Chapter 9 Memory systems

Topics:

Technology issues

Cache design

Virtual memory and operating system support

Reading:

P&H 5.1-5.5

CPU speed vs. memory speed

Main memory:

- main storage for active programs
- Based on *Dynamic Random Access Memory* (DRAM)

In MIPS pipeline, organized into:

- Instruction memory
- Data memory

Each memory takes one cycle to access (or else…)

Typical CPU clock rate: 2 GHz

cycle time = 500 ps

Good reference on memory technology:
http://arstechnica.com/paedia/r/ram_guide/ram_guide.part1-1.html

What are DRAM speeds? According to
http://en.wikipedia.org/wiki/DDR2_SDRAM

In 2009, the fastest DDR2 SDRAM (*Double Data Rate Synchronous* DRAM) has cycle time < 4 ns.

Not good enough for our pipeline design!

"Instruction memory" and "data memory" are actually *instruction cache* and *data cache*.

A *cache* is a smaller, faster memory, between main memory and CPU, using *Static RAM* (SRAM) technology.

Memory hierarchy concepts

Levels of memory hierarchy:

- Registers
- Cache (1 to 3 levels; static RAM)
- Main memory (dynamic RAM)
- Disk / secondary storage (magnetic disk)

SRAM: .5-2.5ns access time, $2k-$5k per GB (2008)

DRAM: 50-70ns access time, $20-$75 per GB

Disk: 5M – 20M ns access time, $.2-$2 per GB

Closer to CPU: small and fast, expensive

Further from CPU: large and ~~fast~~ `slow`, cheap

Suppose we are running a program that works with some data.

Most of the time, we are just

- Executing a small number of instructions
  (ex.: loop, small block)
- Working with a small amount of data
  (ex.: a few variables, a small piece of an array)

This is the *principle of locality* observed in programs.

If we keep the few instructions/data that we need in the fast storage, then we can get to them quickly, *most of the time.*

Similar to keeping variables in registers!

Two types of locality

- Temporal locality

  If a memory address is referenced at the current time, the same address will be referenced again soon.

  Examples: `instructions in loop, recursive function`

  `loop counter, array index`

- Spatial locality

  If a memory address is referenced at the current time, an address close to it will be referenced soon.

  Examples: `instructions in straightline code`

  `sequential array access`

Technology issues: DRAM vs. SRAM

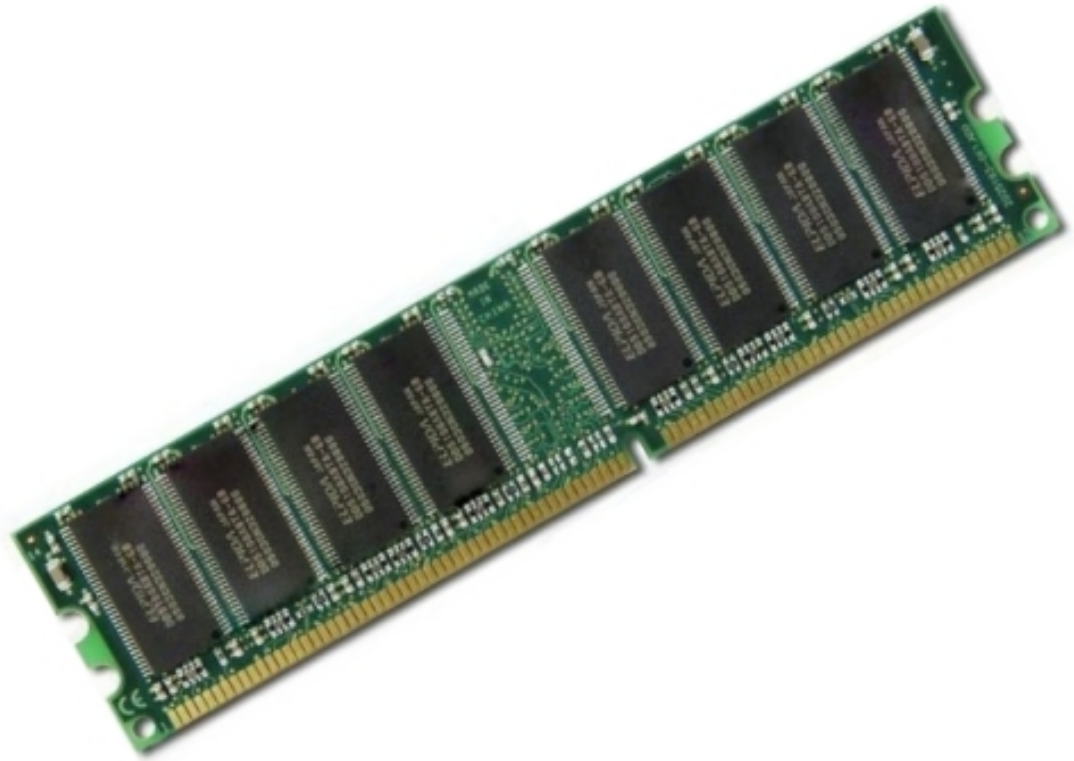
Main memory constructed from DRAMs
(dynamic random access memory)

- one transistor holds one bit
- destructive reads; periodically refreshed

DRAMs organized as dual inline memory modules
(DIMMs)

- typically 8 to 16 DRAM chips per DIMM
- usually 8 bytes wide

A 1 GB DDR2 DIMM:



Operation:

- address divided into row and column
- send row address and row access strobe (RAS)

  Read row of bits from memory array
- send column address and column access strobe (CAS)

  Select bit(s) from within row, send out to CPU

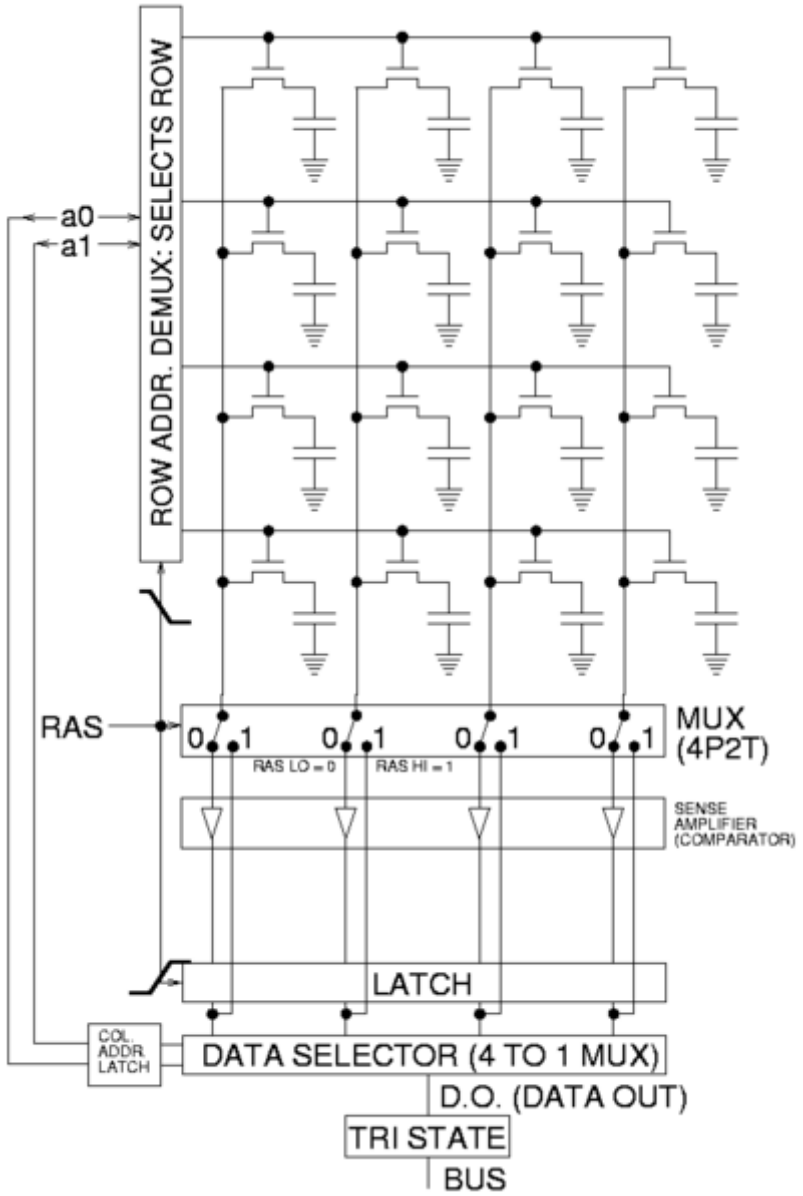
Early DRAMs were *asynchronous* (no clock)

Most fast DRAMs today are synchronous

Decent reference:

http://pcguide.com/art/sdramBasics-c.html

DRAM block diagram:



Performance considerations:

- DRAM much slower than CPU clock cycle time!
- Each memory access takes many cycles

Use static RAM (SRAM) for caches

SRAM basics:

- Reads are not destructive; no refresh necessary
- typically 6 transistors per bit
- DRAMs 4 to 8 times denser per chip than SRAMs
- SRAMs 8 to 16 times faster than DRAMs
- SRAMs 8 to 16 times as expensive than DRAMs

Terminology

Consider pair of connecting levels:

- Cache / main memory
- Main memory / disk

Upper level is fast but small.
Lower level is slower but larger.

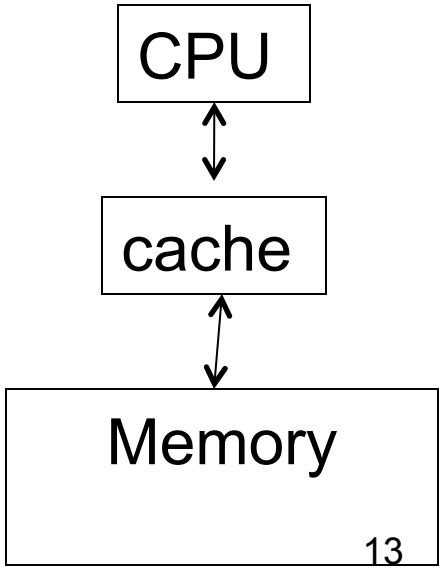
Goal: keep instructions/data in upper level when needed.

Terminology (con't)

Hit: an access found in the upper level
Miss: an access not found in the upper level

hit rate (or hit ratio) = no. of hits / no. of accesses
miss rate (or miss ratio) = no. of misses / no. of accesses

hit time = time for a hit
miss time = time for a miss
miss penalty = miss time - hit time



Basic principles of caches

Usually, main memory much slower than CPU.

Cache: small fast memory array close to CPU

(AMD Opteron: 64KB L1 instruction cache, 64KB L1 data cache, 1MB L2 cache)

Block or *line*: Basic unit of information transferred between cache and CPU

Simplest: one-word block

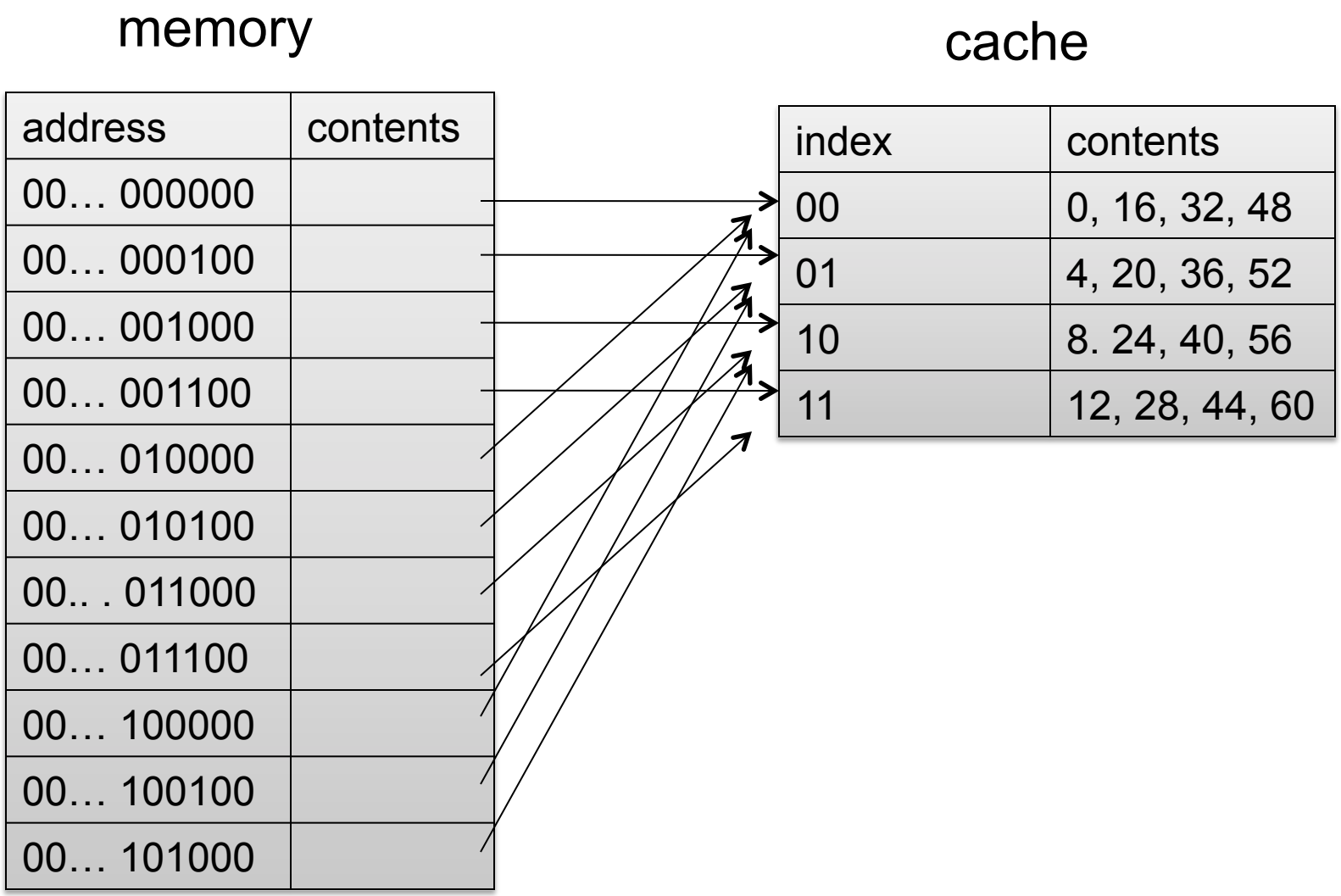
Since cache is much smaller than main memory, must hash 32-bit memory address to small cache index.

Direct-mapped cache with one-word block

Direct-mapped: modulo hashing

Suppose cache has 4 blocks.

memory word 0…0 00 0000	cache block 00
0…0 01 0000	
0…0 10 0000	
…	
memory word 0…0 00 0100	cache block 01
0…0 01 0100	
0…0 10 0100	
…	
memory word 0…0 00 1000	cache block 10
0…0 01 1000	
0…0 10 1000 …	
memory word 0…0 00 1100	cache block 11
0…0 01 1100	
0…0 10 1100 …	


memory
cache
address
contents
00… 000000
00… 000100
00… 001000
00… 001100
00… 010000
00… 010100
00.. . 011000
00… 011100
00… 100000
00… 100100
00… 101000
index
contents
00
0, 16, 32, 48
01
4, 20, 36, 52
10
8. 24, 40, 56
11
12, 28, 44, 60

Suppose cache has N blocks.

Hash 32-bit memory address into logN bit cache index.

Divide 32-bit address into:

1) byte offset (2 bits)
2) cache index (logN bits)
3) tag (remaining bits)

Note: each block in cache may contain a block from several different memory addresses!

The *tag* identifies which memory block is in that cache block.

Cache with 1024 blocks, each one word Fig. ~~5.7~~ 5.10:

Each cache slot contains:

- Valid bit (Is block empty, or contains real data?)
- Tag (identifies where data came from, if valid)
- Data

By convention: 16KB cache means cache contains 16KB *data* only

Usually have *separate* caches for instructions and data.

Example:

Suppose direct-mapped cache has 64KB data and one-word blocks. How many total bits does it contain? (32-bit memory address)

```
# blocks = 64K / 4 = 16K
# bits in index = log (16K) = 14
# bits in offset = 2 bits
# bits in tag = 32 - 14 - 2 = 16

each block contains valid bit, 16-bit tag, 32-bit data

total # bits in cache = 49 * 16K
```

Basic cache operation

Each cache block has *valid* bit to indicate full/empty.

Start: empty cache (or requested word not in cache)

Consider instruction cache:

- Get memory address from PC
- Hash memory address to get index, tag.
- Use index to access cache; valid bit off, *miss* (or tags do not match; *miss*)
- Turn on valid bit, move information from memory, store in cache block, store new tag in cache tag

Note: memory block brought into cache only when it is accessed (demand fetch)

Operations for a hit (instruction cache again):

- Get memory address from PC
- Hash memory address to get index, tag.
- Use index to access cache, match tags, discover *hit*
- Get information directly from cache block

Example: 4KB direct-mapped cache, 1-word blocks
Loop is at address 0x40 0020
Code fragment ($s0 initialized to 3):

```
loop:  [instr 1]
       [instr 2]
       addi $s0, $s0, -1
       bne $s0, $0, loop
```

Trace behavior of instruction cache for this loop.

Cache trace:

```
0x400020 Instr 1
         Instr 2
         Addi
         Bne
         Instr 1
         Instr 2
         Addi
         Bne
         Instr 1
         Instr 2
         Addi
         Bne
```

Cache index	V	contents tag	data
…	?	…	
1000	~~0~~ 1	0..010000000000	
1001	~~0~~		
1010	~~0~~		
1011	~~0~~		
…	?		

```
#blocks = 4KB/4 = 1K
10-bit index
20-bit tag
0x400020 = 0..0 0100 0000 0000 0000 0010 0000
```

Example: 4KB direct-mapped data cache, 1-word blocks

Code fragment:

```
lw $s4, 0($s0) # $s0 = 0x1001 0040
lw $s5, 0($s1) # $s1 = 0x1001 c040
lw $s6, 0($s0)
lw $s7, 0($s2) # $s2 = 0x1001 0840
lw $t0, 0($s0)
```

stopped 11/17/2016

Cache operation and pipelines

Usually one instruction cache, one data cache (split caches)

(Pipeline needs up to two accesses per cycle)

If cache hit, pipeline executes one instruction per cycle.

On cache miss, *stall* pipeline

When instruction/data transferred to cache from memory, pipeline resumes operation.

Multiword blocks

One-word cache blocks exploit temporal locality, not spatial locality.

Most CPUs have more than one word per cache block (block size and cache size usually given in bytes, not words)

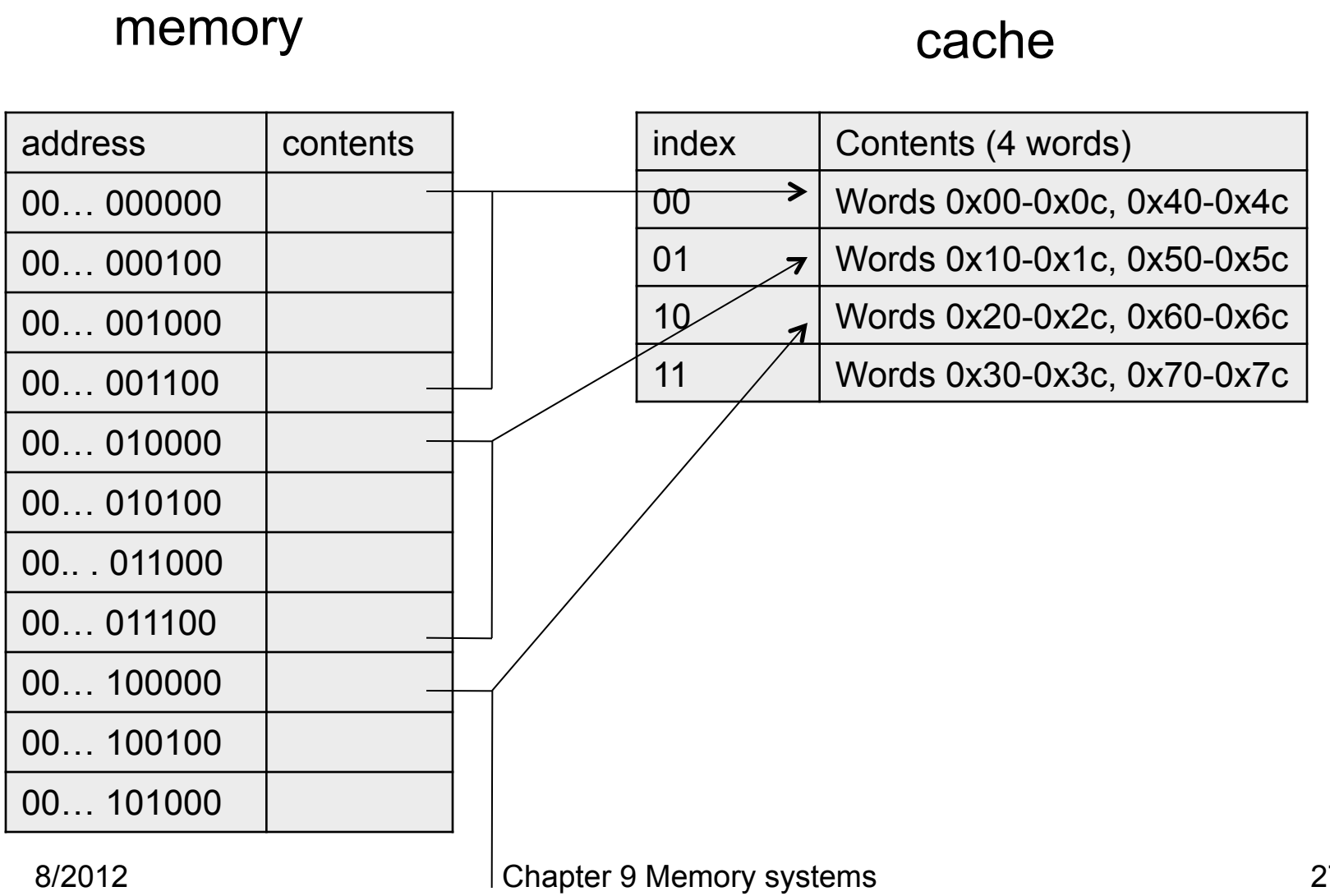
Consider cache of size C bytes, with blocks of size B bytes.

No. of blocks in cache = C / B

No. of bits in block offset = log B

No. of bits in index = log (C / B)

Determine block index and tag:



Note that words are grouped into blocks in a very rigid way! (Otherwise won't work properly…)

Example: 64KB cache with 16-byte blocks

Sketch cache tag and data layout…

Increasing block size usually decreases miss rate.

p. 465 Fig. ~~5.8~~ 5.11

But extremely large block sizes are bad!

Why?

Very large block sizes also increase miss times.

Example: 4KB direct-mapped cache, 4-word blocks
Loop is at address 0x40 0020
Code fragment ($s0 initialized to 4):

```
loop:  [instr 1]
       [instr 2]
       addi $s0, $s0, -1
       bne $s0, $0, loop
```

Write-through and write buffers

So far, only consider reads and read misses.

What about writes?

Suppose we have a 4KB data cache, 4-byte blocks

Consider MIPS code sequence:

```
lw $t0, ($s0)        # $s0=0x10010008, cache index=2
add $t0, $t0, 1
sw $t0, ($s0)
```

After lw $t0, ($s0):

cache

Cache index	contents
2	4
etc	

mem

Mem address	contents
0x10010008	4
etc	

After sw $t0, ($s0):

Cache index	contents
2	5
etc	

???

Mem address	contents
0x10010008	4
etc	

Effect of write: cache block and memory block become *inconsistent* (contain different data)

Simplest write policy: *write-through*

For each write to the cache, write the data to memory.

On write miss:

- Fetch block from memory into cache
- Write word to cache
- Write word to memory; stall until write is done

On write hit:

- Write word to cache
- Write word to memory; stall until write is done

Both write hits and write misses are very slow!

Write-through operation on write hit:

After lw $t0, ($s0):

cache

Cache index	contents
2	4
etc	

mem

Mem address	contents
0x10010008	4
etc	

After sw $t0, ($s0):

Cache index	contents
2	5
etc	

Mem address	contents
0x10010008	5
etc	

Improvement: use a *write buffer*

(small FIFO with hardware that handles write to memory)

On a write to cache, write to write buffer

Write buffer hardware will complete write to memory

Do not stall unless write buffer is full

(If there are too many writes, write buffer will be full often!)

Better write policy: writeback

Each cache block has *dirty bit*

- indicates whether cache block is different from memory block.

On a write, only write to cache block.

- But turn on dirty bit.

Dirty block written to memory when it is kicked out (replaced) by another block from memory.

Writeback operations in detail

Read hit: no change

Write hit: write into cache only, set dirty bit

Read miss:

- if cache block dirty, write to memory
- get new block from memory, place in cache
- clear dirty bit

Write miss:

- if cache block dirty, write to memory
- get new block from memory, place in cache
- write new data to cache block, set dirty bit

A real cache: Intrinsity FastMATH

[4/2010: Apple bought Intrinsity, designer of A4 CPU core for iPad]

Intrinsity FastMATH:

- fast embedded MIPS-based processor
- 12-stage pipeline
- In same cycle: may request up to 1 instruction word and 1 data word
- Split instruction/data cache
- Each cache: 16KB, 16-word blocks, direct-mapped

Tag, index, offset layout:

p. 469 Fig. ~~5.9~~ 5.12

Cache reads: similar to standard operation

- Must select 1 of 16 words with offset

Writes: support both write-through and writeback

One-entry write buffer

Miss rates:

instruction miss rate = .4%

data miss rate = 11.4%

effective combined miss rate = 3.2%

Tracing cache operations

Given system: 64-byte direct-mapped cache with 16-byte blocks

Code example:

```
float x[10],sum=0.0, prod=1.0;
int i;
        [… some code not shown…]
        for (i=0;i<10;i++)
                sum = sum + x[i];
        for (i=0;i<10;i++)
                prod = prod * x[i];
```

Assumptions: I, sum and prod allocated to registers

&x[0] = 0x10 0018, array not in cache when first for loop starts.

Trace the loops. Count the number of hits and misses.
Compute miss rate.

Same hardware assumptions as before. New loop:

```
for (i=0;i<20;i++)
     sum = sum + x[i];
for (i=0;i<20;i++)
     prod = prod * x[i];
```

Count number of hits and misses. Calculate miss rate.

Example (p. 493):

instruction cache miss rate = 2%

data cache miss rate = 4%, 36% load/stores

CPI without memory stalls = 2 cycles

miss penalty = 100 cycles for all misses

Find 1) actual performance

2) performance with perfect cache (no misses)

3) How much faster is perfect cache?

Handling conflicts: set-associative caches

In direct-mapped caches, block X from memory can only be placed in one block in cache.

Several memory blocks may need same cache block: conflict misses.

Example: instruction cache, 8KB cache, 16-byte blocks
Construct code with 100% miss rate.

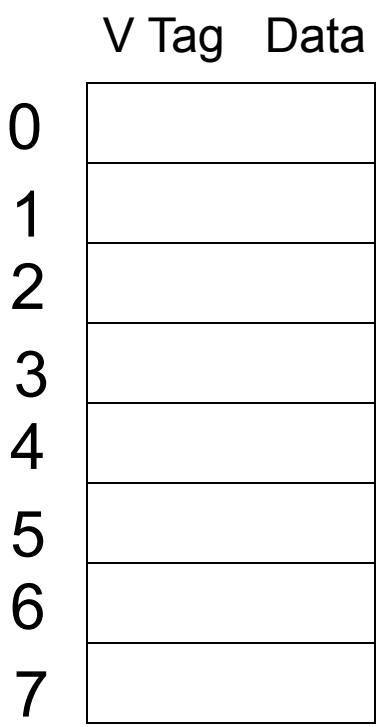
Suppose `main` is at 0x400028.

Cache index =

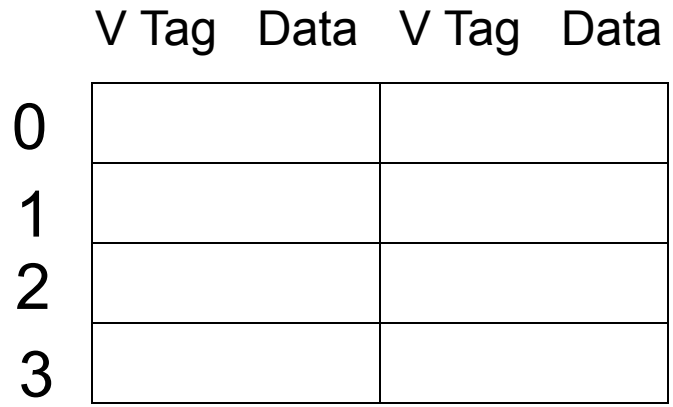
```
main:
```

Cache index	contents
…	
…	

Basic idea: 2-way set-associative cache



Direct-mapped 8-block cache



2-way set-associative cache with 4 sets

In a k-way set-associative cache with N blocks, the N blocks are divided into N/k sets, each with k blocks.

After computing index, block X from memory can be placed in any one of k blocks.

On an access, first compute index, then must check all k blocks, match k tags.

How to compute tag, index, offset? (C-byte cache, B-byte blocks)

offset:

index:

tag:

Earlier example (`main` at 0x400028):

Cache index =

```
main:
```

index	V	tag	data	V	tag	data
…						
…						

4-way set-associative cache design Fig. ~~5.17~~ 5.18:

Performance benefits:
(Intrinsity FastMath data cache, SPEC 2000)

Associativity	**Miss rate**
1 (direct-mapped)	10.3%
2	8.6%
4	8.3%
8	8.1%

Example (p. 486):

Assume cache with 4K blocks, 32-bit address, 4-word blocks.

Find total no. of tag bits for 1) direct-mapped 2) 2-way set-associative 3) 4-way set-associative 4) fully associative

Replacement policy

For k-way set-associative cache, on a miss,

get block from memory
check k cache blocks
if one cache block is empty, place new block there

If all k choices are full, must *replace* a block with new block.

How to decide which block to replace?

Most common: least recently used (LRU)

Others: random, FIFO

Multilevel caches

Cache must be fast (ideally, match CPU clock rate).

To ensure speed, cache must be small.

But small cache means high miss rate.

Solution: 2-level caches

- Level 1 cache: small and fast
- Level 2 cache: larger and slower (but faster than memory)

L1 caches: usually split instruction/data, on-chip, SRAM

L2 cache: usually unified, on or off-chip, SRAM

Virtual memory: basic concepts

Two processes running on a time-shared system:

Process 0:

[code not shown]

```
sw $16, 0($29)
bne $12, $18,…
```

Process 1:

[code not shown]

```
lw $12, 0($29)
sub $20, $20, $12
…
```

Earlier we saw how P0 and P1 *timeshare* the CPU.

Suppose for both P0 and P1, $29 = 0x7fff c6c0.

P0 and P1 should access *different* memory locations with sw and lw, even if the addresses are the same!

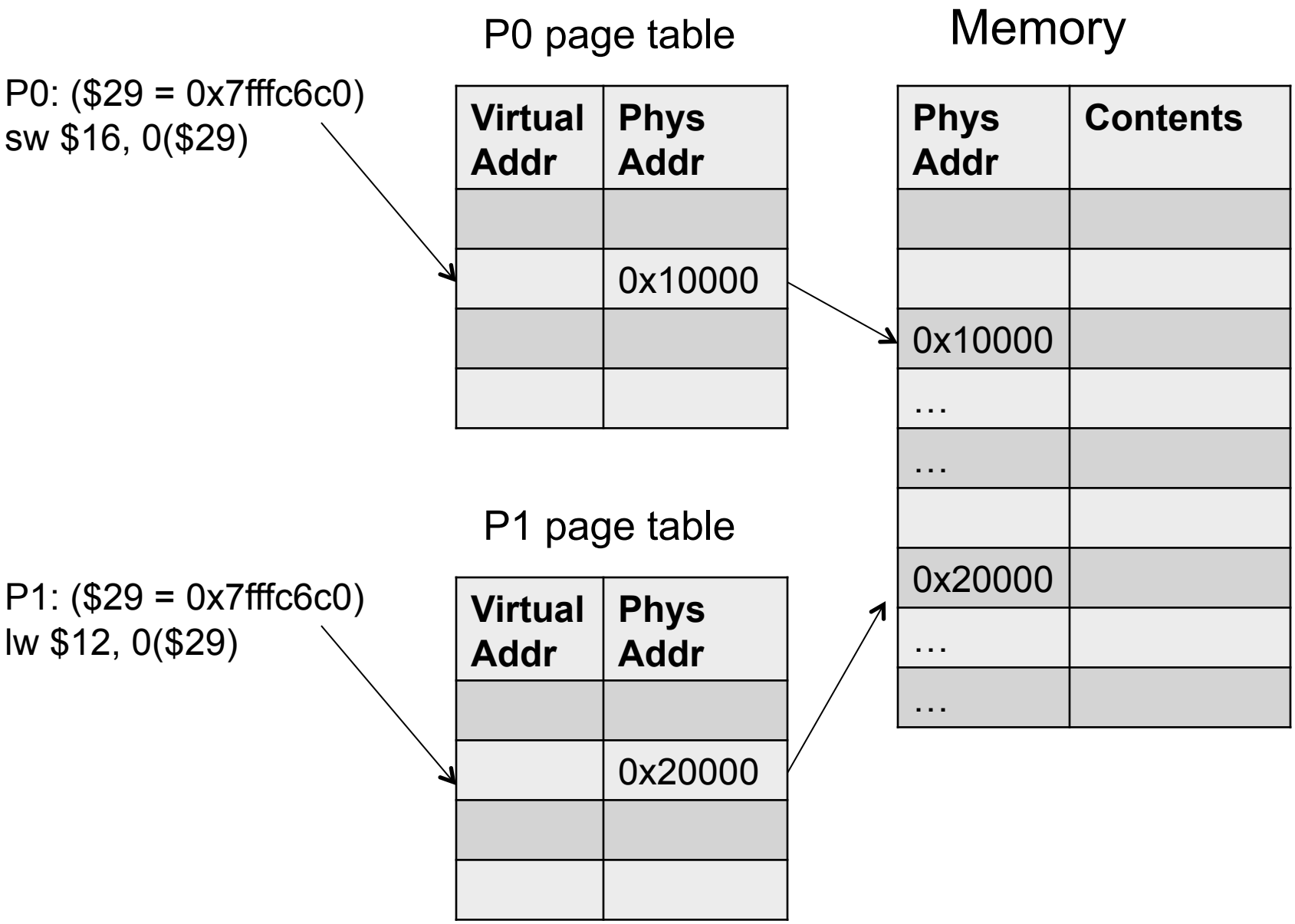
How is this done?

Modern operating systems have *virtual memory*:

- User programs generate *virtual address*
- Virtual address translated to *physical address,* using a *page table*
- Physical address used to access memory

Each user process has own *address space*.

Identical virtual addresses may map to different physical addresses.


P0 page table
Memory
P0: ($29 = 0x7fffc6c0)
sw $16, 0($29)
Virtual Addr
Phys Addr
0x10000
Phys Addr
Contents
0x10000
…
…
0x20000
…
…
P1 page table
P1: ($29 = 0x7fffc6c0)
lw $12, 0($29)
Virtual Addr
Phys Addr
0x20000

Recall: with 32-bit memory address, each process can "see"/address 4GB of memory.

This is called the *address space* of the process.

Memory is organized into *pages:*

- All pages are usually of the same size.
- Each page contains *contiguous* memory locations
- Suppose a page contains P bytes.

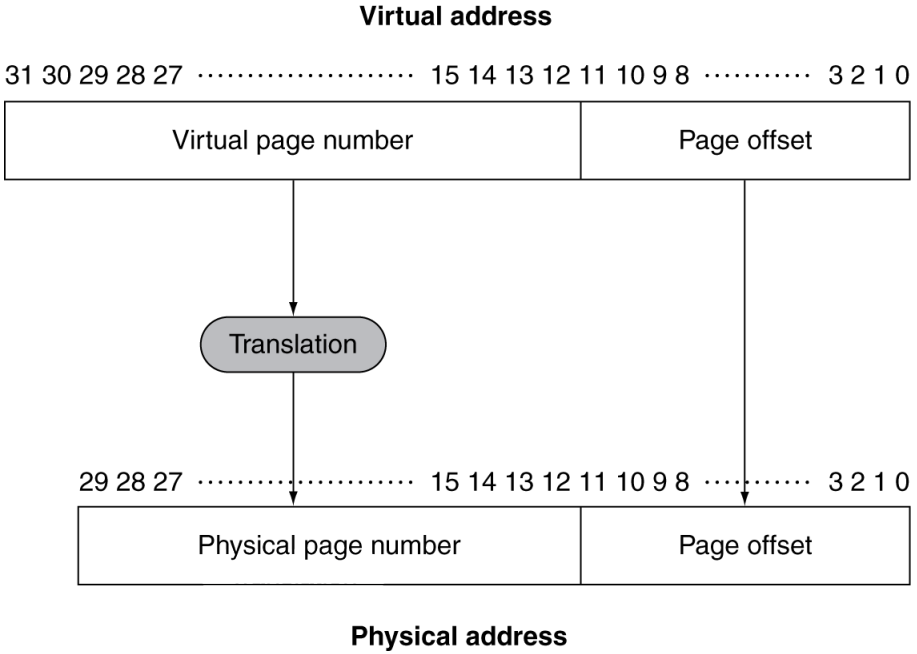
  Page offset: least significant logP bits of address

  Page number: remaining bits of address

Example: 32-bit virtual address, page size = 4KB

0x00000000 to 0x00000ffc are in page 0
0x00001000 to 0x00001ffc are in page 1
0x00002000 to 0x00002ffc are in page 2
0x00003000 to 0x00003ffc are in page 3 etc

Translating from a virtual address to a physical address:



Note:

The page table is *indexed* with the virtual page number.

Each page table entry contains a physical page number.

The *page offset* (position of the address within a page) is the same for the virtual address and the physical address.

With 32-bit memory addresses, each process has a 4 GB address space.

If we have 10 processes, they can address 40 GB!

But only a small fraction of a process' address space may actually be in main memory at a time.

Usually,

- A process has < 1 GB of its address space mapped to physical memory
- Unmapped parts of address space are (usually) on disk
- A page table entry indicates whether a page is mapped (in physical memory), or on disk Fig. ~~5.22~~ 5.28

A *miss* in virtual memory is called a *page fault*:

- Page is on disk, not in physical memory

Page faults are *very expensive* (millions of cycles!)

- Always handled in software
- Top priority: reduce page fault rate

Page sizes:

- Typical 4 to 16KB
- Newer systems 32 to 64 KB
- Some embedded systems 1KB

Write policy: always write-back!

Where are page tables?

- Also in memory

Too slow!

Translation Lookaside Buffer (TLB):

- Cache for page table entries
- Usually one instruction TLB, one data TLB

Virtual memory and protection

Basic ideas:

Each process has own virtual address space (own page table)

Each page table entry has valid, dirty, write-control bits

User process should not be able to change own page table

More in CSc 415, CSc 656

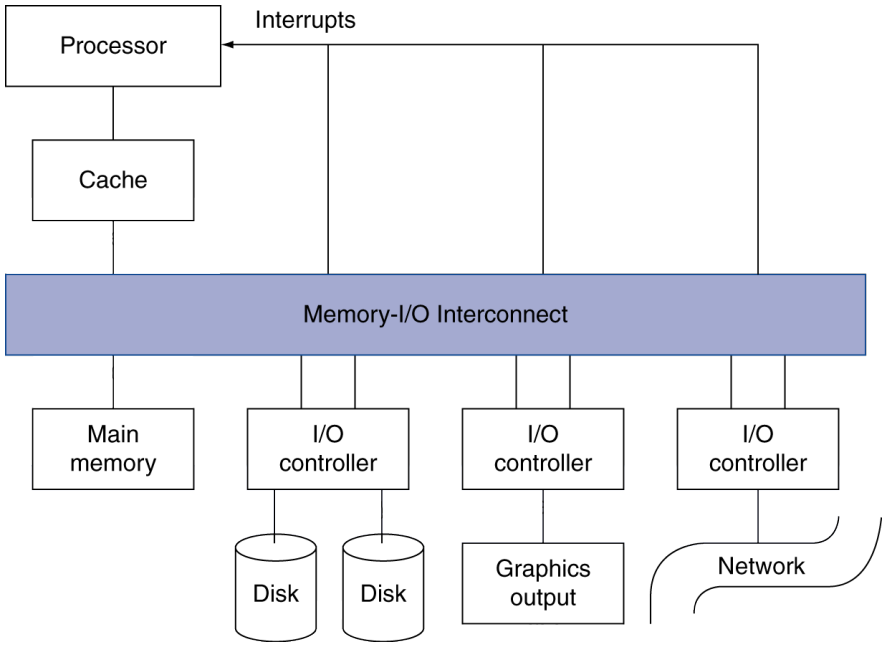
Intel Nehalem & AMD Opteron X4 memory systems

	Intel Nehalem	AMD Opteron X4
L1 caches (per core)	L1 I-cache: 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/ allocate, hit time n/a	L1 I-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles L1 D-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, write-back/ allocate, hit time 9 cycles
L2 unified cache (per core)	256KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	512KB, 64-byte blocks, 16-way, approx LRU replacement, write-back/allocate, hit time n/a
L3 unified cache (shared)	8MB, 64-byte blocks, 16-way, replacement n/a, write-back/ allocate, hit time n/a	2MB, 64-byte blocks, 32-way, replace block shared by fewest cores, write-back/allocate, hit time 32 cycles

Input/output basics

I/O devices are diverse:

- Input only, output only, or storage (input/output)
- Human partner or machine partner
- Data rate



Common I/O devices Fig. 6.2 p. 571

Some examples:

Device	Behavior	Partner	Data rate (Mbit/s)
Keyboard	Input	Human	.0001
Mouse	Input	Human	.0038
Sound out	Output	Human	8
Graphics monitor	Output	Human	3.2
Network/LAN	Input/output	Machine	100k – 1000k
Network/Wifi	Input/output	Machine	11k – 54k
Magnetic disk	Storage	Machine	800-3000

Note: I/O devices *much slower* than CPU

Dependability is important, especially for storage.

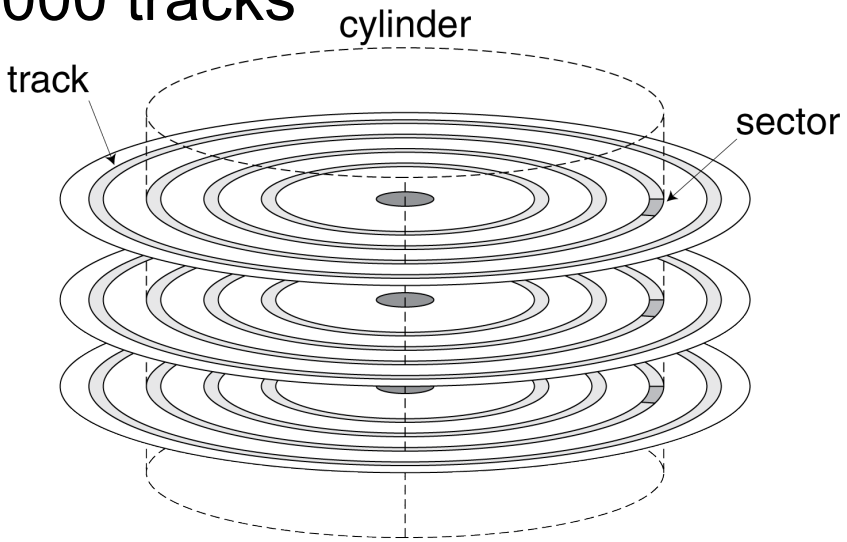
Performance measures:

- Throughput
  - More important in servers
- Latency or response time
  - more important in desktops and laptops

Disks

Characteristics:

- Non-volatile
- Each disk: one or more platters
- Rotate at 5400 to 15000 rpm
- Each surface: 10,000 to 50,000 tracks



- Each track: 100-500 sectors
- Each sector: typically 512 bytes (4096 proposed)
- Older disks: all tracks have same # bits
- Zone bit recording (ZBR): outer tracks have more bits

Read/write heads must be over correct sector for access.

Seek: move head over correct track

Seek time: time taken to seek to correct track

Average seek times: 3 ms to 14 ms

Actual seek time may be < 30% of average

locality of disk accesses

After head moved over correct track:

- Wait for correct sector to rotate under head

Average rotational latency: time to rotate halfway around disk

= 0.5 / (rotational speed in rpm / 60)

For 5400 rpm: rotational latency = 5.6ms

For 15000 rpm: rotational latency = 2 ms

Next: transfer data to/from sector

Transfer time: depends on sector size, rotation speed, recording density

2008 transfer rates from sector: 100 MB/sec

Sector may be in *disk cache*; cached transfer rates up to 320 MB/sec

Disk controller handles control of disk and transfer between disk and memory.

4 components of disk access time:

seek time + rotational latency + transfer time + controller time

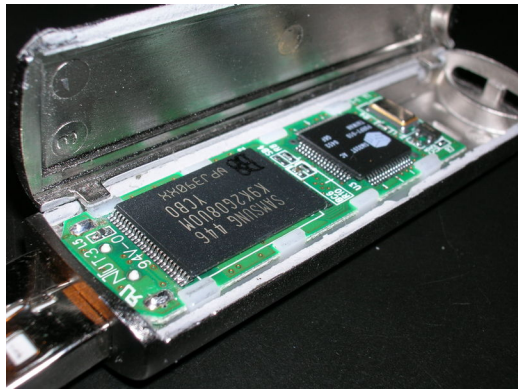
Example: calculate average time to read/write 512-byte sector

Rotational rate = 15,000 rpm, average seek time = 4 ms, transfer rate = 100 MB/sec, controller overhead = 0.2 ms

Flash storage

Characteristics:

- Non-volatile (semi-conductor based)
- 10 – 100 times faster than disk
- Small, low power, more robust, but more expensive



Buses

Basic concepts:

- single set of shared wires
  1) control lines (requests and acknowledgements)
  2) data lines (data, addresses etc)
- only one device can put info on wire at a time
- all devices can read

Input: from device to memory

Output: from memory to device

Typical bus transaction:

- Send address
- Read or write data
- (more steps for control!)

Types of buses:

- Processor-memory buses (short and fast)
- I/O buses (longer and slower)
  - Connect variety of devices
- Special purpose buses (ex. Graphics)

I/O buses usually standardized.

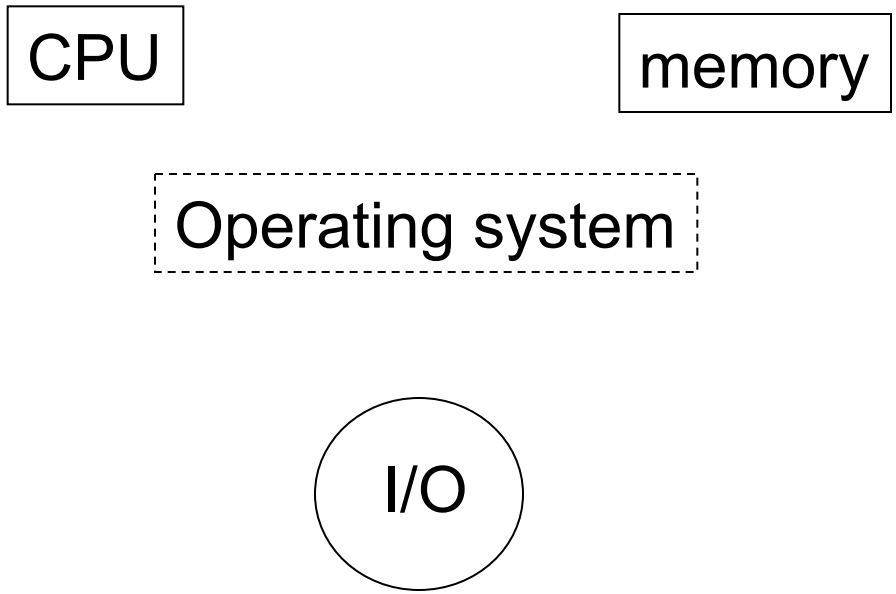
(Processor-memory buses are not…)

Example: Firewire vs. USB

Characteristic	Firewire (1394)	USB 2.0
Data bus width	4	2
Clocking	Asynchronous	Asynchronous
Peak bandwidth	50MB/s or 100MB/s	0.2 MB/s, 1.5 MB/s, or 60 MB/s
Max # devices	63	127
Max length	4.5m	5m

Typical x86 system organization Fig 6.9

I/O interfacing: CPU, memory, OS



C++: `cin >> x` or `cout << x`

How does data move between memory and I/O devices?

Some basic questions:

- In MIPS: I/O goes through syscall instruction
  - Below syscall: what is actual I/O operation? How to communicate with I/O device?
- How is data moved between memory and I/O devices?

I/O operations go through operating system:

- Multiple processes use same I/O devices

  OS isolates processes from each other, provides protection

  Ex.: process should not be allowed to read/write a file unless it has proper permissions

- I/O often uses interrupts

  system interrupt handler is part of OS
- I/O control is complex and device-specific

  low level details should be hidden from users

  OS provides high level abstractions for I/O ops

Communication between OS and I/O devices:

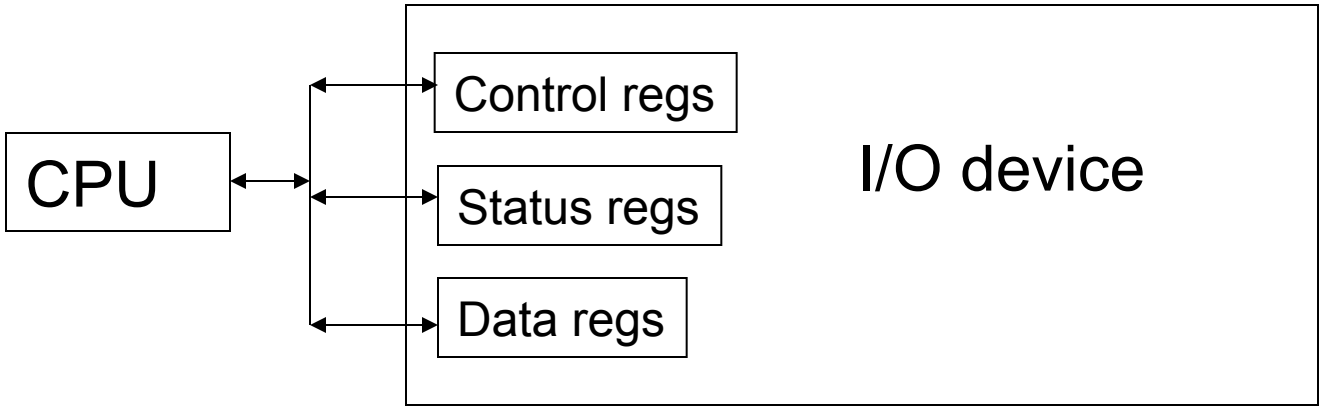
- How does OS send commands to I/O devices?
- How does device notify OS of status? (Ready, done, error, etc)
- How is data transferred between memory and device?

Code for controlling I/O operations

An I/O device usually has registers for controlling access:

- control registers for commands (read, write etc)
- status registers (data ready, full/empty etc)
- data registers

CPU is able to read and write registers in I/O device.



What do I/O registers look like to CPU?

What instructions are used to access I/O registers?

Option 1 (less common): dedicated I/O instructions

- I/O registers located at numbered ports
- Special instructions work with I/O registers
  - Write a command into a control register
  - Read a status register
  - Read/write a data register
- Special instructions only available in kernel mode

Option 2: memory-mapped I/O

- I/O registers “located” at pre-defined memory addresses (only accessible to OS)
- Load/store instructions access I/O registers!

  Write a command into a control register: sw

  Read a status register: lw

  Read/write a data register: lw/sw

Simple printer example:

- Status register (*done* bit, *error* bit)
- Data register (1 byte)

CPU writes character to data register

Wait for done bit to be set; write next character

Interaction between CPU and I/O

CPU generally much faster than I/O device.
How to coordinate interaction?

3 main options:

- Polling
- Interrupt-driven I/O
- Direct memory access (DMA)

Polling: CPU actively checks I/O device.

Output (printer) example:

- CPU writes character to printer data register
- Wait for printer to finish printing (check done bit)
- Write next character, etc

Input (mouse) example:

- CPU checks if mouse moved (check status reg)
- If mouse moved, get movement data, update cursor
- Keep checking for mouse to move again, etc

CPU is much faster than most I/O devices.

Many cycles wasted!

Problem with polling (for input devices):

CPU *must* assume device is *always* transmitting; otherwise data may be lost.

I/O operations are usually quite rare! Much overhead.

Better approach: interrupt driven I/O

- CPU sends command to device; moves on to other tasks
- Device handles transfer; when done, interrupts CPU
- CPU handles transfer, sends next command, etc

Interrupt-driven I/O is nice for small transfers.

But: still needs one interrupt per transfer.

Inefficient for big block transfers (like paging).

Third approach: Direct memory access (DMA)

Need: DMA controller (simple microprocessor) on bus

- CPU sends info to DMA controller to setup transfer
- DMA handles transfer between device and memory (uses bus when CPU is not using it) (CPU does other tasks)
- When transfer finished, DMA controller interrupts CPU

Summary

Topics covered in this chapter:

- Memory hierarchy concepts
- Caches
  - direct-mapped caches
  - set-associative caches
  - write policy
- Virtual memory basics
- Input/output basics