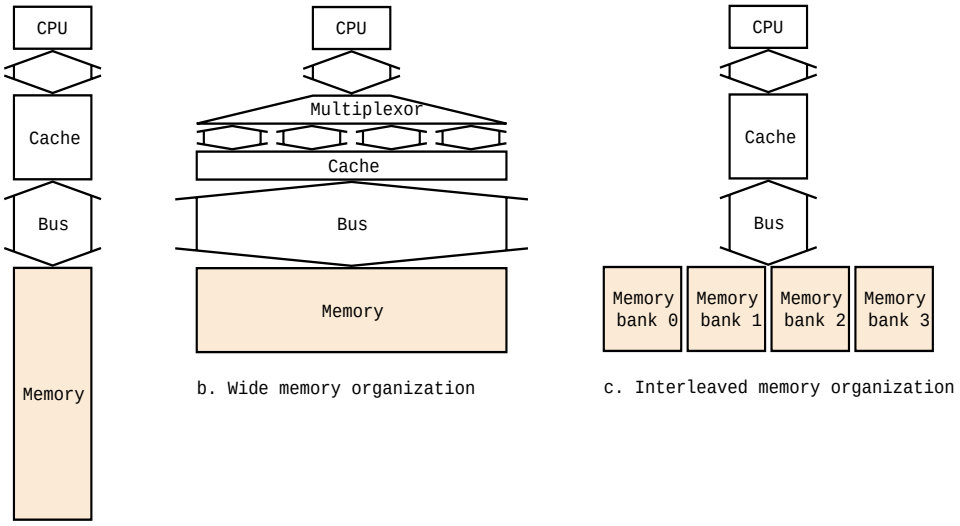
Main Memory (Fig. 7.13)



a. One-word-wide memory organization

b. Wide memory organization

c. Interleaved memory organization

Main Memory

Each memory access

- 1 cycle address
- 5 cycle DRAM (really 10+)
- 1 cycle data
- 4 word cache block

one word wide: (a=addr, d=delay, b=bus)

- addddddbddddddbddddddbddddddbddddddb
- 1 + 4 *(5+1) = 25

Main Memory

Four word wide:

- addddddb
- 1 + 5 + 1 = 7

Interleaved (pipelined)

- addddddb
- ddddd b
- ddddd b
- ddddd b
- 1 + 5 + 4 = 10

Error Correcting Codes (ECC)

Assume small number of random errors - bit(s) get flipped

So in 1 word no errors > single error > two errors > >2 errors

Detection - signal a problem

Correction - restore data to correct value

Most common

- Parity - single error detection
- SECDED - single error correction; double bit detection

1-bit ECC

Power	correct	#bits	comments
nothing	0, 1	1	
SED	00, 11	2	01, 10 detect errors
SEC	000, 111	3	001, 010, 100 => 000 110, 101, 011 => 111
SECDED	0000, 1111	4	one 1 => 0000 two 1's => error three 1's => 1111

ECC

For SECDED

- # 1's: **0 1 2 3 4**
- result: **0 0 error 1 1**

Hamming distance

- no. of changes to convert one code to another
- All legal SECDED codes must be at Hamming distance 4

ECC

Reduce overhead by doing codes on word, not bit

- overhead
- # bits SED SECDED
- 1 1(100%) 3 (300%)
- 32 1 (3%) 7 (22%)
- 64 1 (1.6%) 8 (13%)
- n 1 (1/n) 1 + $\log_2 n$ + a little

64-bit ECC

64 bits data 8 bits check

dddd.....d ccccccc

use eight by 9 SIMMs = 72 bits

Intuition

- one check bit is parity
- other check bits point to
  - error in data
  - error in all check bits
  - no error

ECC

To store

- use $data_0$ to compute $check_0$
- store $data_0$ and $check_0$

To load

- read $data_1$ and $check_1$
- use $data_1$ to compute $check_2$
- syndrome = $check_1$ xor $check_2$

ECC Syndrome

Correction	Parity	Implication
0	0	data1==data0
n	0	flip bit n of data1 to get data0
x	1	signal error

Virtual Memory

Basic idea

- move data from disk and main memory like
- caches to/from main memory

But

- miss penalty for first byte is 1M cycles, not 10-100
- therefore engineered differently
- later, we will return to the 4 questions

Virtual Memory

Blocks are called pages

- typically 4K-16K
- fixed size per system

Picture (draw program pages in memory & disk)

Architecture presents programs with a simple view

- memory addressed with 32-bit addresses
- lw $1, 0x100028 => 0x100028 is the “virtual address”
- system maps VA to physical address (PA)
- 0x100028 -> 0xF028 (page 15, offset 28 for 4K page)

Virtual Memory

someone else and I run unrelated programs each

- lw $1, 0x100028
- VA must map to different PA

Thus, VA allows

- use more physical memory than system has
- think it is the only program running in memory
- think it always starts at address 0x0
- be protected from rogue programs
- start running when most of the program is still on disk

Virtual Memory

A VA miss is called a page fault

- an exception that saves the PC
- OS gains control and initiates disk access
- OS usually runs someone else in the meantime
- interrupt when disk access is complete
- original instruction restarts

Unlike cache misses, why is OS used to handle a page fault?

Address Translation

VA -> PA

E.g., 4K pages

Use page tables of 4B PTEs

- index with page offset
- address of PTE = PTBR + page offset*4

Address Translation

PTE contains

- page frame number
- valid bit
- protection bits

Each program has own PT; switch by chaging PTBR

Translation Buffer

VM causes 100% overhead - 2 memory accesses - PTE + data!

What to do?

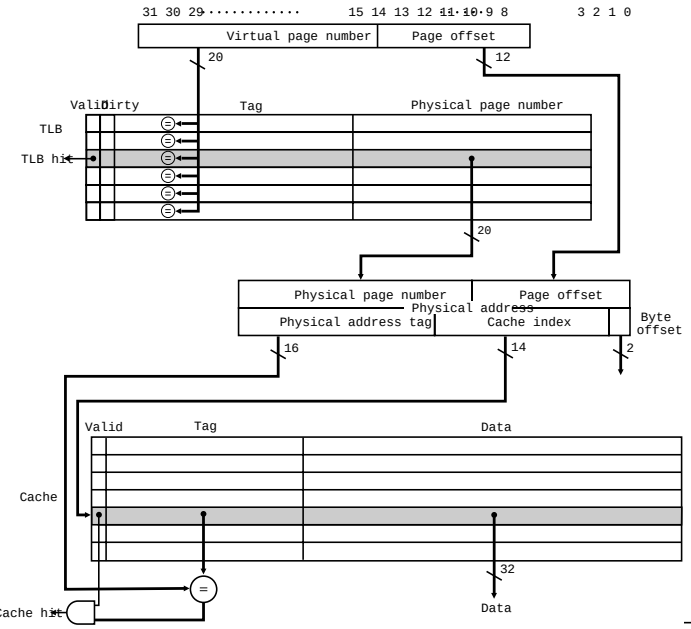
- temporal and spatial locality

Translation (Lookaside) Buffer

- a cache of translations
- valid tag data
- valid page# page frame# rest of PTE
- 1 20 20 12?

could make Fully/Set associative/Direct mapped

Example (Fig. 7.25)



Other Issues

Virtual address caches are also possible

- faster
- but synonym problem

On context switch

- change PTBR
- either flush TLB or add PIDs to TLB tags

Virtual Memory

4 Questions

Where is a page placed

- fully associative - any page on any frame

How is page found

- not associative search but indirection through PT

Virtual Memory

Which page is replaced

- approx LRU clock
- use page reference bit

What happens on a write

- write-backs
- use page dirty bit

Protection

User VAs map to different PAs - no overlap

But may want sharing

- user-user
- user-kernel (mode bit, syscall interface)
- In PTE and TLB entry
  - invalid (had before)
  - read-only
  - read-write (had before)

Page Table Size

How big is the PT?

- 2^{32}/4K * 4 = 4M per program

To make smaller

- define a limit register
- do limit registers for a few regions - stack, heap
- page a part of PT (terminate recursion)
- Segmented VA (noncontiguous alloc, segment table->PT)
- use Hash table to map PA-VA - called inverted PT

More Optimizations

Non-blocking caches

- handle hits under misses Interleaved/banked caches
- multiple requests simultaneously (poor-man's multiporting)

Write Buffers

- miss penalty of dirty blocks

Out-of-order CPU

- tolerate cache hit and miss latencies

More Optimizations

Compiler optimizations

- get rid of memory accesses (register allocation, reuse)
- improve locality (blocking, tiling)
- insert prefetch code
- scheduling

Real Stuff

DEC Alpha 21264 (550 MHz)

- L1 cache
  - 4 way out-of-order CPU pipeline
  - 2 loads/stores per cycle (phase pipelined)
  - 3 cycles hit latency, 8+ GB/s bandwidth
- L2 cache
  - 12 cycle hit latency, 4+ GB/s bandwidth
- System interface
  - 64 bit bus, 80 cycle latency, 2+ GB/s bandwidth

Real Stuff

Charac	Pentium Pro	PowerPC
VA	32 bits	52 bits
PA	32 bits	32 bits
Page size	4 KB, 4 MB	4 KB, select-able, 256 MB
TLB	split I and D 4-way assoc pseudo random I - 32, D - 64 TLB miss H/W	split I and D 2-way assoc LRU I - 128, D- 128 TLB miss H/W

Real Stuff

Charac	Pentium Pro	PowerPC
cache	split I and D	split I and D
size	8KB each	16 KB each
assoc	4-way	4-way
replace	approx LRU	LRU
block	32 bytes	32 bytes
write	write-back	write-back or write-through

Summary

Temporal and spatial locality, Memory hierarchy

Cache design - block size, associativity, write back/through

Multilevel cache hierarchies

Virtual memory, translation (VA -> PA), page table (PT)

VM design - page size, FA through PT, reference bit, dirty bit

Fast translations - TLB

Protection, page faults (exceptions)

Summary

4 Questions - cache, VM, TLB

- Where can a block be placed
  - one (DM), a few (SA), any (FA)
- How is a block found
  - indexing (DM), search (SA/FA), table lookup (PT)
- What is replaced on a miss
  - LRU or random
- How are writes handled
  - write through or write back; write back for VM