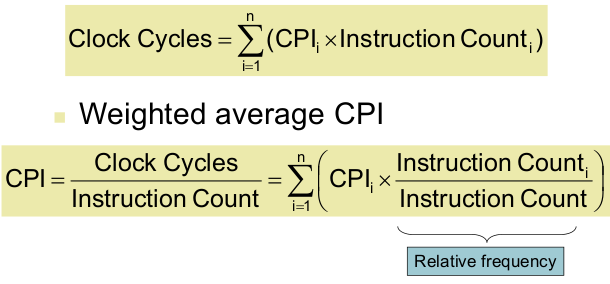
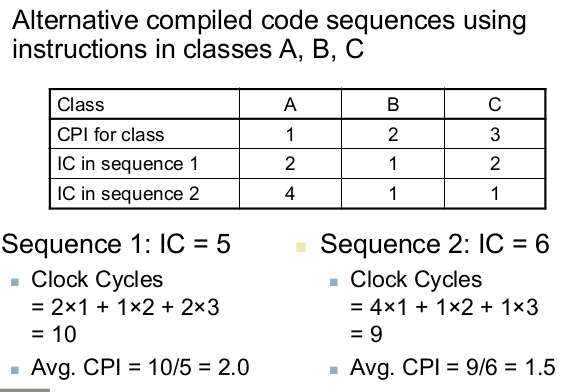
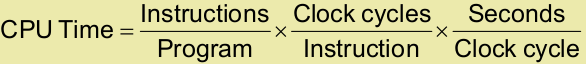
C is Not a Low-level Language.

Computer science pioneer Alan Perlis defined low-level languages this way:

“A programming language is low level when its programs require attention to the irrelevant.”

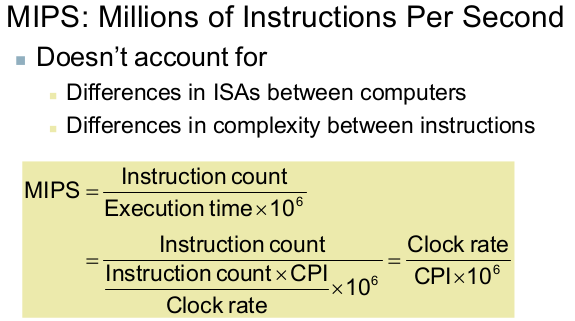
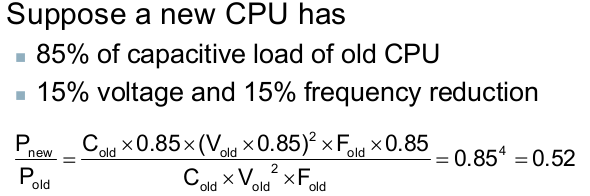
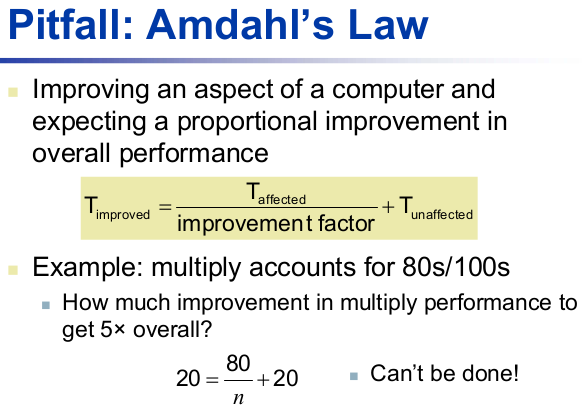
A typical heuristic for C code is that there is a branch, on average, every seven instructions. If you wish to keep such a pipeline

full from a single thread, then you must guess the targets of the next 25 branches.



In CMOS IC technology:

**Power = Capacitive load x Volt2 x Freq**

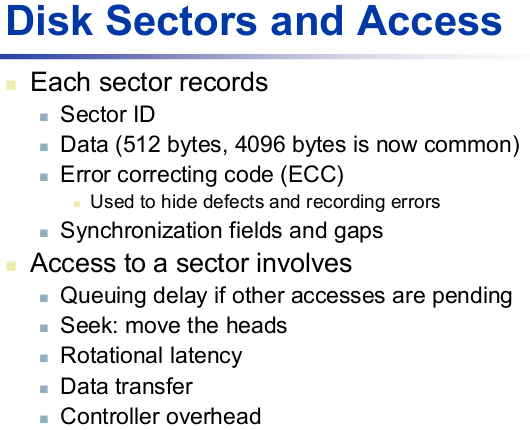


**Unsigned Binary Integer:**

32bits: [0, +4,294,967,295]

**2’s Complement Signed Integer:**

32bits: [–2,147,483,648,+2,147,483,647]



**MIPS Register numbers:**

**Data types:**

* Instructions are all 32 bits
* byte(8 bits), halfword (2 bytes), word (4 bytes)
* a character requires 1 byte of storage
* an integer requires 1 word (4 bytes) of storage

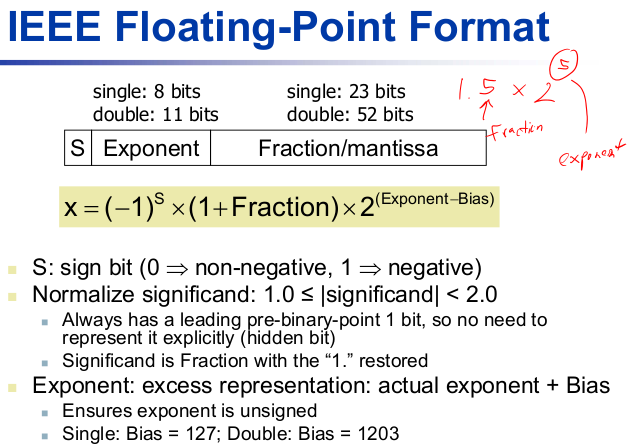
**Literals:**

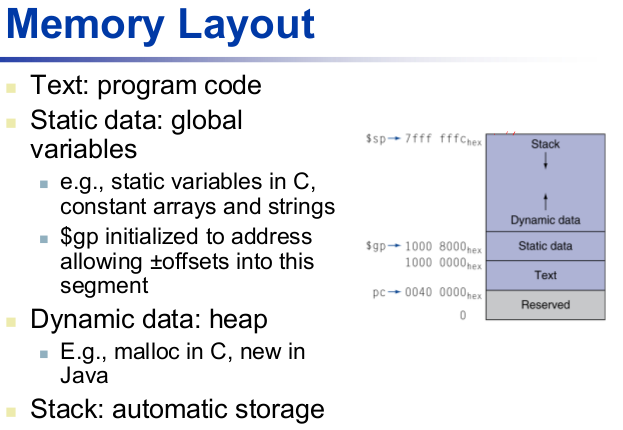
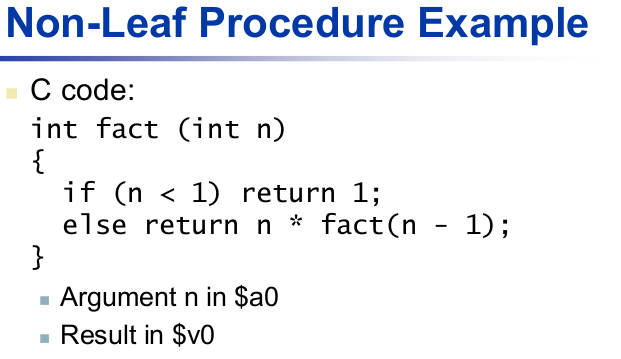
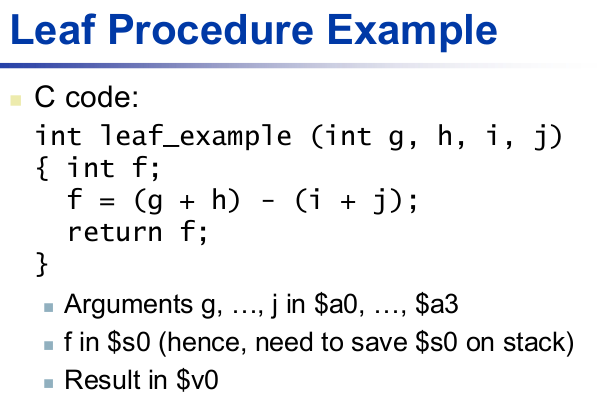
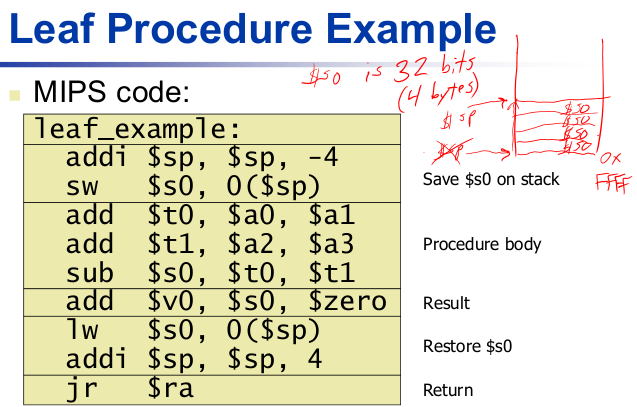
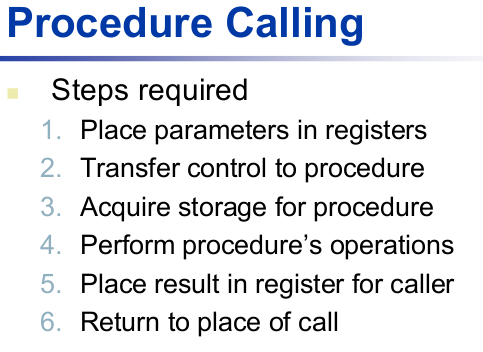
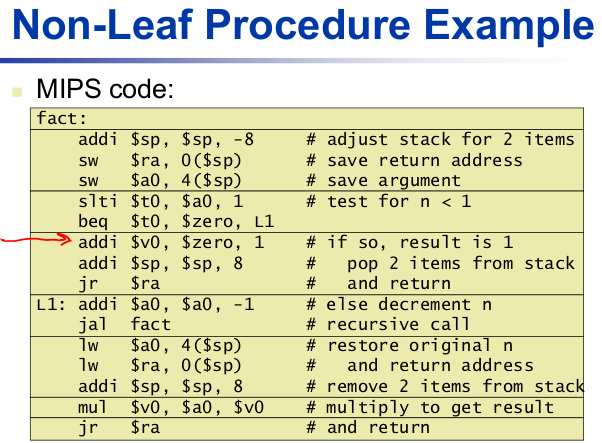
* numbers entered as is. e.g. 4
* characters enclosed in single quotes. e.g. 'b'
* strings enclosed in double quotes. e.g. "A string"

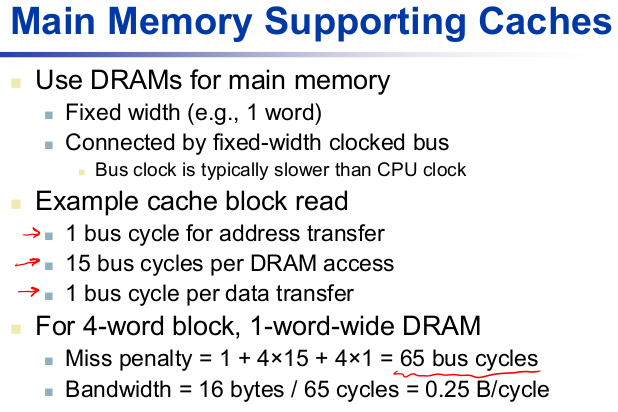
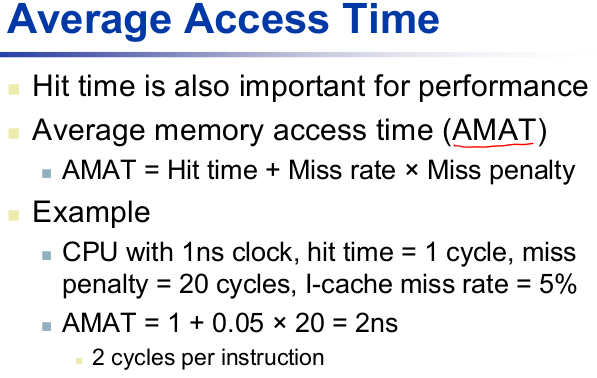
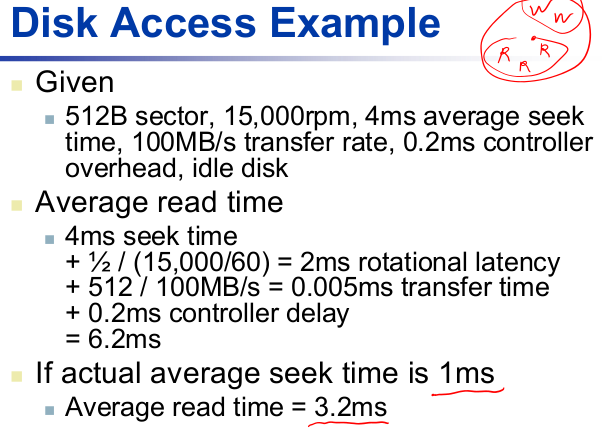
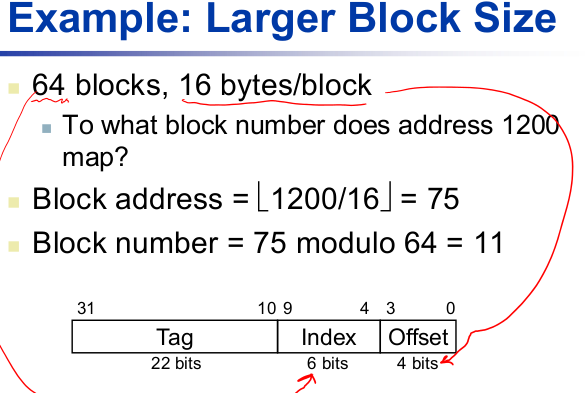
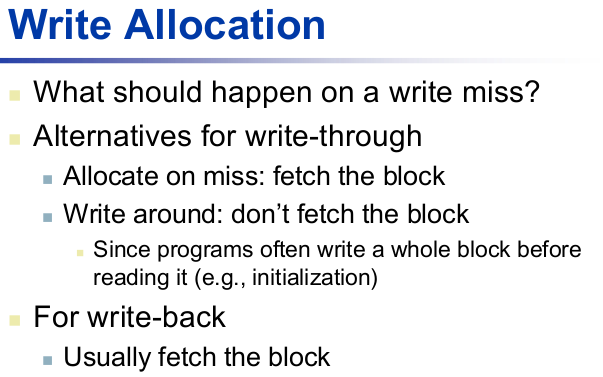
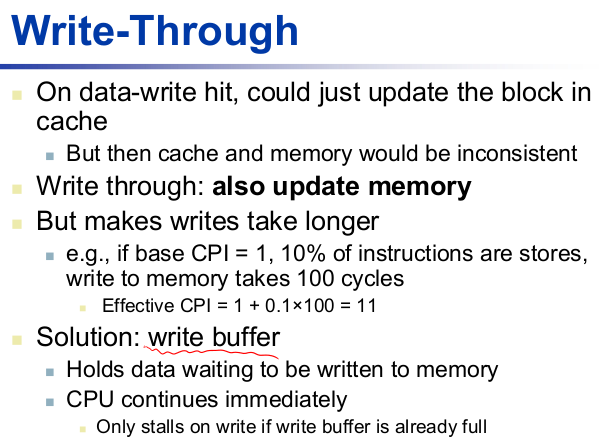
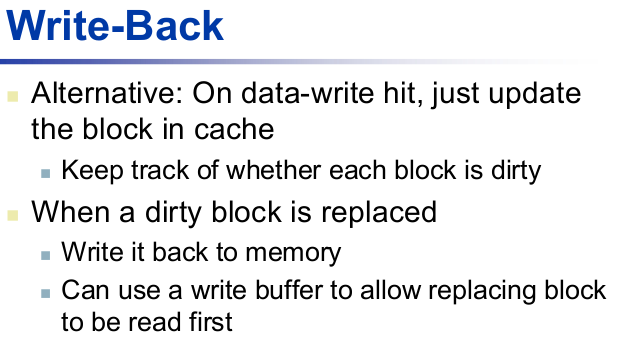
**Registers**

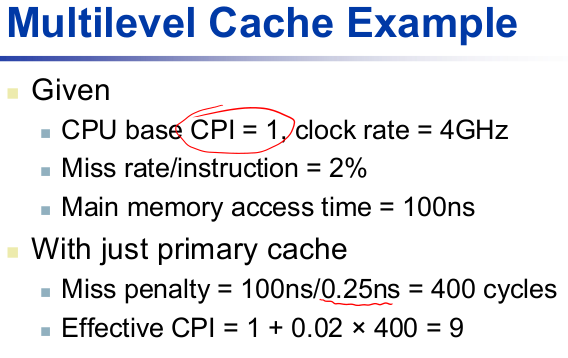
* 32 general-purpose registers
* register preceded by $ in assembly language instruction
* two formats for addressing:
  + using register number e.g. $0 through $31
  + using equivalent names e.g. $t1, $sp
* special registers Lo and Hi used to store result of multiplication and division
  + not directly addressable; contents accessed with special instruction mfhi ("move from Hi") and mflo ("move from Lo")
* stack grows from high memory to low memory

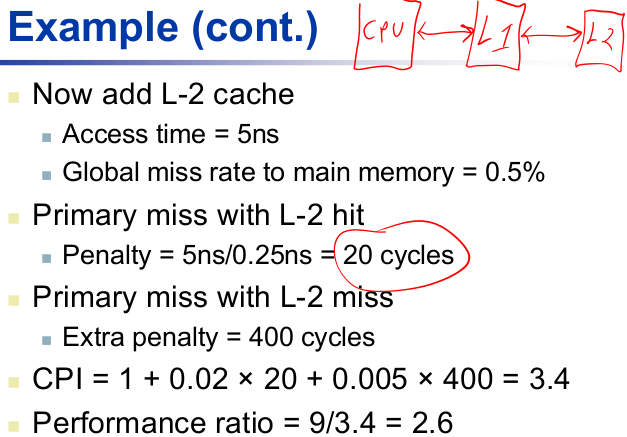
|  |  |  |
| --- | --- | --- |
| **Reg#** | **Alt** | **Description** |
| 0 | zero | the value 0 |
| 1 | $at | (assembler temporary)  reserved by the assembler |
| 2-3 | $v0 - $v1 | (values) from expression evaluation and function results |
| 4-7 | $a0 - $a3 | (arguments) First four parameters for subroutine.  Not preserved across procedure calls |
| 8-15 | **$t0 - $t7** | (temporaries) Caller saved if needed. Subroutines can use w/out saving.  Not preserved across procedure calls |
| 16-23 | **$s0 - $s7** | (saved values) - Callee saved.  A subroutine using one of these must save original and restore it before exiting.  Preserved across procedure calls |
| 24-25 | $t8 - $t9 | (temporaries) Caller saved if needed.  Subroutines can use w/out saving.(These are in addition to $t0 - $t7 above.)  Not preserved across procedure calls. |
| 26-27 | $k0 - $k1 | reserved for use by the interrupt/trap handler |
| 28 | $gp | **g**lobal **p**ointer.  Points to the middle of the 64K block of memory in the static data segment. |
| 29 | $sp | **s**tack **p**ointer  Points to last location on the stack. |
| 30 | $s8/$fp | saved value / **f**rame **p**ointer  Preserved across procedure calls |
| 31 | $ra | return address |











|  |  |  |
| --- | --- | --- |
| **Tag** | **Index** | **Offset** |
| 31-9 (23bits) | 8-5 (4bits) | 4-0 (5bits) |

= **8words/ block**





= **512 bytes of entries**

Including space for the valid bits, tags, and actual block data, how many bits would be required to actually implement this hypothetical cache?

**Total Cache size =**

**( # of Rows ) \*( valid bit + tag bits + data bits )**

= **4,480 bits**

**Number of pages in a table**

**= Total possible Logical Address Entries / Page size**

**Total size of page table**

**= Page Table Entry \* Number of pages**

Q) For a 4KiB page, and a 32 bit address space, calculate the amount of memory needed to store a process's page tables. Assume each entry in the page table requires 12 bytes. Show all calculations.

**Page** = 4KiB = 4,096 bytes = 212 bytes

**Address space** = 32bit

**PTE** = 12 bytes

**max # of page** = 2(32-12) = 220 = 1Mi pages

**Total size of a page table** = 12 bytes \* 220 = 12 MiB

|  |  |  |
| --- | --- | --- |
| **Tag** | **Index** | **Offset** |
| 31-6 (26bits) | 5-3 (3bits) | 2-0 (3bits) |

a. What is the cache block size (in words)?

3 bits allocated to the byte offset.

2 3 = 8 bytes  8/4 = 2 words

b. How many block indices does the cache have?

3 bits allocated to the index. 2 3 = 8 entries (i.e. indices)

c. Including space for the valid bits, tags, and actual block data, how many bits would be required to actually implement this hypothetical cache?

For each row index

(since there are N=2 block entries for each index):

2 \* 8 bytes of data = 128 bits

2 \* 1 valid bit for each entry = 2 bits

2 \* 26 bits for each tag = 52 bits

Adds up to 182 bits per index

182 bits \* 8 entries = 1456 total bits required

(In real life there could actually be additional requirements such as an LRU bit,dirty bit, etc. For this problem you didn’t have to take those into consideration.)

Block Address = floor(byte address/block size in bytes)

Block Index = (block address % num of indices)

