in the computer



Order does NOT need to be preserved.
Physical page just finds an available space to hold virtual page.

MMU (Memory management unit) Allocated virtual pages to physical pages.

Both Program A and Program B start at x1000 (Virtually)



When the programs are accessed, the addresses in Virtual address are converted into a real address in physical memory via translation. This is how both Program A and B can start at the same address, but not overwrite each other.

Size of page table:

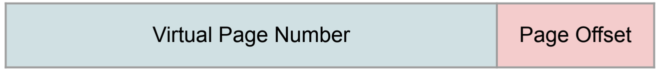
How large is the page table is there is 4GB of virtual memory, adn each page in physical memory is represented by 16 bits
Since 4GB = 4 x 2^{30} = 2^{32}

We would have 2 x 2^{32} = 2^{34} bytes in the page table. (16 bits = 2 bytes). In GB = 8GB

2^{10} = 1024 = 1K
1KB = 2^{10} bytes
1K of 1KB = 1MB = 2^{10} x 2^{10} bytes = 2^{20} bytes
1K of 1MB = 1GB = 2^{10} x 2^{20} = 2^{30} bytes

CONVERTING A VIRTUAL ADDRESS INTO A PHYSICAL ADDRESS

1. Partitional Virtual Address into a page number and offset



Page offset = n bits for a address consisting of 2^n bits
EG: If each page is 4K, then 4 x 2^{10} = 2^{12}, so offset = 12

The size of a virtual page is the same as the size of a physical page. First byte in virtual page is the first byte in the physical page, and the second byte in the virtual page is the same as the second byte in the physical page. Therefore, the position of a byte in the virtual page is the same as the position of the byte in the physical page.

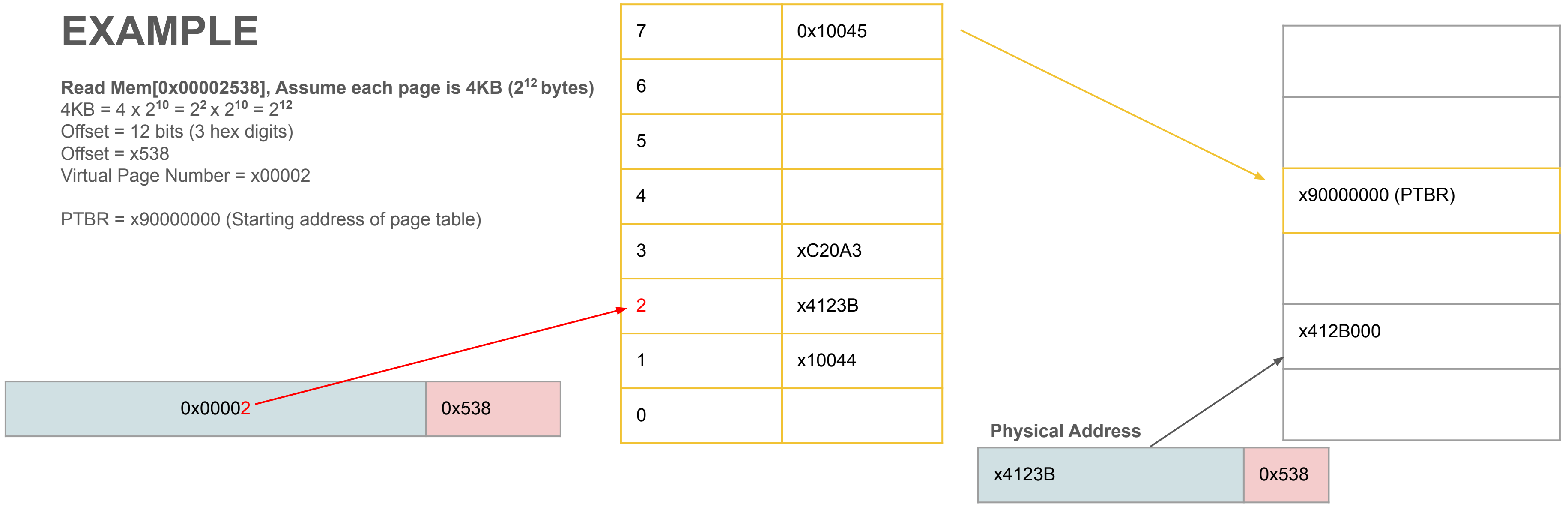
2. Lookup Page Table

Concatenate both to derive the physical address. Simply lookup the page number, and concatenate the physical page number with the offset value.

EXAMPLE

Read Mem[0x00002538], Assume each page is 4KB (2^{12} bytes)
4KB = 4 x 2^{10} = 2^2 x 2^{10} = 2^{12}
Offset = 12 bits (3 hex digits)
Offset = x538
Virtual Page Number = x00002

PTBR = x90000000 (Starting address of page table)



7	0x10045
6	
5	
4	
3	xC20A3
2	x4123B
1	x10044
0	

2^{10} = 1024 = 1K
1KB = 2^{10} bytes
1K of 1KB = 1MB = 2^{10} x 2^{10} bytes = 2^{20} bytes
1K of 1MB = 1GB = 2^{10} x 2^{20} = 2^{30} bytes

EXERCISE

Assume that the virtual address consists of 32 bits, the size of each virtual page is 8KB, the physical address consists of 30 bits, and the first five entries of the page table are as below. Consider virtual address 0x00004321.

- What is the offset value? x0321
- What is the VPN? 00002 0321
- What is the PPN? x1812E 0321 (0001 1000 0001 0010 1110 0 0011 0010 0001)
- What is the physical address that corresponds to the virtual address? (0 0011 0000 0010 0101 1100 0011 0010 0001 = 0x3025C321

Read Mem[0x00004321], Assume each page is 8KB (8 x 2^{10} bytes = 2^3x 2^{10} = 2^{13})
Offset = 13 bits

x4321 = 0100 0011 0010 0001
Offset = 0 0011 0010 0001 = x0321
VPN = **0000 0000 0000 0000 010**0 0011 0010 0001

x0000 0000 0000 0000 010	0x0321

x00002	0x0321

4	x12345
3	x00891
2	x1812E
1	x0891A
0	x16789

x1812E	0x0321

INVALID PAGES

	KERNEL			USER			
Valid Bit	R	W	X	R	W	X	Number
0							Disk B
1							x10045
0							
0							
1							xC20A3
1							x4123B
1							X10044
0							

Page table also consists of condition bits to represent access permissions for data stored in corresponding virtual page.

KERNEL: OS System Permission
User: Permissions for Users

Sometimes, the physical memory is not large enough to gold all the instructions and the data of the programs.

Hence, we should use Paging/Swapping to counter this issue.

Paging/Swapping

Paging allows us to run processes larger than physical memory. (View memory as a "cache" for secondary storage)
Swap: Swap memory pages out to disk when not in use
Page: Page them back in when needed

A page fault occurs when the accessed location is not in the physical memory
- Null entry: illegal address (location is not allocated to the program to used)
- On disk: Bring in the page from disk

Use Temporal/Spatial Locality
- Pages used recently most likely to be used again soon

If a swapped-out page is pages into the physical memory later, it might be stored at a different physical address.

When V=0:
1. CASE1: Page entry is empty → instruction tries to access a memory location that hasn't been allocated for program to use (Illegal Address). System needs to terminate execution of program and generate an ERROR message (SEGMENTATION FAULT).
2. CASE2: Entry for this page is NOT empty. Corresponds to a virtual address that still resides on disk. Needs to be brought into main memory.