Computer organisation and architecture (CS31007)

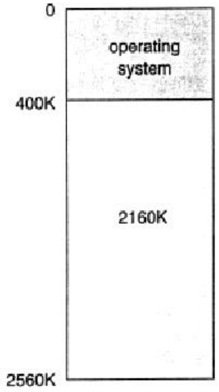
Department of Computer Sc & Engg
IIT Kharagpur

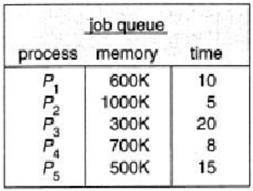
October 31, 2022

Scenario of memory usage

- Processes are allocated blocks of memory
- Memory released on process termination
- Initially, the entire available memory is one big hole
- When a process arrives, it is allocated contiguous memory from a hole large enough to accommodate it
- For such an allocation scheme, information is maintained about:
  1. allocated partitions
  2. free partitions (hole)
- Holes (block of available memory) of various size are scattered throughout memory as blocks of memory are allocated and freed
- This is called (external) fragmentation of memory
- Here, external indicates fragmentation happening outside allocated memory chunks

Memory requirements

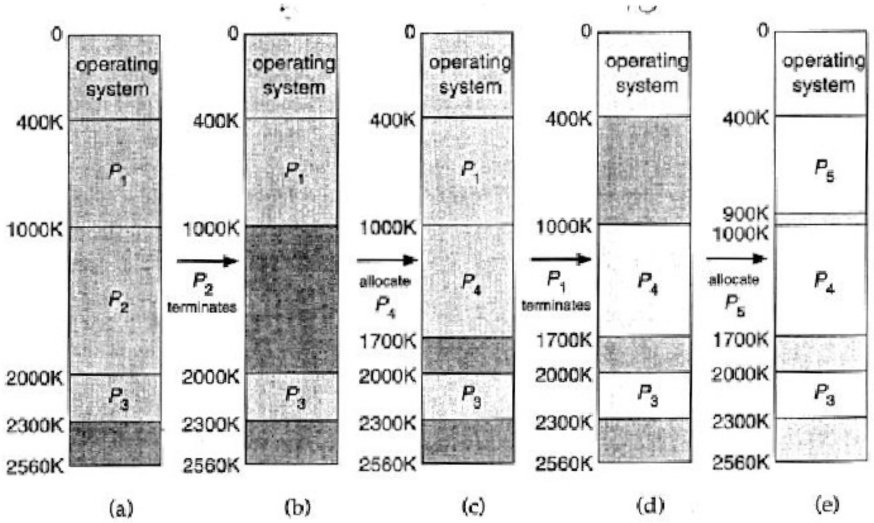




	job queue	
process	memory	time
P_1	600K	10
P_2	1000K	5
P_3	300K	20
P_4	700K	8
P_5	500K	15

P_6 200K ?

External fragmentation of memory by holes



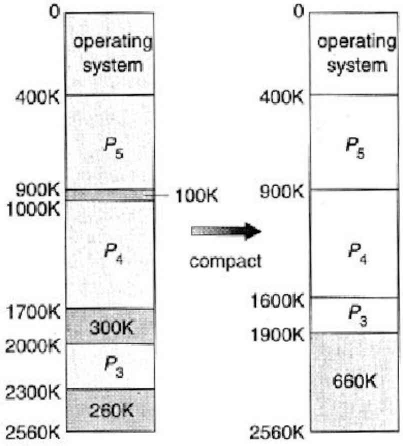
Common contiguous storage allocation strategies

First-fit:	Allocate the first hole that is big enough (fast, but fragments)
Best-fit:	Allocate the smallest hole that is big enough; must search entire list, unless ordered by size (slow, but small fragments)
Worst-fit:	Allocate the largest hole; must also search entire list (slow, but leaves large holes)

All of the above allocation strategies lead to varying degrees of external fragmentation.

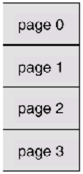
Problems from external fragmentation

- Total memory space exists to satisfy a request, but it is not contiguous
- Statistics indicate about 30% wastage
- External fragmentation may be handled by compaction
    - Move memory contents to coalesce all free memory together in one large block
    - Most absolute memory references must be translated – not all!
    - PC relative addresses are undisturbed

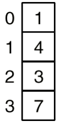


Paging

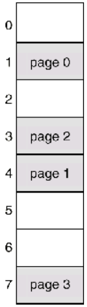
- Divide physical memory into fixed-sized blocks called frames (size is power of 2, between 512 bytes and 8192 bytes)
- Divide logical memory into (contiguous) blocks of same size called pages
- Keep track of all free frames
- To run a program of size *n* pages, need to find *n* free frames and load program (for now, assume all pages are loaded)
- Some space in the last allocated frame may remain unutilised – this is internal fragmentation



Logical pages



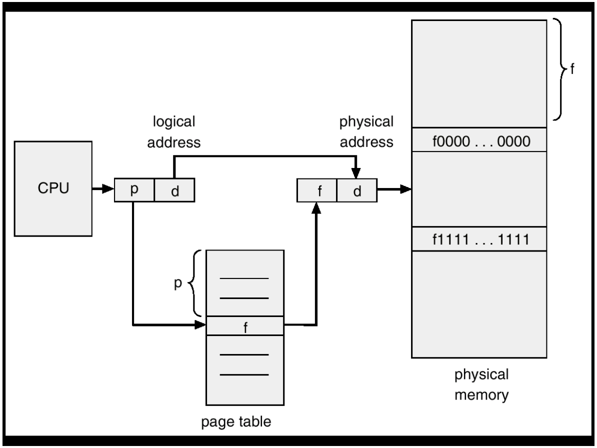
Page table for logical to physical mapping



Page frames and page mapping

Page address translation

The page-table base register (PTBR) stores the address in memory where the page table is stored (starting location)

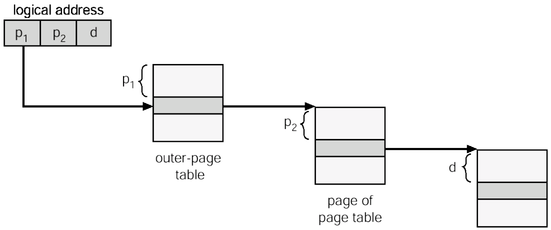


Address translation with paging

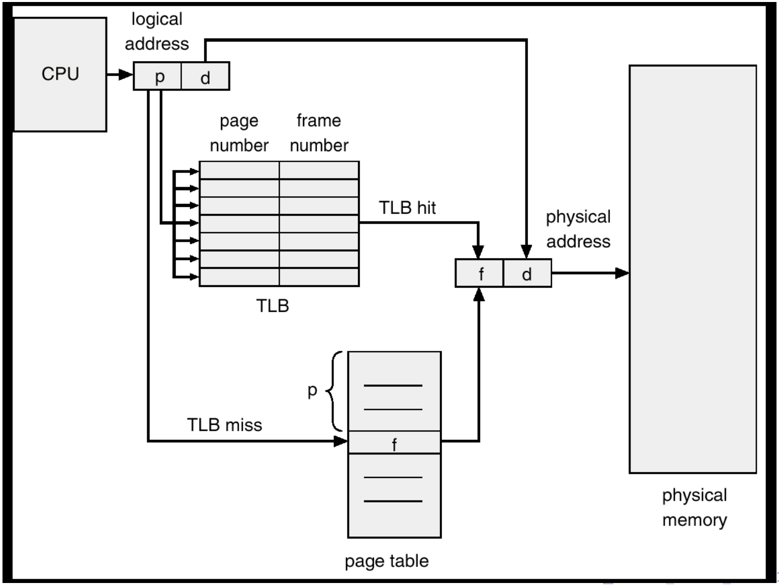
- In simple paging every data/instruction access requires two memory accesses
- One for the page table and one for the data/instruction
- Speedup is possible by the using a fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)
- Associative memory – parallel search

Multi-level paging

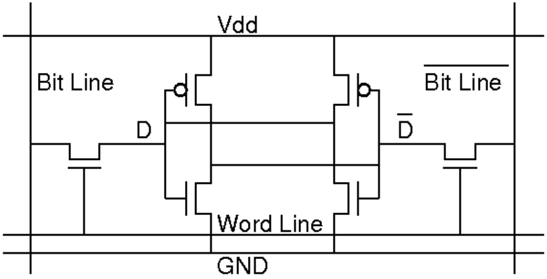
- Address bits: m
- Memory size: 2^m
- Page frame size: 2^k, $k < m$
- Size of page table: 2^{m-k} (worst case)
- Too much of memory wasted for page table
- Use multi-level paging (3-level, in practice)



Paging using TLB

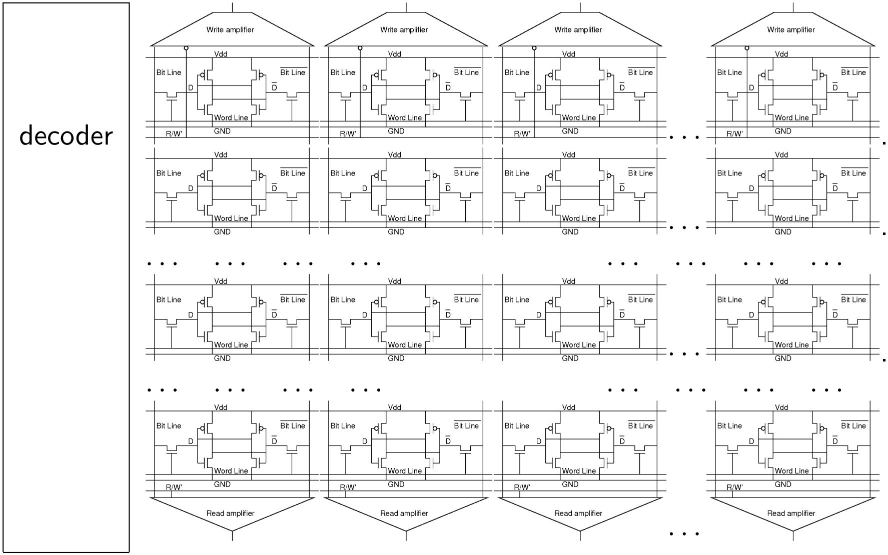


SRAM cell circuit for storage

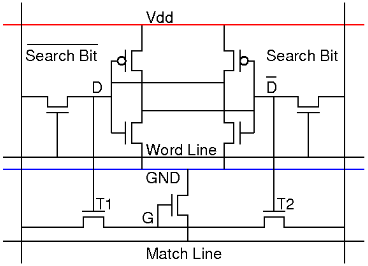


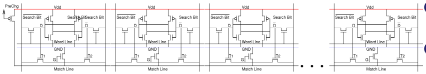
- Word Line is raised to a high voltage to enable the steering gates
- Bit Line and Bit Line are set to complementary values to force value into cross coupled inverter

Simple organisation of a SRAM



CAM cell circuit for matching





S	$\bar{D}$	$\bar{S}$	D	G	M
1	0	0	1	0	–
1	1	0	0	1	0
0	0	1	1	1	0
0	1	1	0	0	–

- Match line is pre-charged by a PMOS pull-up device
- Word Line is kept at a low voltage
- Search Line and $\overline{\text{Search Line}}$ are set to complementary values consistent with the bit of the key to be matched
- Note the inversion in polarity of the bit and search lines
- Match Line runs across all the bits in a key
- Match Line is pulled down when G goes high
- The pull-up, the Match Line and the transistors with gates marked G form a multi-input NOR gate
- Output of NOR is 0 (via pull-down) when there is a mismatch between the searh value and the stored value at any bit position
- Operation of cell is explained via table
- How to avoid matching uninitialised lines?

The Memory Hierarchy

Some typical characteristics of various storage media:

1. Processor registers:
   - 32 registers of 32 bits each = 128 bytes
   - access time = few nanoseconds
2. L1 cache memory (on-chip): LA $\rightarrow$ value, before address translation
   - capacity = 8 to 32 Kbytes
   - access time = 10 nanoseconds
3. L2 cache memory (on-chip/off-chip): PA $\rightarrow$ value, after address translation
   - capacity = few hundred Kbytes
   - access time = tens of nanoseconds
4. Main memory:
   - capacity = tens of Mbytes
   - access time = $\sim$100 nanoseconds
5. Hard disk:
   - capacity = few Gbytes
   - access time = tens of milliseconds

Cache operation

- The processor operates without stalling at high clock rate only when the memory items it requires are held in the cache.
- The overall system performance depends strongly on the proportion of the memory accesses which can be satisfied by the cache
- An access to an item which is in the cache: hit
- An access to an item which is not in the cache: miss.
- The proportion of all memory accesses that are satisfied by the cache: hit rate
- The proportion of all memory accesses that are not satisfied by the cache: miss rate
- The miss rate of a well-designed cache is low

Principle of cache operation – locality of reference

- During execution of a program, memory references by the processor, for both instructions and data, tend to cluster
- Once an area of the program is entered, there are repeated references to a small set of instructions (loop, subroutine) and data (components of a data structure, local variables or parameters on the stack)
- Temporal locality (locality in time): If an item is referenced, it will tend to be referenced again soon
- Spacial locality (locality in space): If an item is referenced, items whose addresses are close by will tend to be referenced soon
- The program is often found to spend 90% of its execution time in just 10% of the code

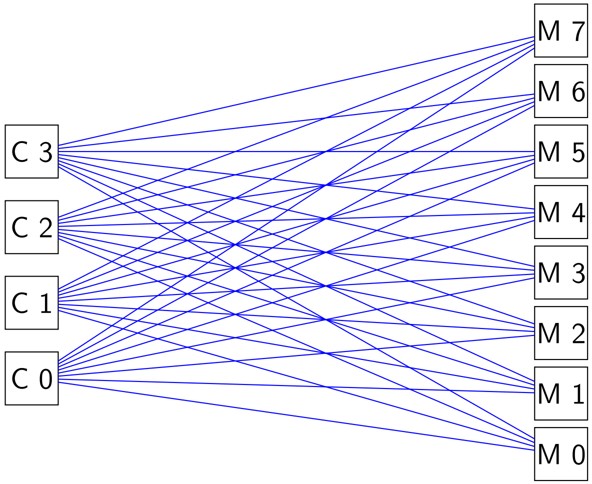
Cache and memory organisation

- Copy of data from memory is stored in fast memory close to the CPU
- Various mapping mechanisms are used
- Rather than reading a single word or byte from main memory at a time, multiple bytes are read
- These constitute a single cache entry known as a “cache line” or “cache block”
- A whole cache line is read and cached at once
- This help to utilise the principle of spatial locality of reference – if one location is read then nearby (following) locations are likely to be read soon afterward
- If the memory supports reading 2^k bytes at a time and the cache line is of length $2^k 2^n$ bytes, then 2^n memory banks are required
- A word at address m is placed in bank $\frac{m}{2^k} \mod 2^n$
- This is called 2^n-way memory interleaving
- Cache line lengths may be 64 bytes and 128 bytes; memories commonly support reading 1 to 4 bytes at a time

Cache and memory organisation

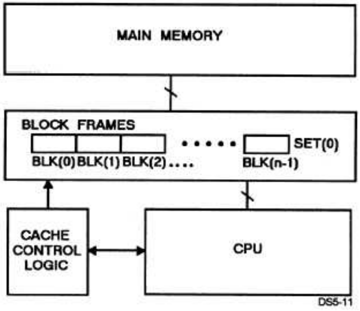
- Copy of data from memory is stored in fast memory close to the CPU
- Various mapping mechanisms are used
- Rather than reading a single word or byte from main memory at a time, multiple bytes are read
- These constitute a single cache entry known as a "cache line" or "cache block"
- A whole cache line is read and cached at once
- This help to utilise the principle of spatial locality of reference – if one location is read then nearby (following) locations are likely to be read soon afterward
- If the memory supports reading 2^k bytes at a time and the cache line is of length $2^k 2^n$ bytes, then 2^n memory banks are required
- A word at address m is placed in bank $\frac{m}{2^k} \mod 2^n$
- This is called 2^n-way memory interleaving
- Cache line lengths may be 64 bytes and 128 bytes; memories commonly support reading 1 to 4 bytes at a time

Fully associative mapping

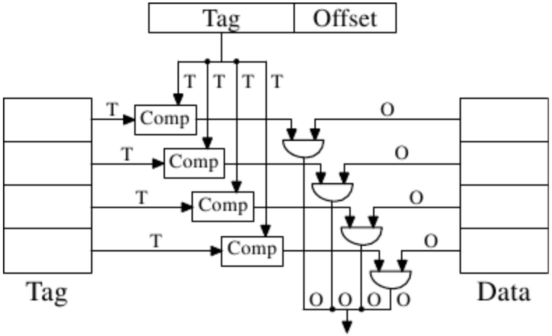


Fully associative mapping (contd.)

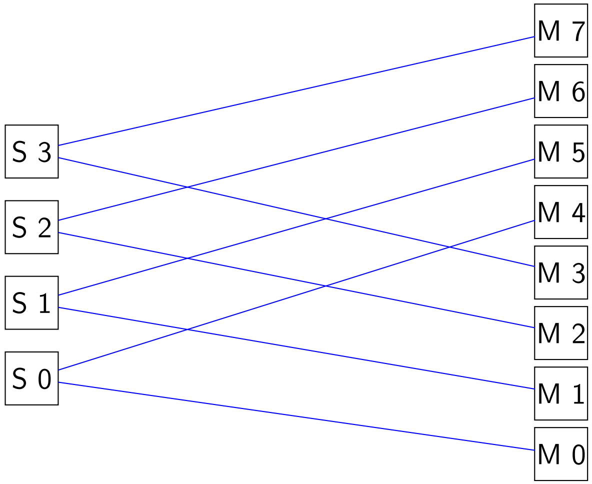
- Data can be at any location of cache
- Most flexible, maximum circuit cost
- Let CPU addresses have $t + o + w$ bits, where the CPU word length is 2^w bytes
- Let the cache line length be $l = 2^{o+w}$ bytes
- The tag (CAM key) is of t bits
- Let O (offset) be the number represent the o bits
- The O-th group of 2^w bytes from the cache line is chosen, if the tag matches with the higher order t bits of the CPU address



Fully Associative cache schematics

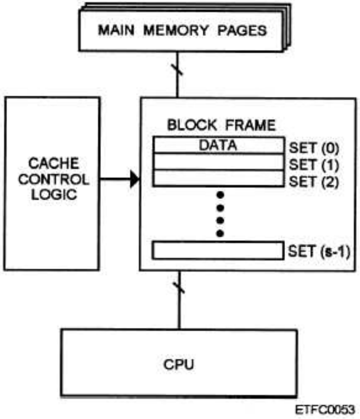


Direct mapping

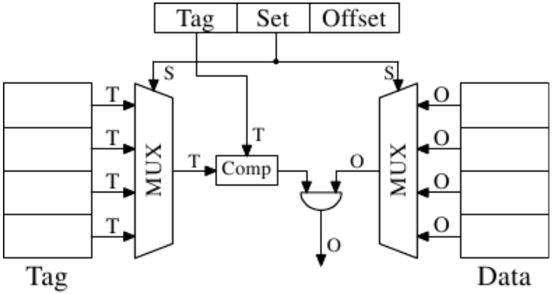


Direct mapping (contd.)

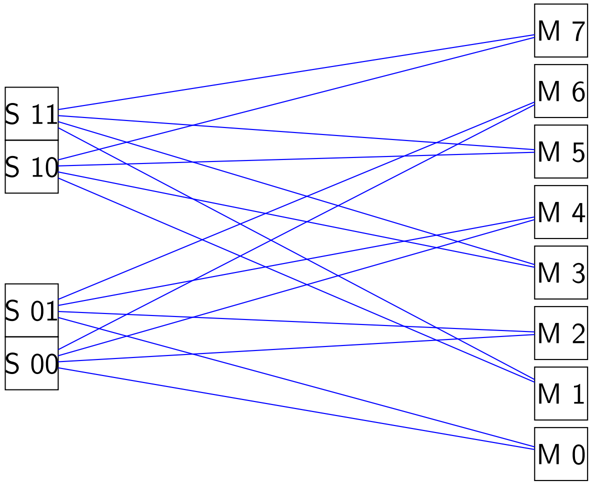
- Data only at fixed, set location of cache
- Least flexible, minimum circuit cost
- Let the cache have 2^s lines
- Let a CPU address a have $t + s + o + w$ bits, where the CPU word length is 2^w bytes
- Let the cache line length be $l = 2^{o+w}$ bytes
- Address a is mapped to set $S = \frac{a}{2^{o+w}}$ mod 2^t
- Let O (offset) be the number represent the o bits
- The O-th group of 2^w bytes from the cache line is chosen, if the tag of line S matches with the higher order t bits of the CPU address



Direct mapping cache schematics

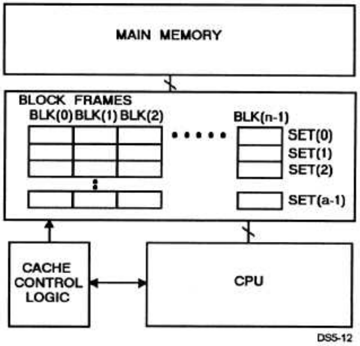


Set associative mapping

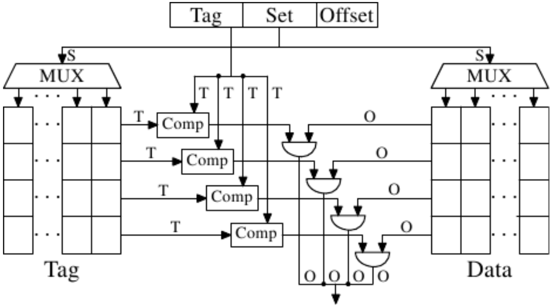


Set associative mapping (contd.)

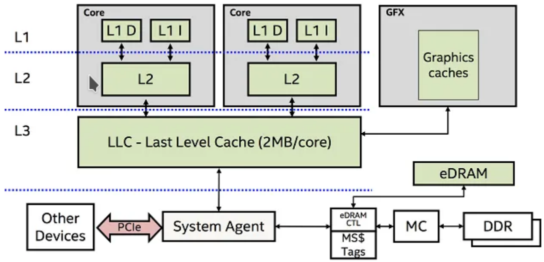
- Data can be at any location within a fixed set of cache locations
- Mapping is fully associative within the set
- Circuit cost intermediate
- Medium flexiblility



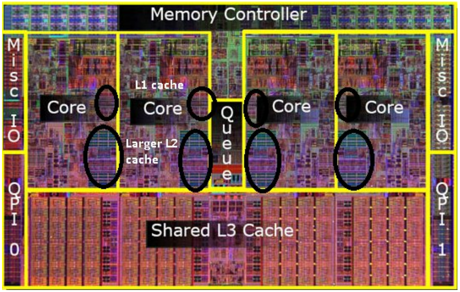
Set associative mapping cache schematics



Cache in CPU

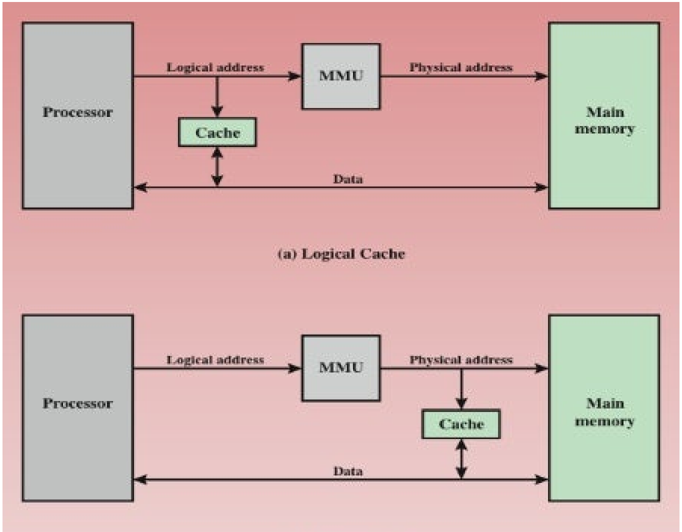


L1
L2
L3
Core
L1 D
L1 I
L2
Core
L1 D
L1 I
L2
GFX
Graphics caches
LLC - Last Level Cache (2MB/core)
eDRAM
Other Devices
PCIe
System Agent
eDRAM CTL
MS$ Tags
MC
DDR



Memory Controller
Misc IO
QPI 0
Core
L1 cache
Core
Larger L2 cache
Queue
Core
Core
Misc IO
QPI 1
Shared L3 Cache

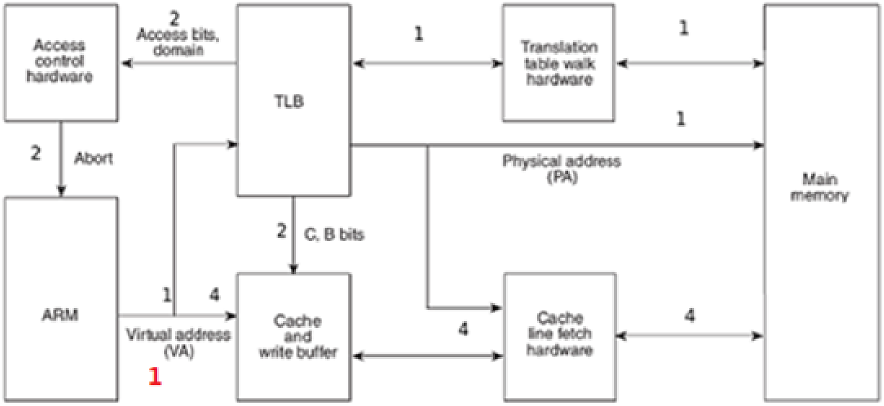
Logical / physical cache



(a) Logical Cache

(b) Physical Cache

Joint operation of caches and TLB



Cache write

Write-through

- when cache memory is updated simultaneously main memory is also updated
- main memory has same data available in the cache memory
- hardware needs sophistication to handle piling up of writes also redundant writes

Write-back

- memory is updated only during replacement or clearing of cache
- cache coherence needs separate handling
- simpler hardware

Example (Average memory access time)

In a certain system the main memory access time is 100 ns. The cache is 10 time faster than the main memory and uses the write though protocol. If the hit ratio for read request is 0.92 and 85% of the memory requests generated by the CPU are for read, the remaining being for write; then the average access time consideration both read and write requests may be estimated.

- $T_{\text{avg}} = hc + (1-h)M$, where h is the hit rate, so $(1-h)$ is the miss rate, c is the cache access time and M is the main memory access time (miss penalty)
- $T_{\text{avg}} =$ 0.85(avg time for read request)+ 0.15(avg time for write request)
- $T_{\text{avg}} = 0.85(0.92*10+0.08*100)+0.15$(avg time for write request)
- hit ratio is 0% for write request (write through)
- $T_{\text{avg}} = 0.85(0.92*10+0.08*100)+0.15(0*110+1*100) = 29.62$ ns

Some Cache Architectures

Intel 80486

- a single on-chip cache of 8 Kbytes
- line size: 16 bytes
- 4-way set associative organization

Pentium

- two on-chip caches, for data and instructions.
- each cache: 8 Kbytes
- line size: 32 bytes
- 2-way set associative organization

PowerPC 601

- a single on-chip cache of 32 Kbytes
- line size: 32 bytes
- 8-way set associative organization

Some Cache Architectures (contd.)

PowerPC 603

- two on-chip caches, for data and instructions
- each cache: 8 Kbytes
- line size: 32 bytes
- 2-way set associative organization (simpler cache organization than the 601 but stronger processor)

PowerPC 604

- two on-chip caches, for data and instructions
- each cache: 16 Kbytes
- line size: 32 bytes
- 4-way set associative organization

PowerPC 620

- two on-chip caches, for data and instructions
- each cache: 32 Kbytes
- line size: 64 bytes
- 8-way set associative organization

L1, L2, L3 caches

- Speed-wise: L1 $>$ L2 $>$ L3
- Size-wise: L1 $<$ L2 $<$ L3
- Cost-wise: L1 $>$ L2 $>$ L3 (per bit)
- Technology: varying between CPUs
- Speed variation with technology: SRAM $>$ SDRAM $>$ DRAM
- Speed variation with size: bigger $<$ smaller
  - transmission line delays on bit lines (once for matching key and once for transmission of data value)
  - farther off from CPU $\implies$ more delay
  - off-chip much slower than on-chip
- Older CPUs had L2 cache off-chip
- Now some CPUs even have the L3 cache on-chip
- For word addressable RAMs: longer address $\implies$ more time to decode (not directly applicable to caches)

Belady's anomaly

Belady's anomaly proves that it is possible to have more access faults when increasing the number of frames while using FIFO replacement algorithm. LRU based replacement does not exhibit this anomaly.

Example (Belady's anomaly for FIFO replacement policy)

Access	3	2	1	0	3	2	4	3	2	1	0	4
Newest	3	2	1	0	3	2	4	4	4	1	0	0
		3	2	1	0	3	2	2	2	4	1	1
Oldest			3	2	1	0	3	3	3	2	4	4

9 faults happen with 3 available places

Access req	3	2	1	0	3	2	4	3	2	1	0	4
Newest	3	2	1	0	0	0	4	3	2	1	0	4
		3	2	1	1	1	0	4	3	2	1	0
			3	2	2	2	1	0	4	3	2	1
Oldest				3	3	3	2	1	0	4	3	2

10 faults happen with 4 available places

Least Recently Used behaviour

Example (LRU replacement policy)

Access req	3	2	1	0	3	2	4	3	2	1	0	4
Newest	3	2	1	0	3	2	4	3	2	1	0	4
		3	2	1	0	3	2	4	3	2	1	0
Oldest			3	2	1	0	3	2	4	3	2	1

10 faults happen with 3 available places

Access req	3	2	1	0	3	2	4	3	2	1	0	4
Newest	3	2	1	0	3	2	4	3	2	1	0	4
		3	2	1	0	3	2	4	3	2	1	0
			3	2	1	0	3	2	4	3	2	1
Oldest				3	2	1	0	0	0	4	3	2

8 faults happen with 4 available places

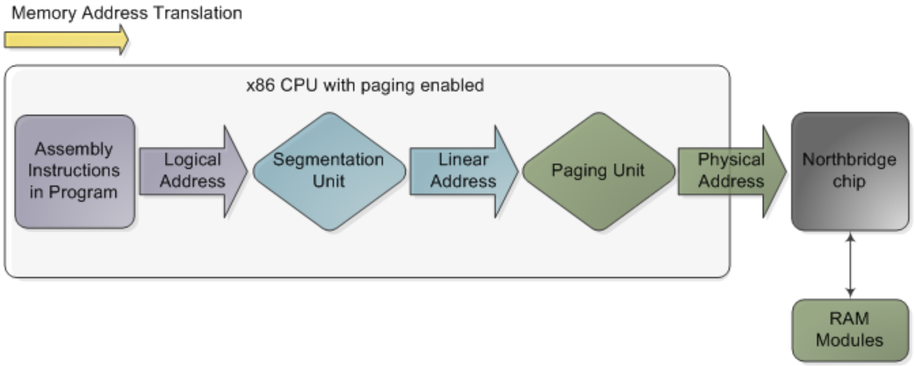
Segmentation

- Original 8086 had 16-bit registers
- Instructions used mostly 8-bit or 16-bit operands
- Size of address space: 2^{16} bytes or 64K
- Keen on letting the CPU use more memory without expanding
- the size of registers and instructions
- Introduced segment registers as a means to tell the CPU which
- 64K chunk of memory to use
    - first load a segment register
    - 16-bit memory addresses interpreted as offsets into selected segment
- Four segment registers
    - one for stack (ss)
    - one for program code (cs)
    - two for data (ds, es)

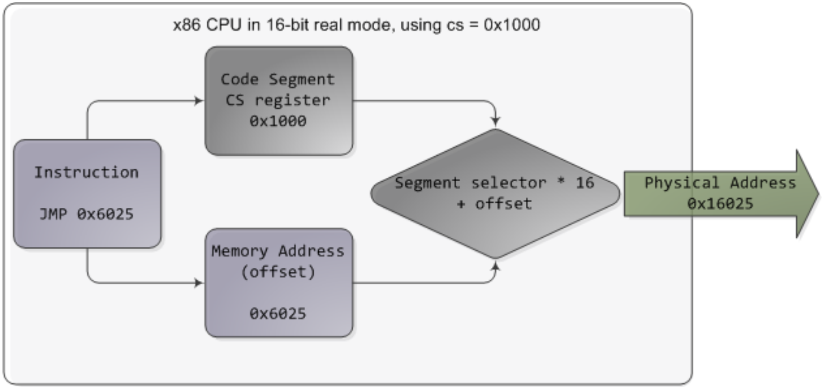
Segmentation (contd.)

- Segmentation is still present and is always enabled in x86 processors
- Each instruction implicitly uses a segment register
- E.g. a jump instruction uses the code segment register (cs)
- E.g. a stack push instruction uses the stack segment register (ss)
- Possible to explicitly override the segment register used by an instruction
- Segment registers store 16-bit segment selectors
- They can be loaded directly with instructions such as MOV
- CS can only be changed by instructions that affect the flow of execution

Address translation with segmentation

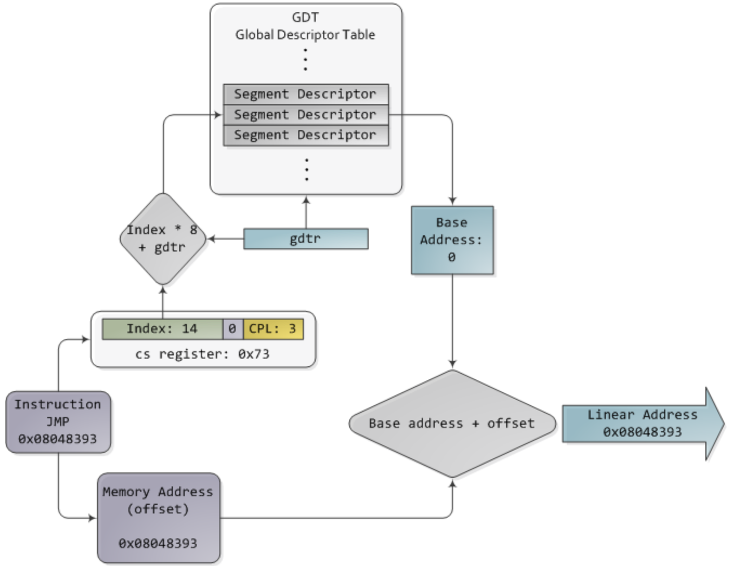


Segmentation in real mode


x86 CPU in 16-bit real mode, using cs = 0x1000
Code Segment CS register 0x1000
Instruction JMP 0x6025
Memory Address (offset) 0x6025
Segment selector * 16 + offset
Physical Address 0x16025

- Intel made a strange decision to multiply the segment selector by 2^4 (or 16), which confined memory addresses to about 1MB
- Real mode segments start from 0 to 0xFFFF0 in 16-byte increments
- A 16-bit offset (the logical address) is added to the segment address
- Address construction is not unique

Segmentation in real mode (contd.)


GDT
Global Descriptor Table
Segment Descriptor
Segment Descriptor
Segment Descriptor
Index * 8
+ gdtr
gdtr
Base
Address:
0
Index: 14
0
CPL: 3
cs register: 0x73
Instruction
JMP
0x08048393
Base address + offset
Linear Address
0x08048393
Memory Address
(offset)
0x08048393