RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

ECE-332:437
DIGITAL SYSTEMS DESIGN (DSD)

Fall 2016 – Lecture 13
Memory I/O Systems, Cache

Nagi Naganathan
December 8, 2016

Topics to cover today – December 8, 2016

- How was Mid Term 2?

- Lecture 13 – Memory I/O Systems, Cache
- Lecture 14 – Review

Announcements – December 8, 2016

- Finals – December 20, 2016 – 8.00 – 11.00 pm
  - Cumulative – All chapters and lectures including some lab questions
- Covered Materials
  - Chapters 1-8 from the text book
  - Lecture Slides 1-14 including the slides on ARM
  - Major Topics
    - Combinational Logic
    - Sequential Logic
    - HDL – SystemVerilog
    - Digital Building Blocks – Complicated Adders, Multipliers, Division not needed – Just the basics
    - Architecture – ISA, MIPS, ARM
    - Microarchitecture – MIPS, ARM
    - Cache – Memory – Serial I/O (I2C, I2S etc., not needed)

Announcements – December 8, 2016

- Please fill out the SIRS Course survey -
  - Deadline Dec 16, 2016
  - https://sakai.Rutgers.edu/portal/site/sirs

Grading

Home Works	- 10%
Quizzes	- 5%
Labs	- 30 %
Mid Term 1	- 15%
Mid Term 2	- 15%
Finals (Cumulative)	- 25%
Total	- 100%

MEMORY & I/O SYSTEMS

Chapter 8

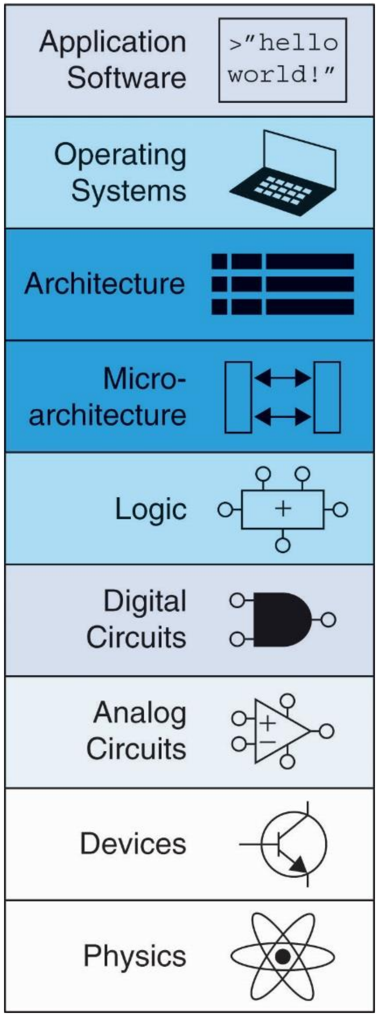
***Digital Design and Computer Architecture*, 2nd Edition**

David Money Harris and Sarah L. Harris

ELSEVIER

Chapter 8 :: Topics

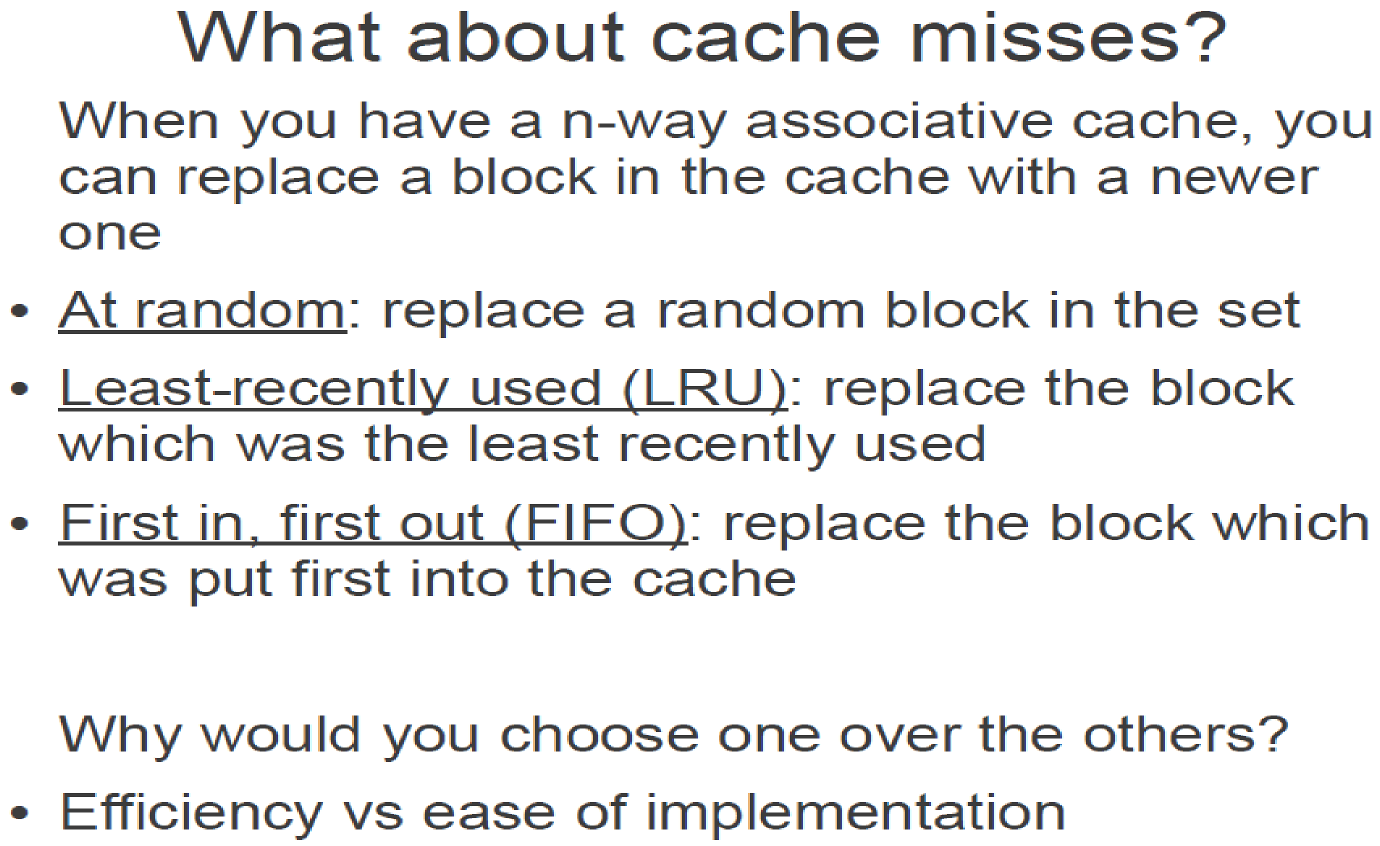
- Introduction
- Memory System Performance Analysis
- Caches
- Virtual Memory
- Memory-Mapped I/O
- Summary



Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access

Adapted from misc sources



What about cache misses?

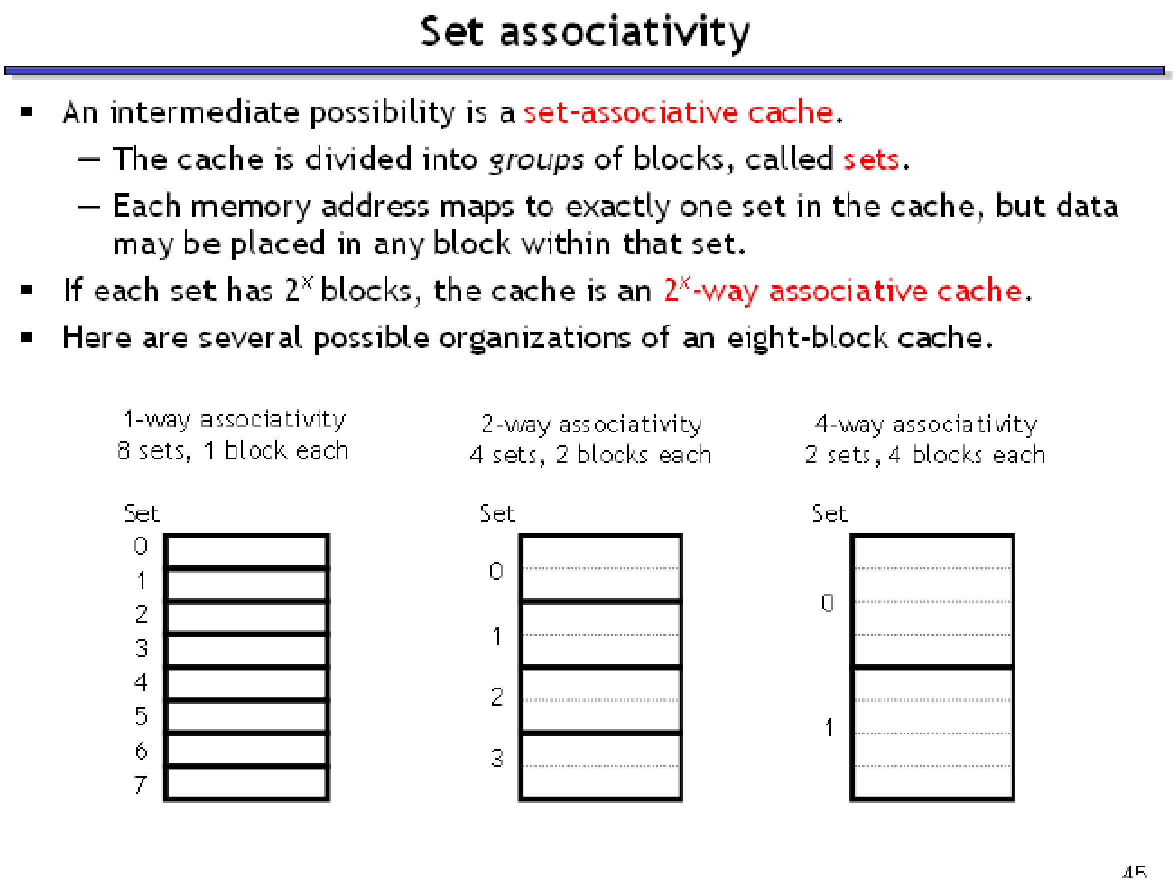
When you have a n-way associative cache, you can replace a block in the cache with a newer one

- At random: replace a random block in the set
- Least-recently used (LRU): replace the block which was the least recently used
- First in, first out (FIFO): replace the block which was put first into the cache

Why would you choose one over the others?

- Efficiency vs ease of implementation

Adapted from misc sources



Set associativity

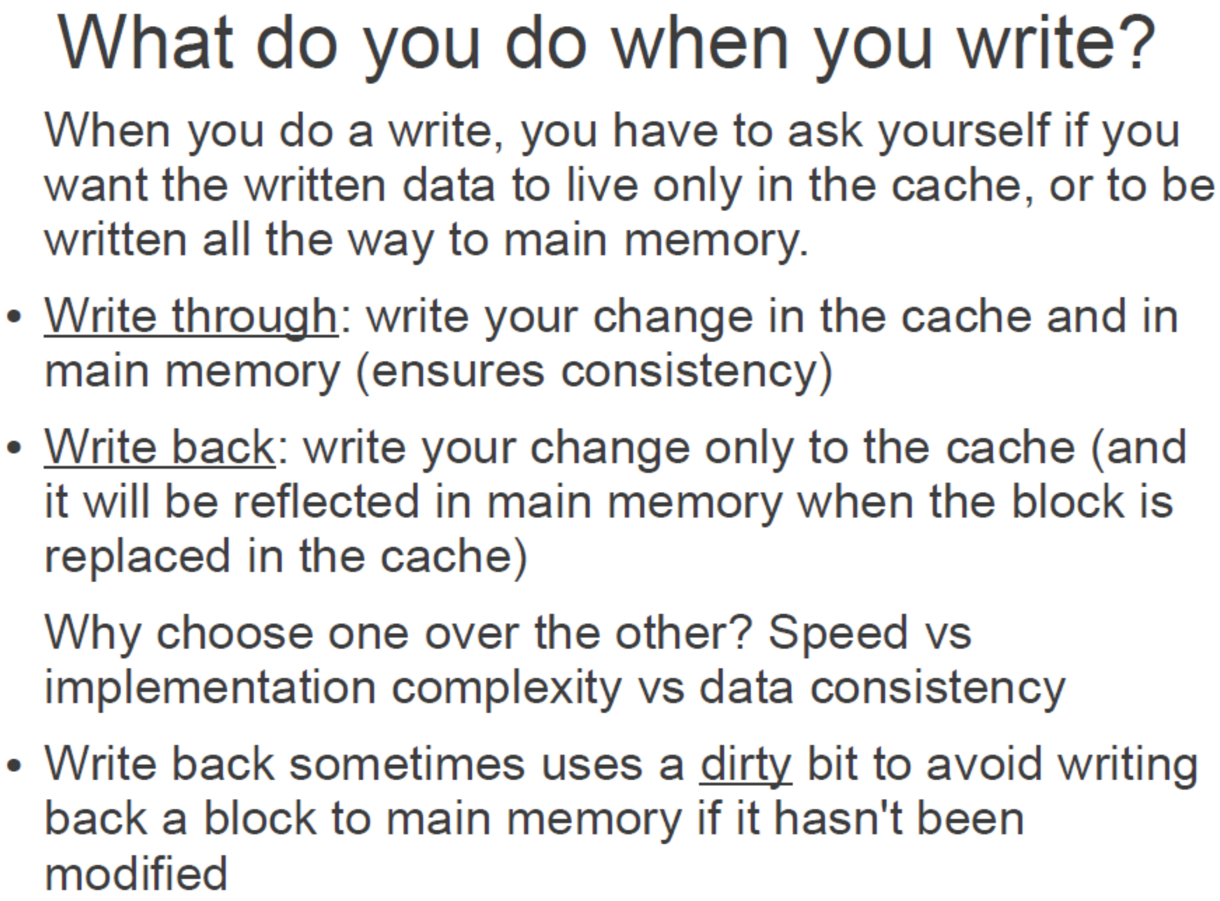
- An intermediate possibility is a set-associative cache.
  - The cache is divided into *groups* of blocks, called sets.
  - Each memory address maps to exactly one set in the cache, but data may be placed in any block within that set.
- If each set has 2^x blocks, the cache is an 2^x-way associative cache.
- Here are several possible organizations of an eight-block cache.

1-way associativity
8 sets, 1 block each

2-way associativity
4 sets, 2 blocks each

4-way associativity
2 sets, 4 blocks each

Adapted from misc sources



What do you do when you write?

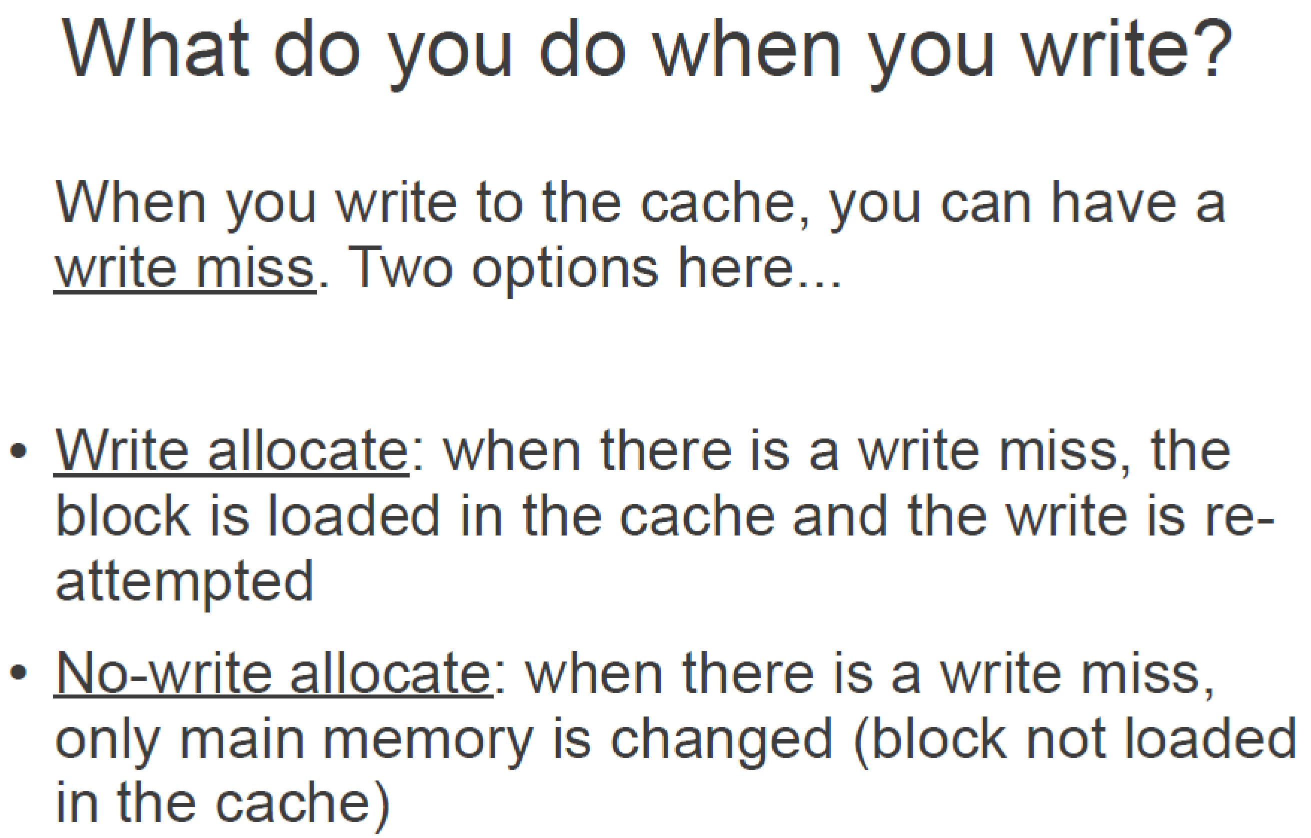
When you do a write, you have to ask yourself if you want the written data to live only in the cache, or to be written all the way to main memory.

- Write through: write your change in the cache and in main memory (ensures consistency)
- Write back: write your change only to the cache (and it will be reflected in main memory when the block is replaced in the cache)

Why choose one over the other? Speed vs implementation complexity vs data consistency

- Write back sometimes uses a dirty bit to avoid writing back a block to main memory if it hasn't been modified

Adapted from misc sources

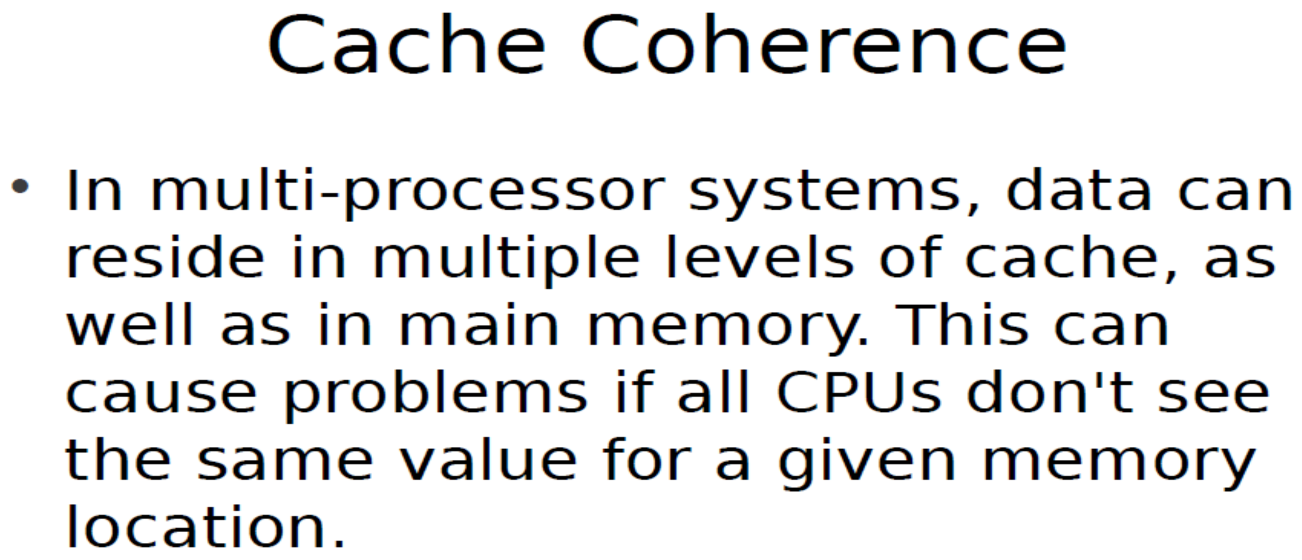


What do you do when you write?

When you write to the cache, you can have a write miss. Two options here...

- Write allocate: when there is a write miss, the block is loaded in the cache and the write is re-attempted
- No-write allocate: when there is a write miss, only main memory is changed (block not loaded in the cache)

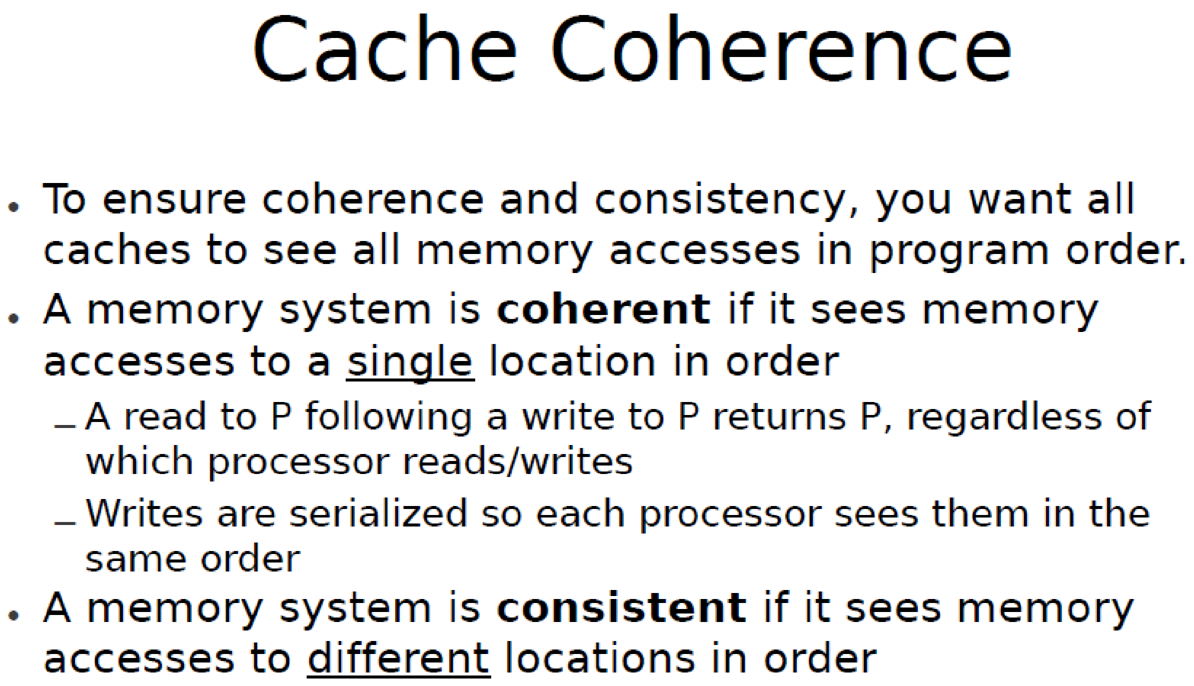
Adapted from misc sources



Cache Coherence

- In multi-processor systems, data can reside in multiple levels of cache, as well as in main memory. This can cause problems if all CPUs don't see the same value for a given memory location.

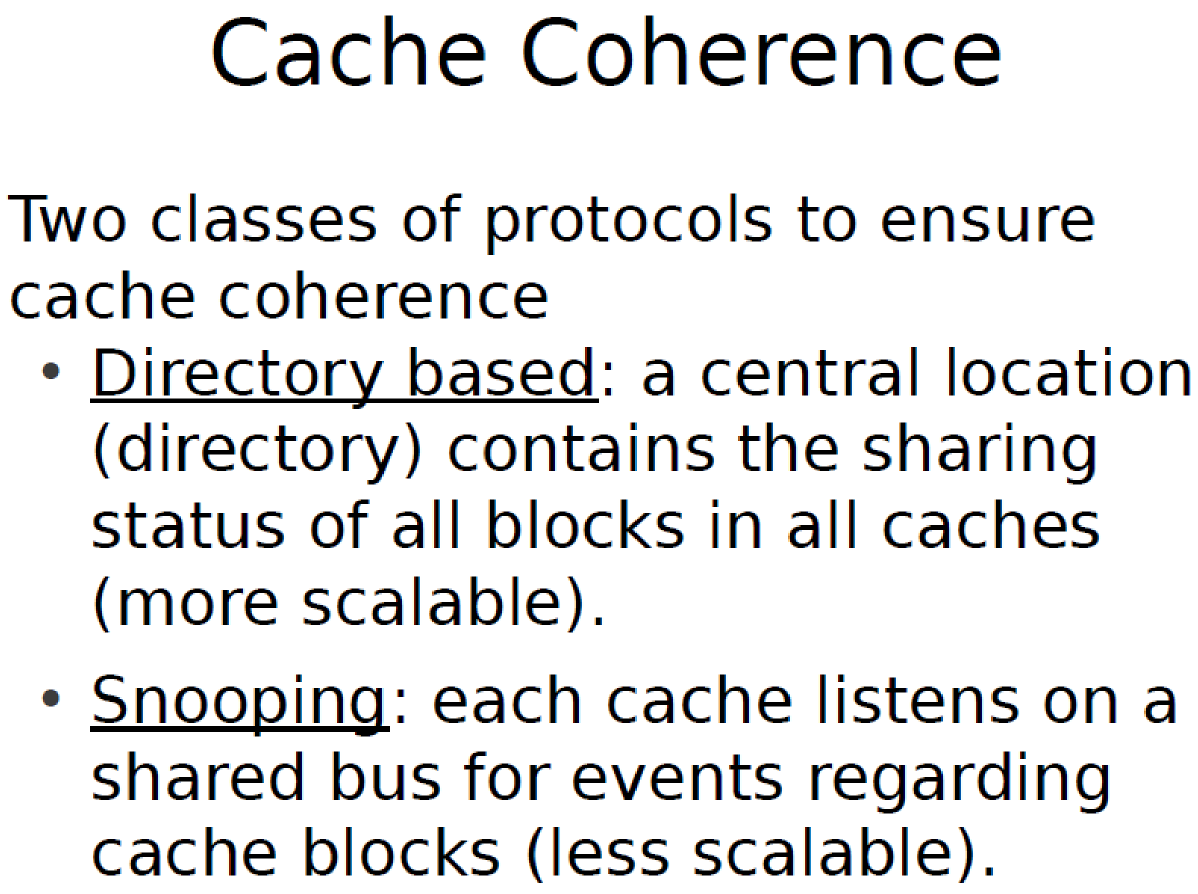
Adapted from misc sources



Cache Coherence

- To ensure coherence and consistency, you want all caches to see all memory accesses in program order.
- A memory system is **coherent** if it sees memory accesses to a single location in order
  - A read to P following a write to P returns P, regardless of which processor reads/writes
  - Writes are serialized so each processor sees them in the same order
- A memory system is **consistent** if it sees memory accesses to different locations in order

Adapted from misc sources

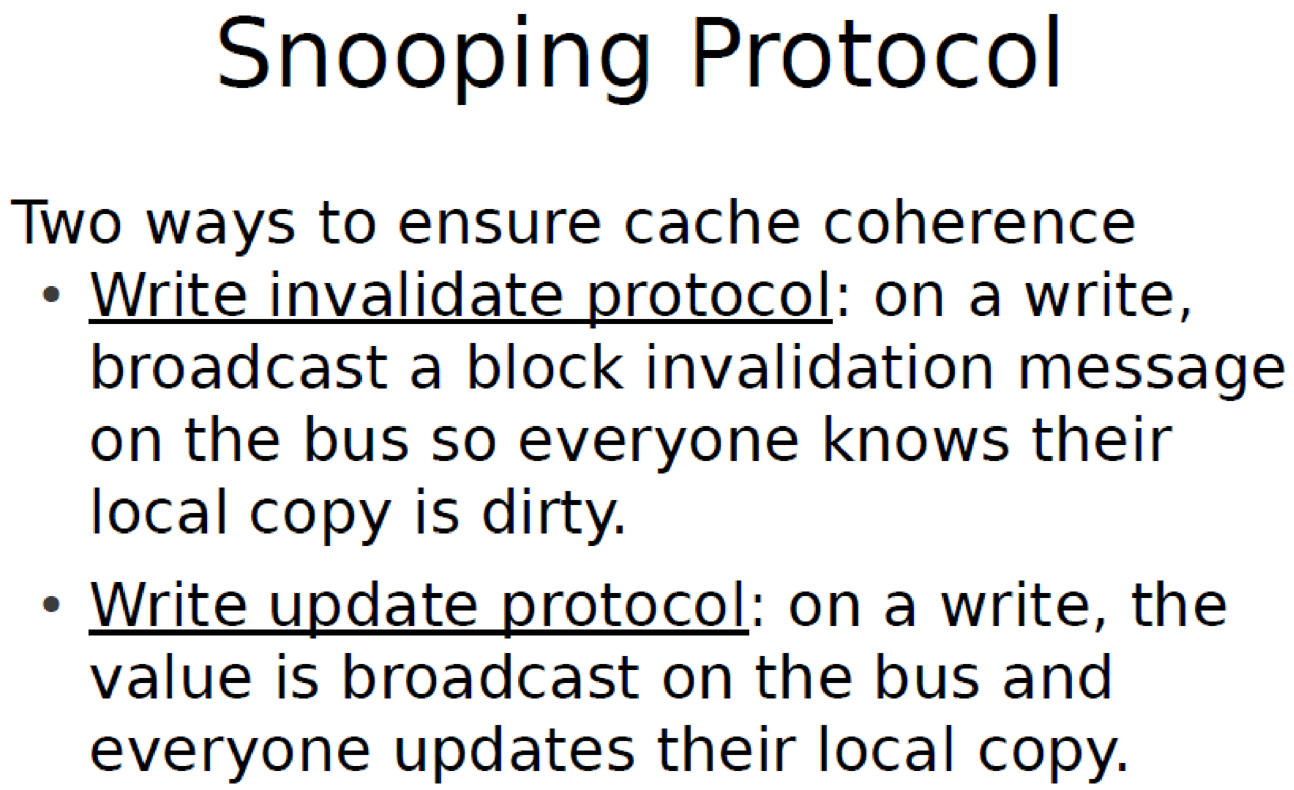


Cache Coherence

Two classes of protocols to ensure cache coherence

- Directory based: a central location (directory) contains the sharing status of all blocks in all caches (more scalable).
- Snooping: each cache listens on a shared bus for events regarding cache blocks (less scalable).

Adapted from misc sources

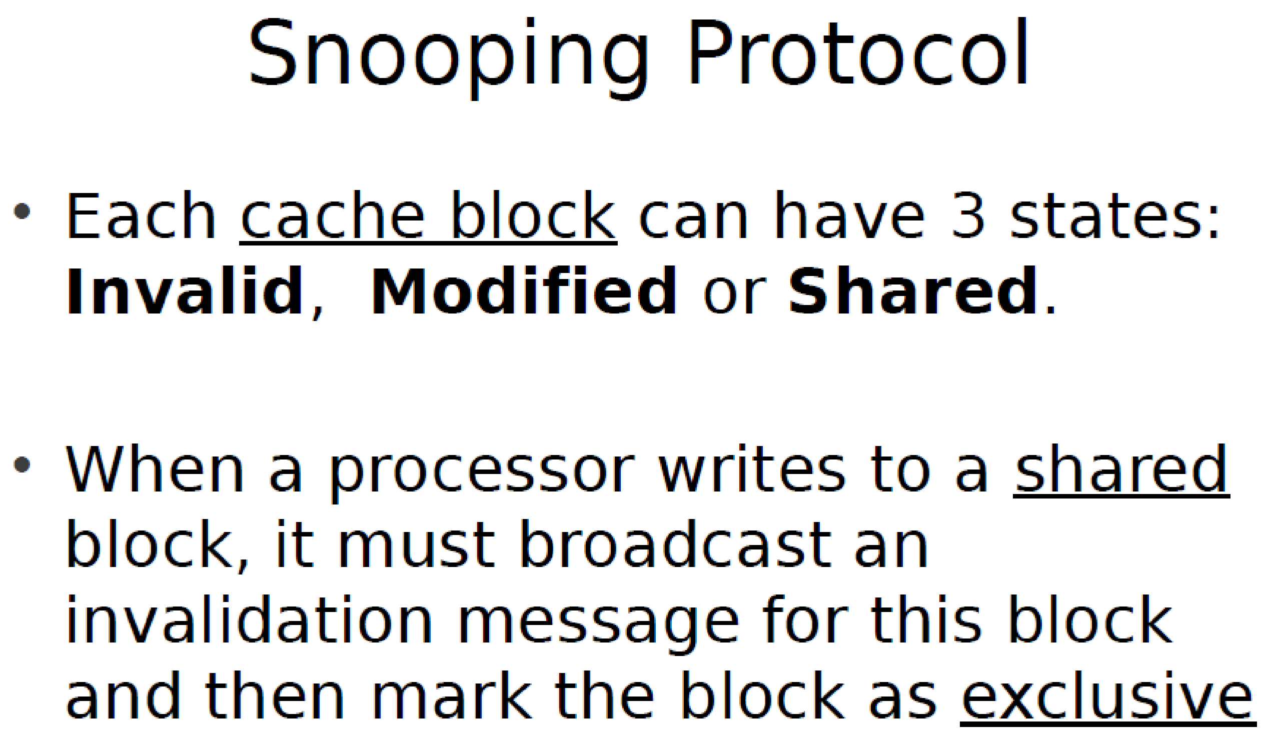


Snooping Protocol

Two ways to ensure cache coherence

- Write invalidate protocol: on a write, broadcast a block invalidation message on the bus so everyone knows their local copy is dirty.
- Write update protocol: on a write, the value is broadcast on the bus and everyone updates their local copy.

Adapted from misc sources

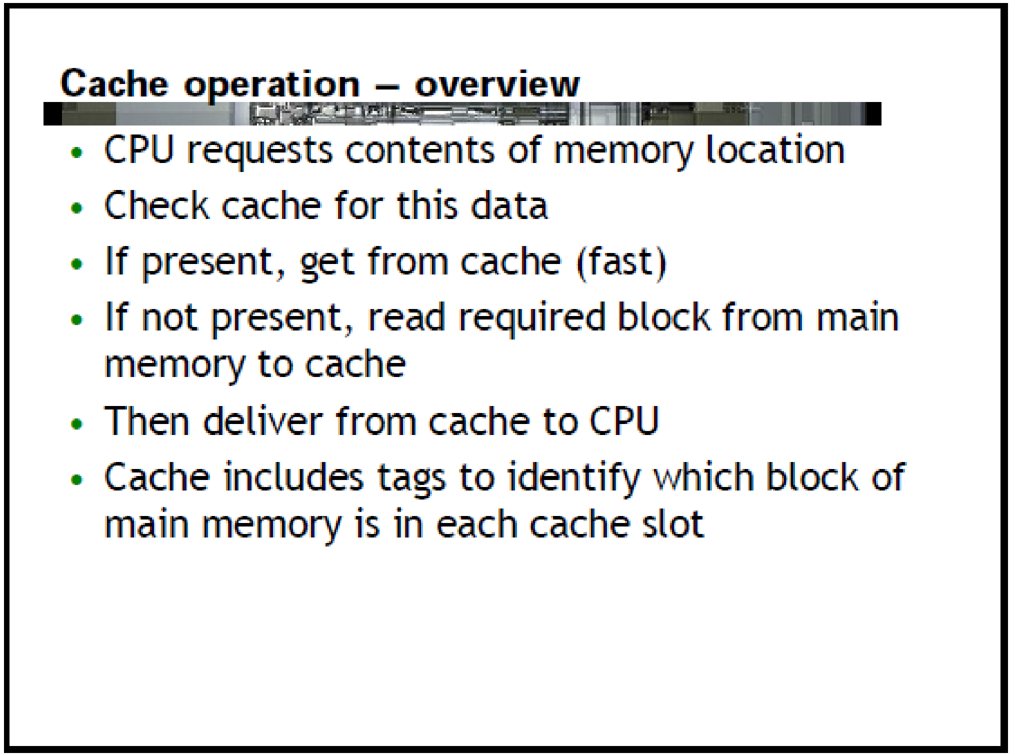


Snooping Protocol

- Each <u>cache block</u> can have 3 states: **Invalid**, **Modified** or **Shared**.

- When a processor writes to a <u>shared</u> block, it must broadcast an invalidation message for this block and then mark the block as <u>exclusive</u>

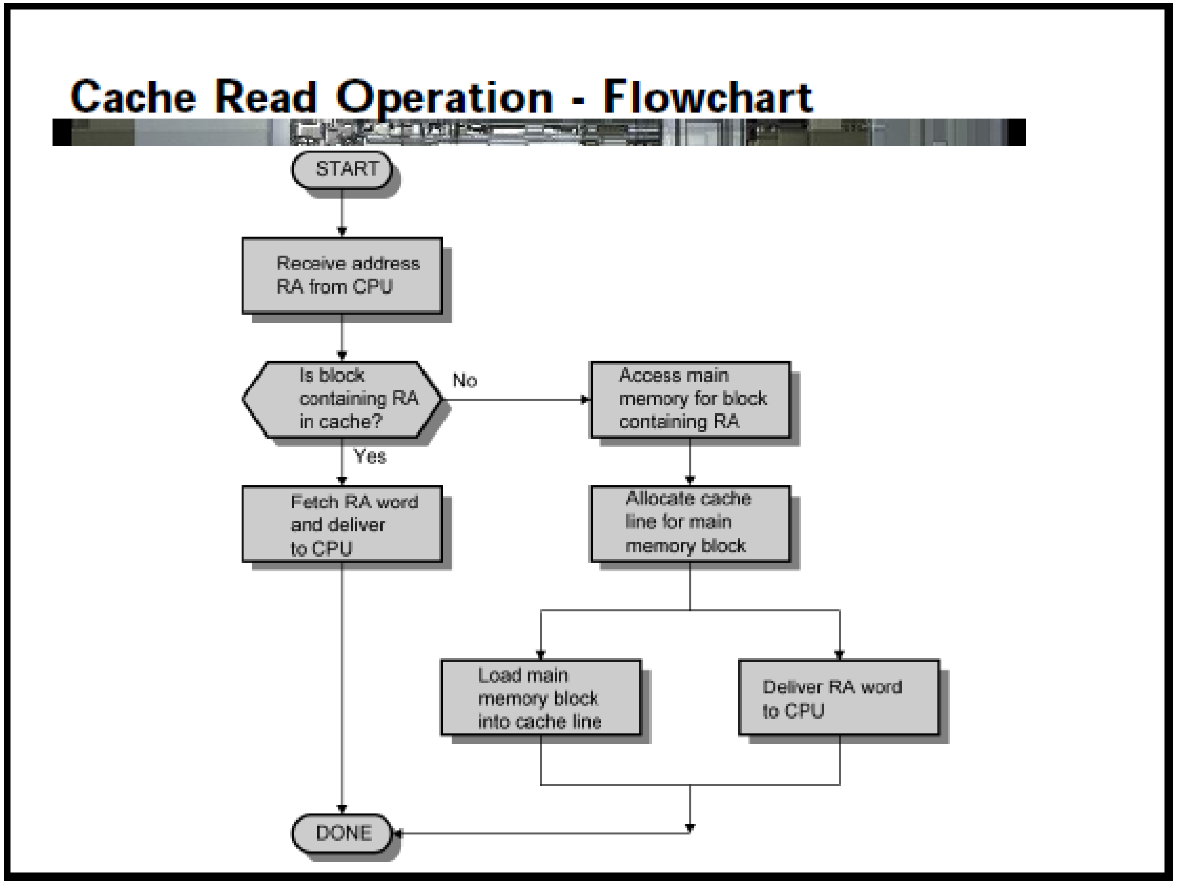
Adapted from misc sources



Cache operation – overview

- CPU requests contents of memory location
- Check cache for this data
- If present, get from cache (fast)
- If not present, read required block from main memory to cache
- Then deliver from cache to CPU
- Cache includes tags to identify which block of main memory is in each cache slot

Adapted from misc sources



Cache Read Operation - Flowchart

START

Receive address RA from CPU

Is block containing RA in cache?

No

Access main memory for block containing RA

Allocate cache line for main memory block

Load main memory block into cache line

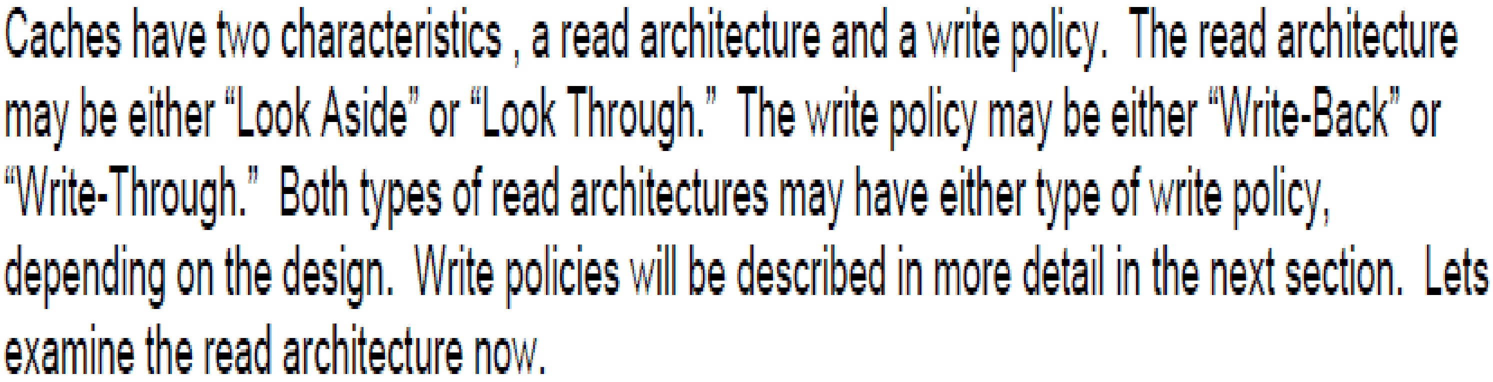
Deliver RA word to CPU

Yes

Fetch RA word and deliver to CPU

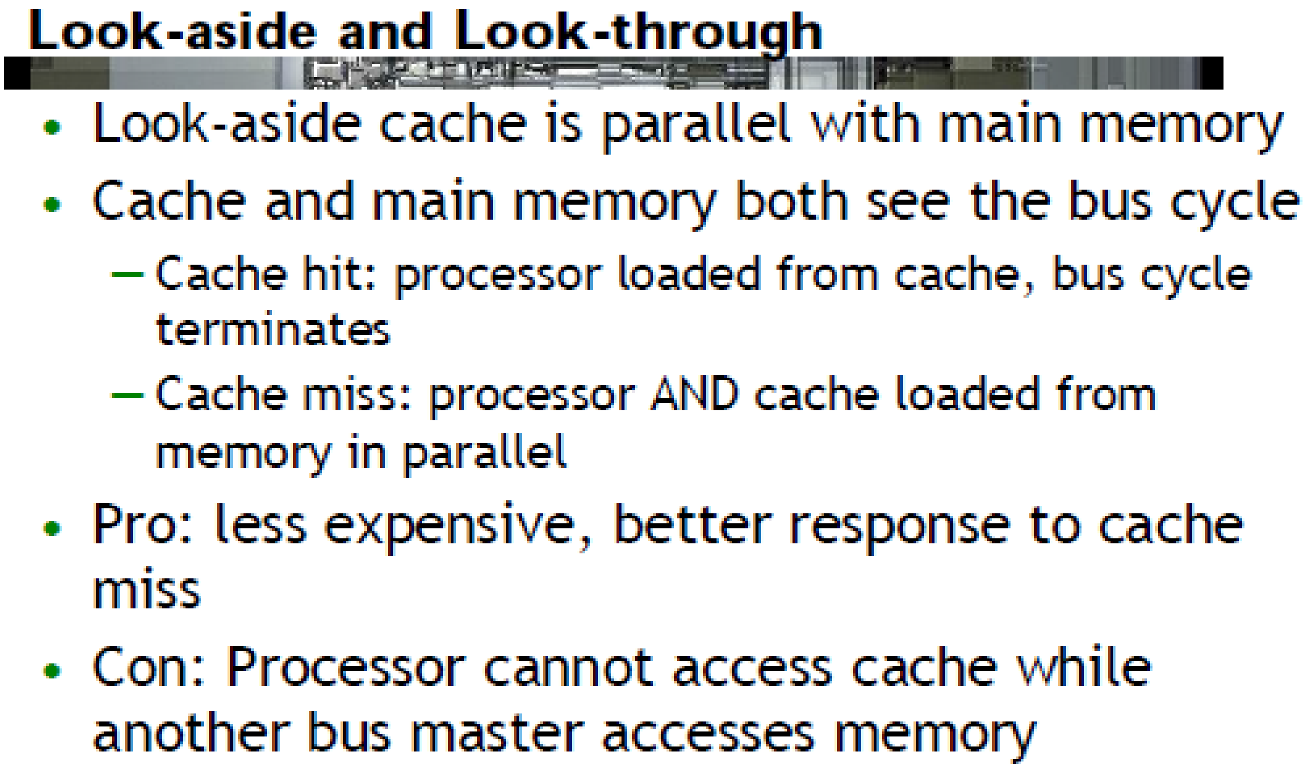
DONE

Adapted from misc sources



Caches have two characteristics , a read architecture and a write policy. The read architecture may be either “Look Aside” or “Look Through.” The write policy may be either “Write-Back” or “Write-Through.” Both types of read architectures may have either type of write policy, depending on the design. Write policies will be described in more detail in the next section. Lets examine the read architecture now.

Adapted from misc sources



Look-aside and Look-through

- Look-aside cache is parallel with main memory
- Cache and main memory both see the bus cycle
  - Cache hit: processor loaded from cache, bus cycle terminates
  - Cache miss: processor AND cache loaded from memory in parallel
- Pro: less expensive, better response to cache miss
- Con: Processor cannot access cache while another bus master accesses memory

Adapted from misc sources

Look Aside

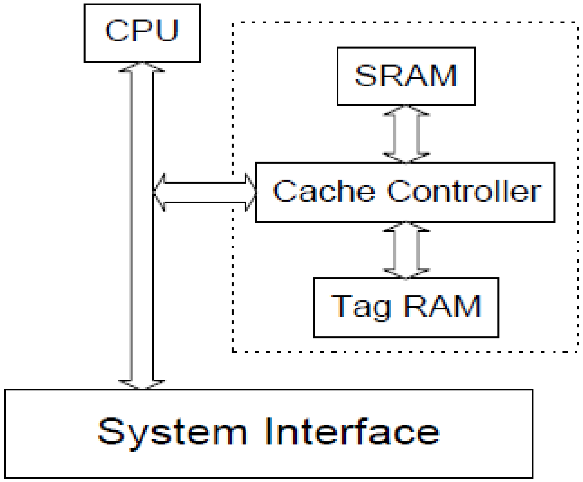
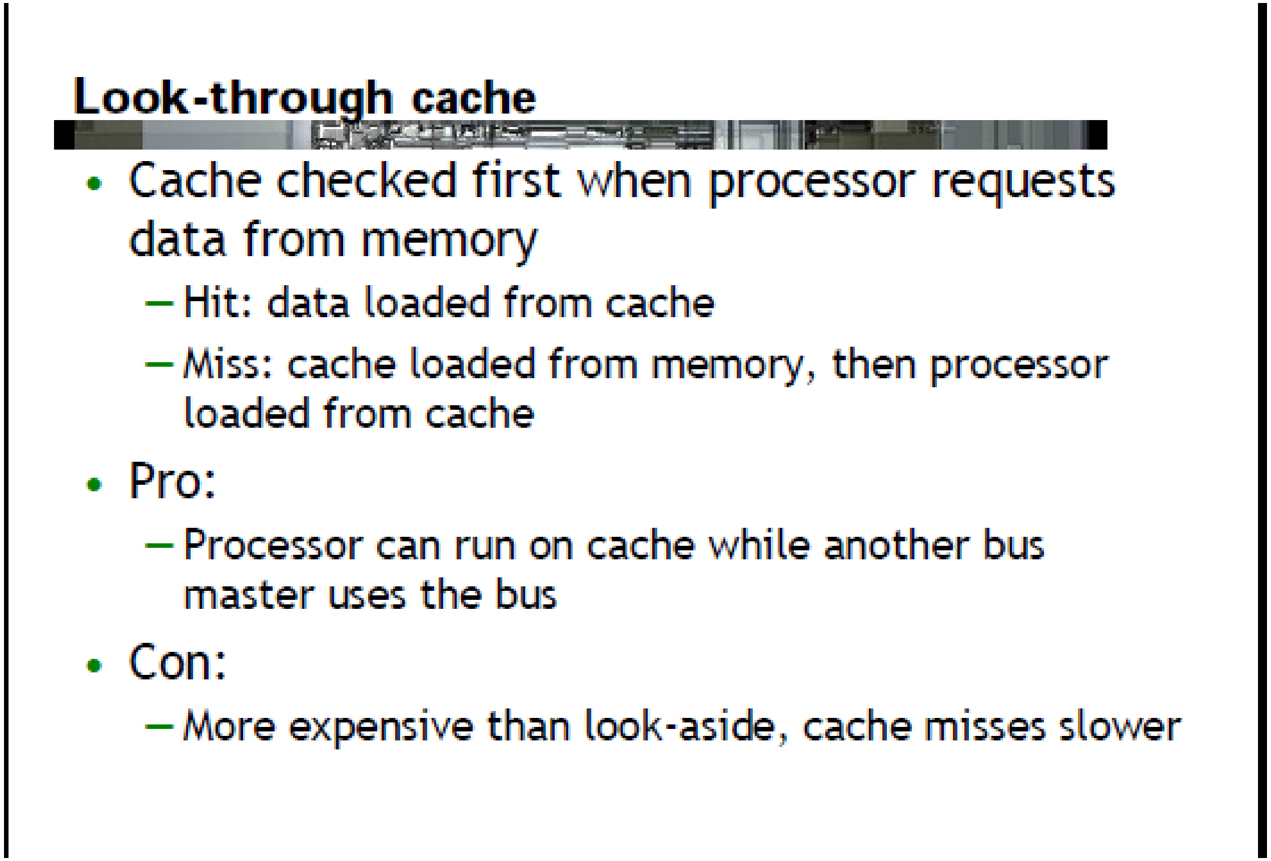


Figure 2-2 Look Aside Cache

Adapted from misc sources



Look-through cache

- Cache checked first when processor requests data from memory
  - Hit: data loaded from cache
  - Miss: cache loaded from memory, then processor loaded from cache
- Pro:
  - Processor can run on cache while another bus master uses the bus
- Con:
  - More expensive than look-aside, cache misses slower

Adapted from misc sources

Read Architecture: Look Through

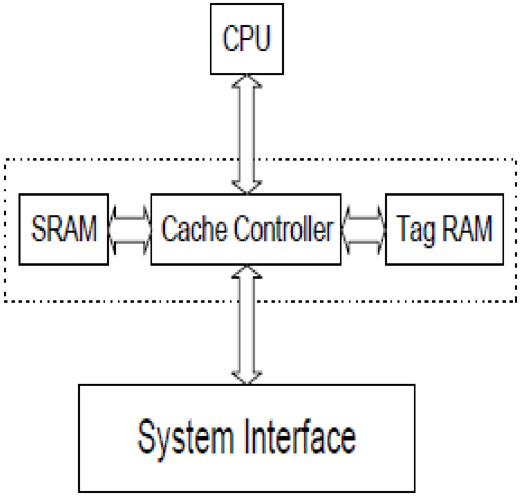
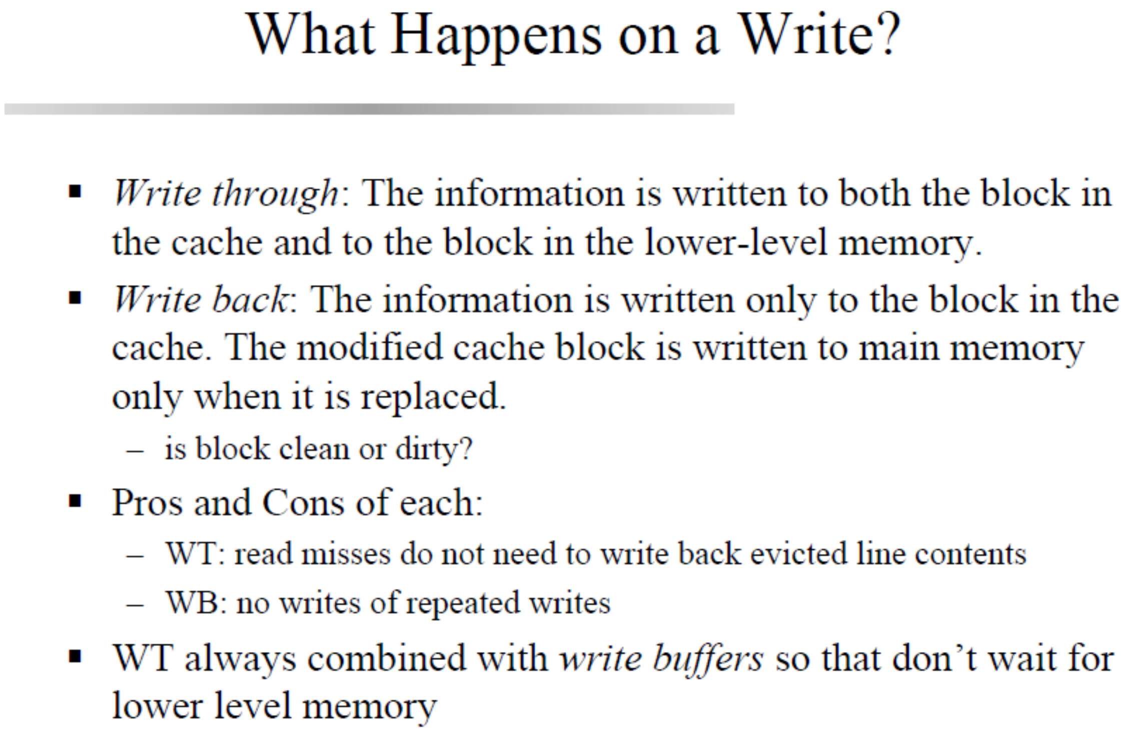


Figure 2-3 Look Through Cache

Adapted from misc sources



What Happens on a Write?

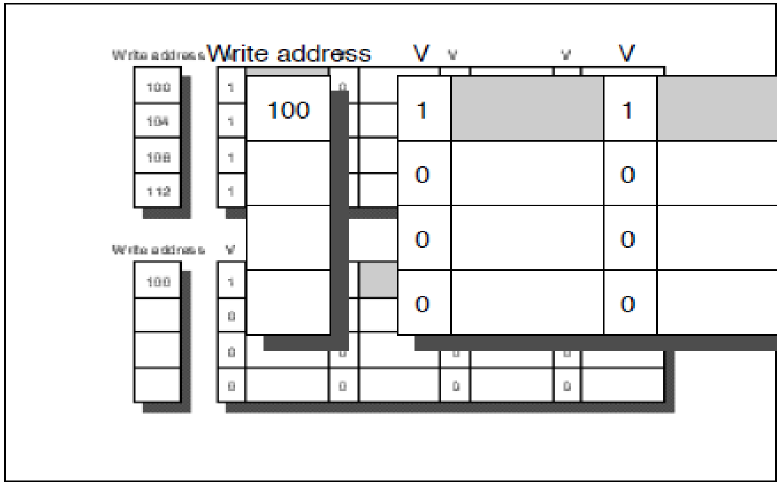
- *Write through*: The information is written to both the block in the cache and to the block in the lower-level memory.
- *Write back*: The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?
- Pros and Cons of each:
  - WT: read misses do not need to write back evicted line contents
  - WB: no writes of repeated writes
- WT always combined with *write buffers* so that don't wait for lower level memory

Write-Through

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also update memory
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = 1 + 0.1×100 = 11
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full

What About Write Miss?

- *Write allocate*: The block is loaded into cache on a write miss
- *No-Write allocate*: The block is modified in the lower levels of memory but not in cache
- Write buffer allows merging of writes



Write-Back

- Alternative: On data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first

Write Allocation

- What should happen on a write miss?
- Alternatives for write-through
  - Allocate on miss: fetch the block
  - Write around: don’t fetch the block
    - Since programs often write a whole block before reading it (e.g., initialization)
- For write-back
  - Usually fetch the block

Types of Misses

- **Compulsory:** first time data accessed
- **Capacity:** cache too small to hold all data of interest
- **Conflict:** data of interest maps to same location in cache

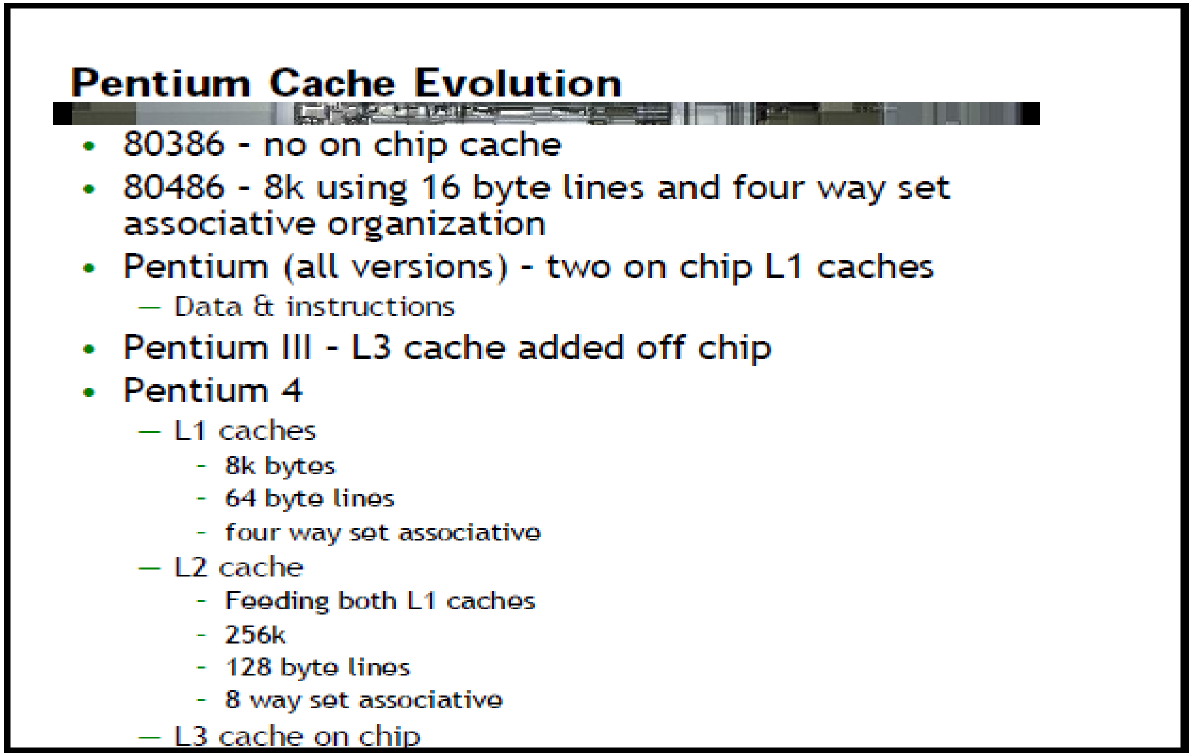
Miss penalty: time it takes to retrieve a block from lower level of hierarchy

Cache Summary

- **What data is held in the cache?**
  - Recently used data (temporal locality)
  - Nearby data (spatial locality)
- **How is data found?**
  - Set is determined by address of data
  - Word within block also determined by address
  - In associative caches, data could be in one of several ways
- **What data is replaced?**
  - Least-recently used way in the set

Multilevel Caches

- Larger caches have lower miss rates, longer access times
- Expand memory hierarchy to multiple levels of caches
- Level 1: small and fast (e.g. 16 KB, 1 cycle)
- Level 2: larger and slower (e.g. 256 KB, 2-6 cycles)
- Most modern PCs have L1, L2, and L3 cache



Pentium Cache Evolution

- 80386 - no on chip cache
- 80486 - 8k using 16 byte lines and four way set associative organization
- Pentium (all versions) - two on chip L1 caches
  - Data & instructions
- Pentium III - L3 cache added off chip
- Pentium 4
  - L1 caches
    - 8k bytes
    - 64 byte lines
    - four way set associative
  - L2 cache
    - Feeding both L1 caches
    - 256k
    - 128 byte lines
    - 8 way set associative
  - L3 cache on chip

Improving Cache Performance

- Reduce the time to hit in the cache
  - smaller cache
  - direct mapped cache
  - smaller blocks
  - for writes
    - Write-through
    - Write-back

- Reduce the miss rate
  - bigger cache
  - more flexible placement (increase associativity)
  - larger blocks (16 to 64 bytes typical)
  - Use “victim cache” – small buffer holding most recently discarded blocks

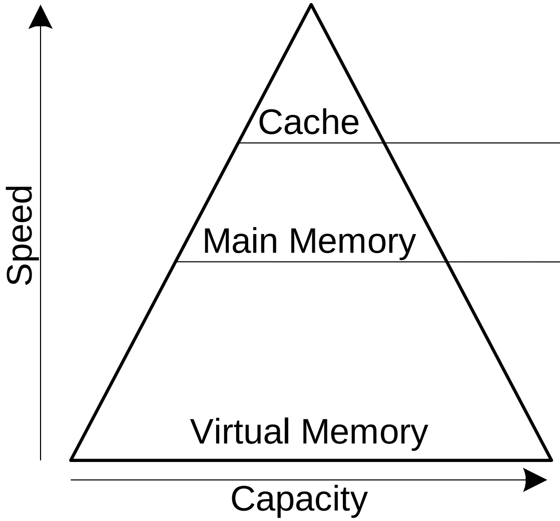
Improving Cache Performance

- Reduce the miss penalty
  - smaller blocks
  - use a write buffer to hold dirty blocks being replaced so don’t have to wait for the write to complete before reading
  - check write buffer (and/or victim cache) on read miss – may get lucky
  - for large blocks fetch critical word first
  - use multiple cache levels – L2 cache not tied to CPU clock rate
  - faster backing store/improved memory bandwidth
    - wider buses
    - memory interleaving, page mode DRAMs

Virtual Memory

- Gives the illusion of bigger memory
- Main memory (DRAM) acts as cache for hard disk

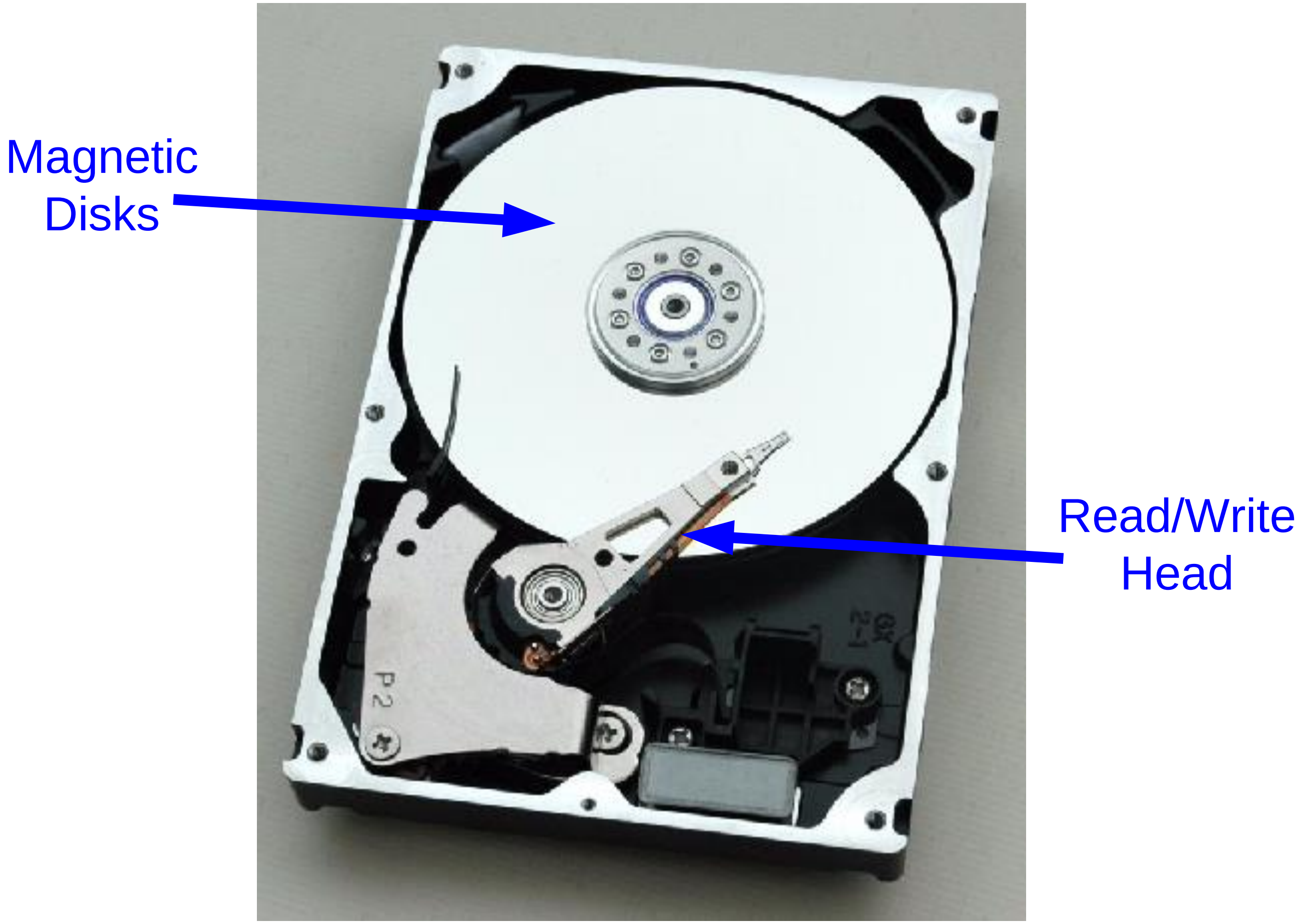
Memory Hierarchy



Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$10,000	1	25+
DRAM	$10	10 - 50	10
SSD	$1	100,000	0.5
HDD	$0.1	10,000,000	0.1

- **Physical Memory:** DRAM (Main Memory)
- **Virtual Memory:** Hard drive
  - Slow, Large, Cheap

Hard Disk



Magnetic Disks

Read/Write Head

Takes milliseconds to *seek* correct location on disk

Virtual Memory

- **Virtual addresses**
  - Programs use virtual addresses
  - Entire virtual address space stored on a hard drive
  - Subset of virtual address data in DRAM
  - CPU translates virtual addresses into ***physical addresses*** (DRAM addresses)
  - Data not in DRAM fetched from hard drive
- **Memory Protection**
  - Each program has own virtual to physical mapping
  - Two programs can use same virtual address for different data
  - Programs don't need to be aware others are running
  - One program (or virus) can't corrupt memory used by another

Cache/Virtual Memory Analogues

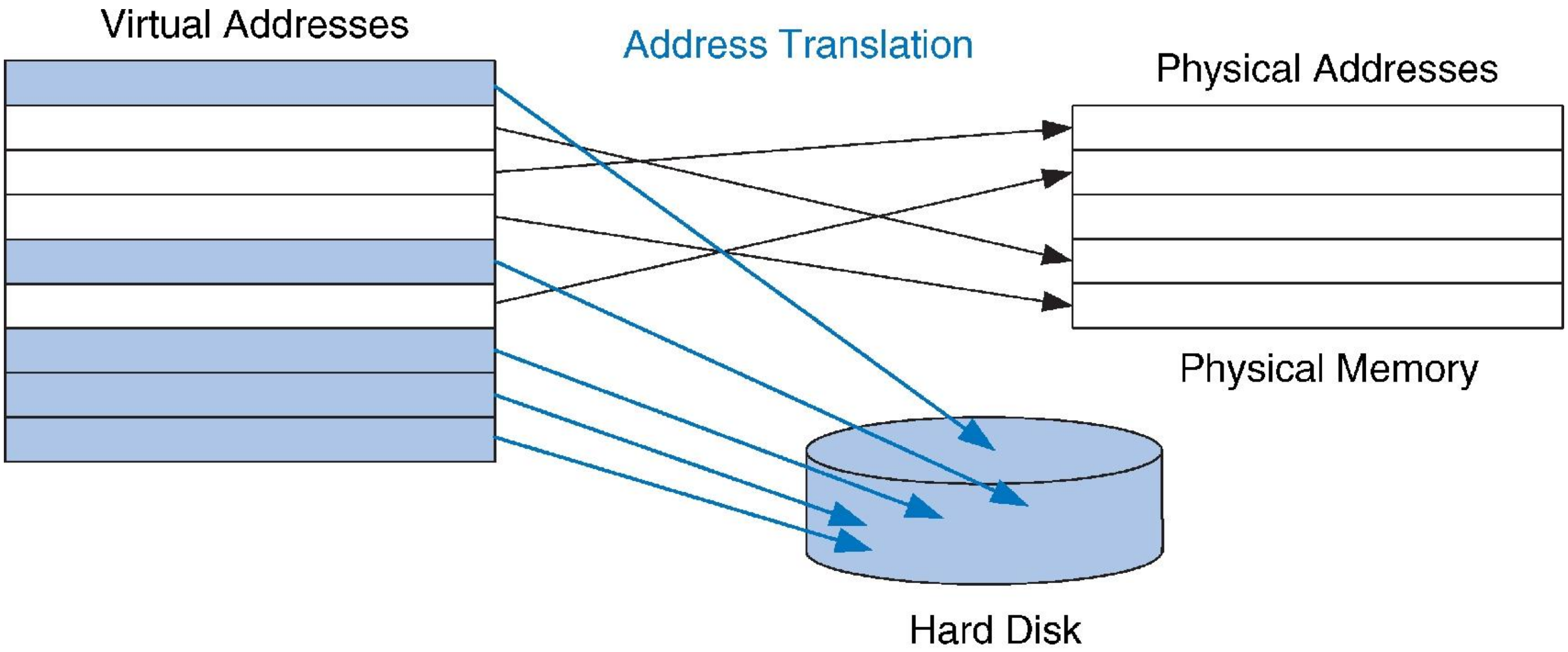
Cache	Virtual Memory
Block	Page
Block Size	Page Size
Block Offset	Page Offset
Miss	Page Fault
Tag	Virtual Page Number

Physical memory acts as cache for virtual memory

Virtual Memory Definitions

- **Page size:** amount of memory transferred from hard disk to DRAM at once
- **Address translation:** determining physical address from virtual address
- **Page table:** lookup table used to translate virtual addresses to physical addresses

Virtual & Physical Addresses

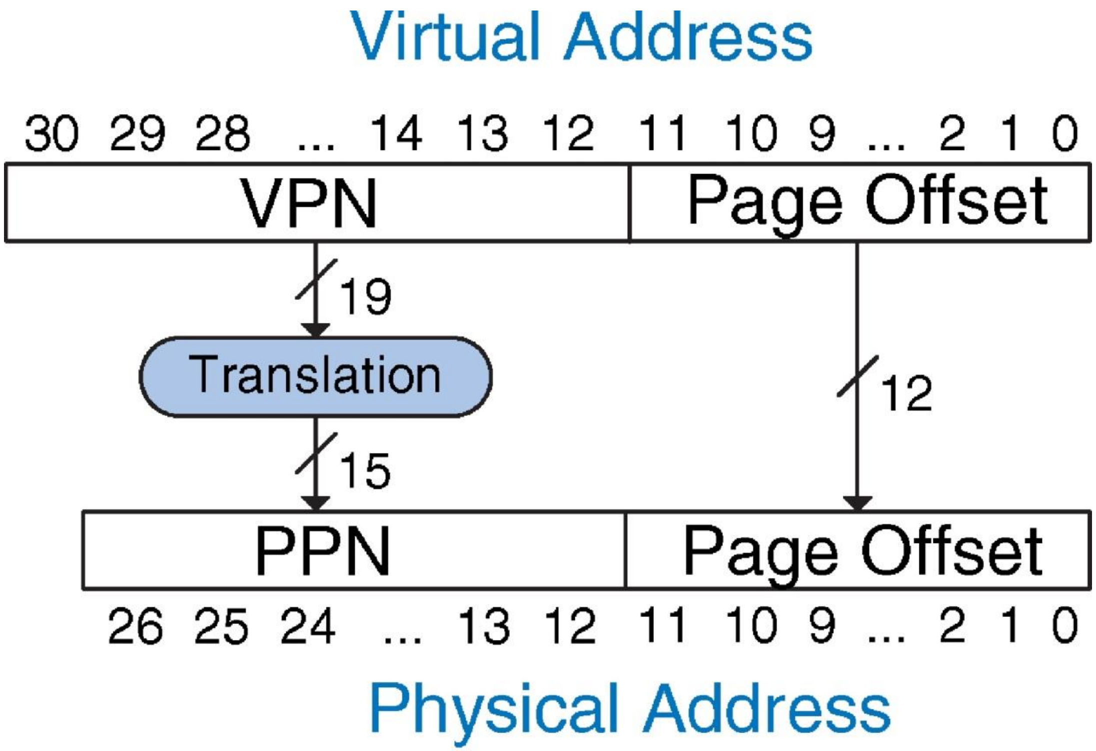


© 2007 Elsevier, Inc. All rights reserved

Most accesses hit in physical memory

But programs have the large capacity of virtual memory

Address Translation



Translation Lookaside Buffer (TLB)

- Small cache of most recent translations
- Reduces # of memory accesses for *most* loads/stores from 2 to 1

TLB

- Page table accesses: high temporal locality
  - Large page size, so consecutive loads/stores likely to access same page
- TLB
  - Small: accessed in < 1 cycle
  - Typically 16 - 512 entries
  - Fully associative
  - > 99 % hit rates typical
  - Reduces # of memory accesses for most loads/stores from 2 to 1

Memory Protection

- Multiple processes (programs) run at once
- Each process has its own page table
- Each process can use entire virtual address space
- A process can only access physical pages mapped in its own page table

Virtual Memory Summary

- Virtual memory increases **capacity**
- A subset of virtual pages in physical memory
- **Page table** maps virtual pages to physical pages – address translation
- A **TLB** speeds up address translation
- Different page tables for different programs provides **memory protection**

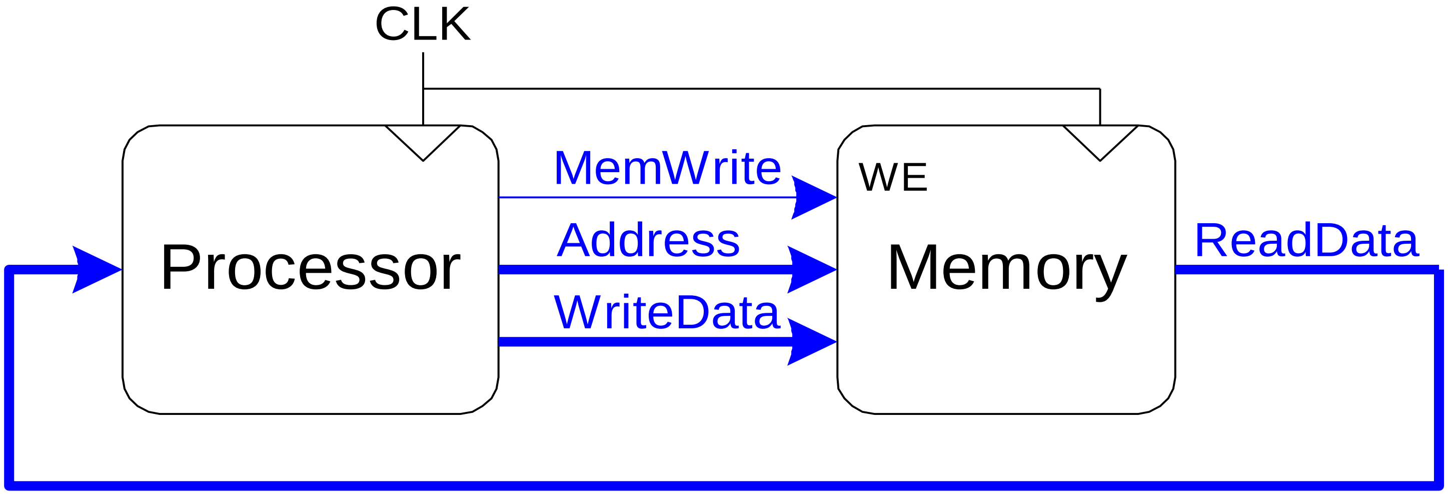
Memory-Mapped I/O

- Processor accesses I/O devices just like memory (like keyboards, monitors, printers)
- Each I/O device assigned one or more address
- When that address is detected, data read/written to I/O device instead of memory
- A portion of the address space dedicated to I/O devices

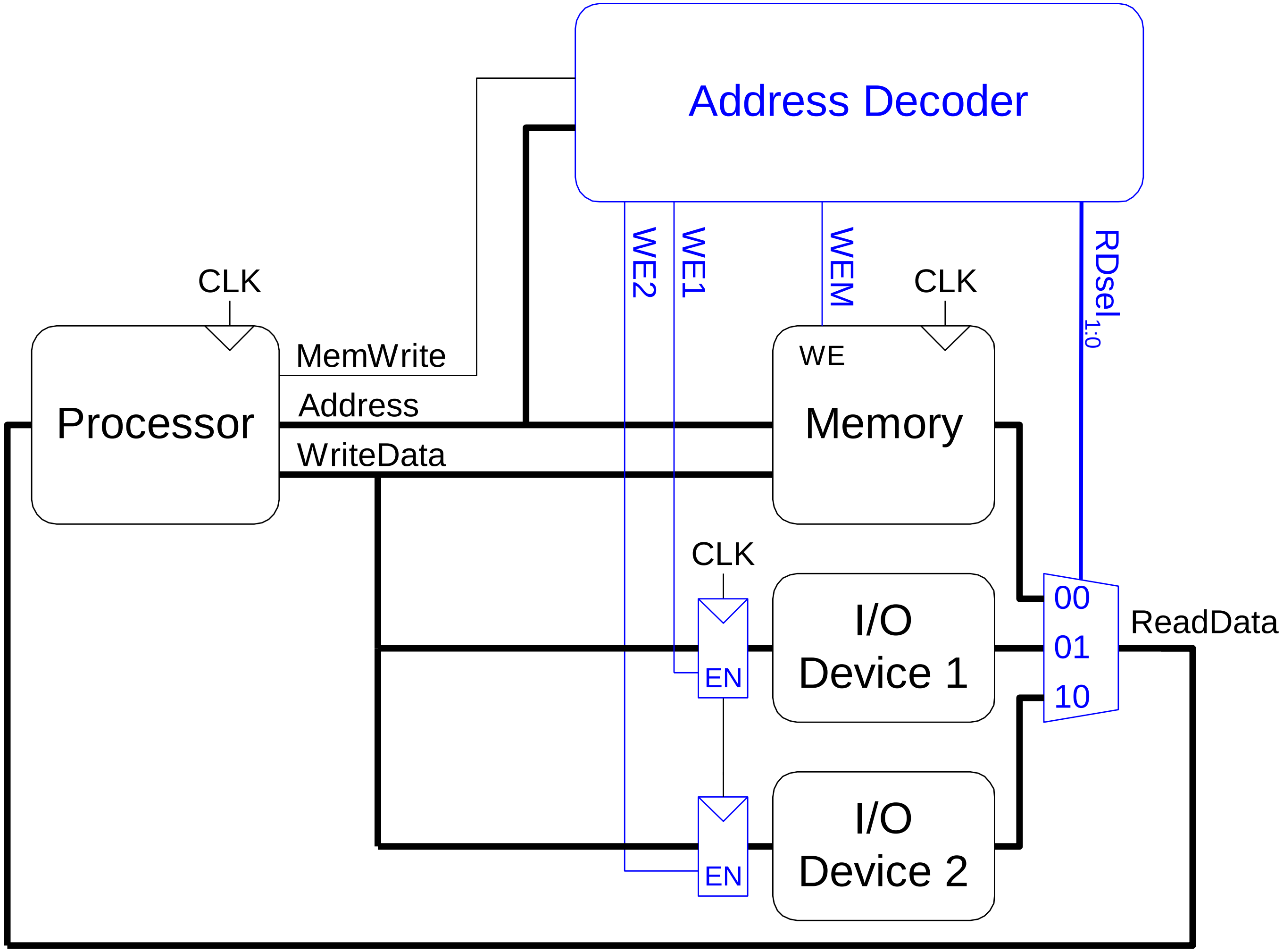
Memory-Mapped I/O Hardware

- **Address Decoder:**
  - – Looks at address to determine which device/memory communicates with the processor
- **I/O Registers:**
  - – Hold values written to the I/O devices
- **ReadData Multiplexer:**
  - – Selects between memory and I/O devices as source of data sent to the processor

The Memory Interface



Memory-Mapped I/O Hardware



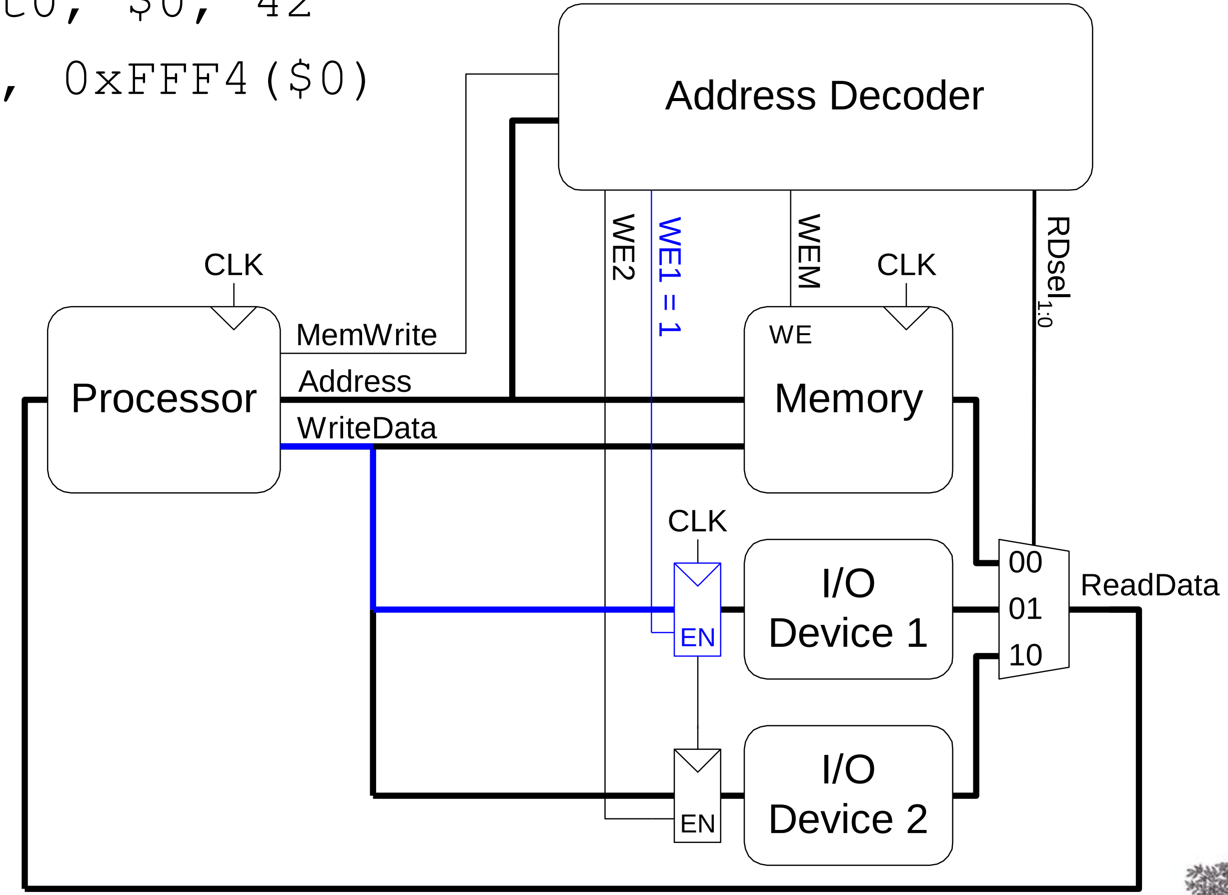
Memory-Mapped I/O Code

- Suppose I/O Device 1 is assigned the address 0xFFFFFFF4
  - Write the value 42 to I/O Device 1
  - Read value from I/O Device 1 and place in `$t3`

Memory-Mapped I/O Code

- **Write the value 42 to I/O Device 1 (0xFFFFFFF4)**

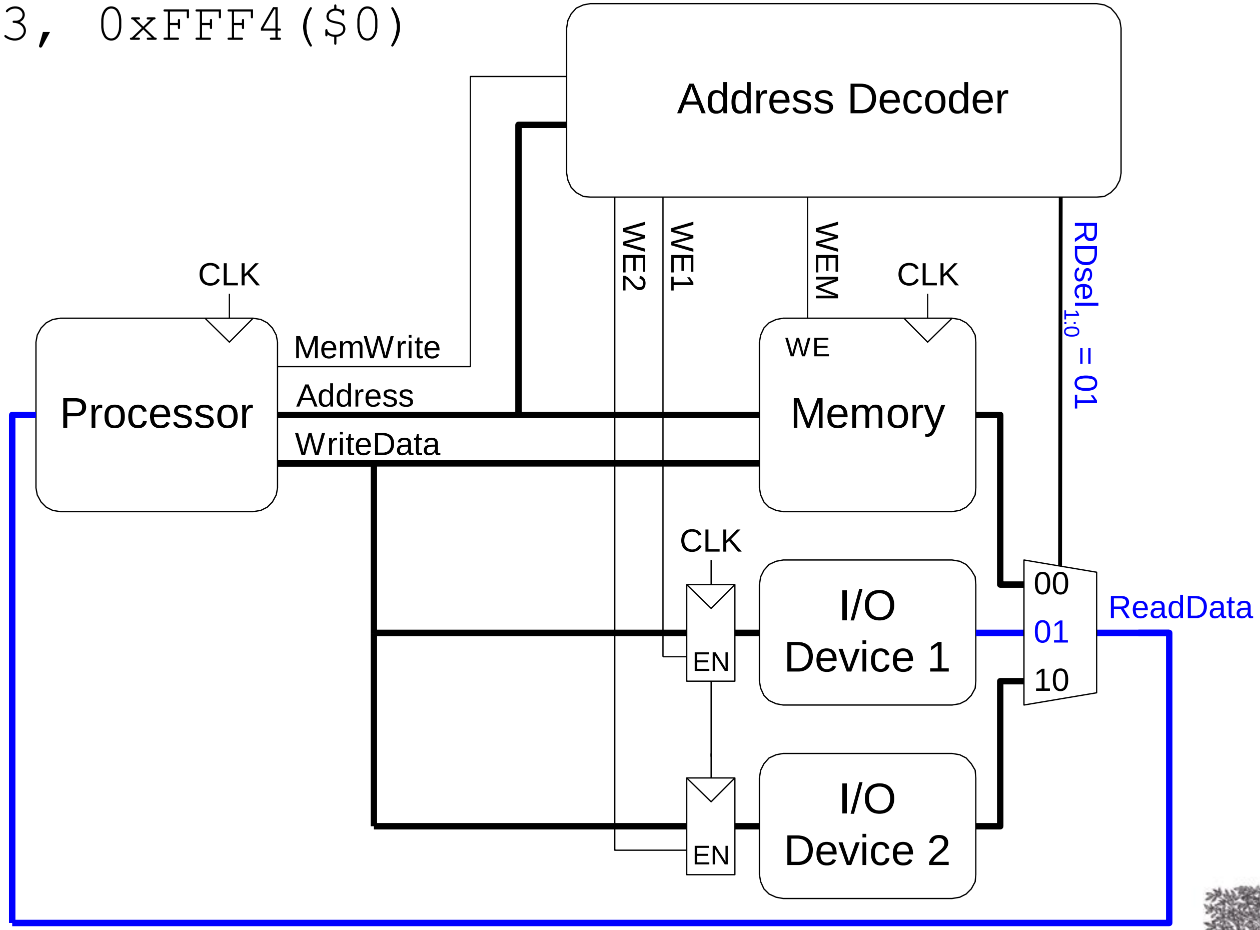
```
addi $t0, $0, 42
sw $t0, 0xFFF4($0)
```



Memory-Mapped I/O Code

- **Read the value from I/O Device 1 and place in `$t3`**

```
lw $t3, 0xFFF4($0)
```



Input/Output (I/O) Systems

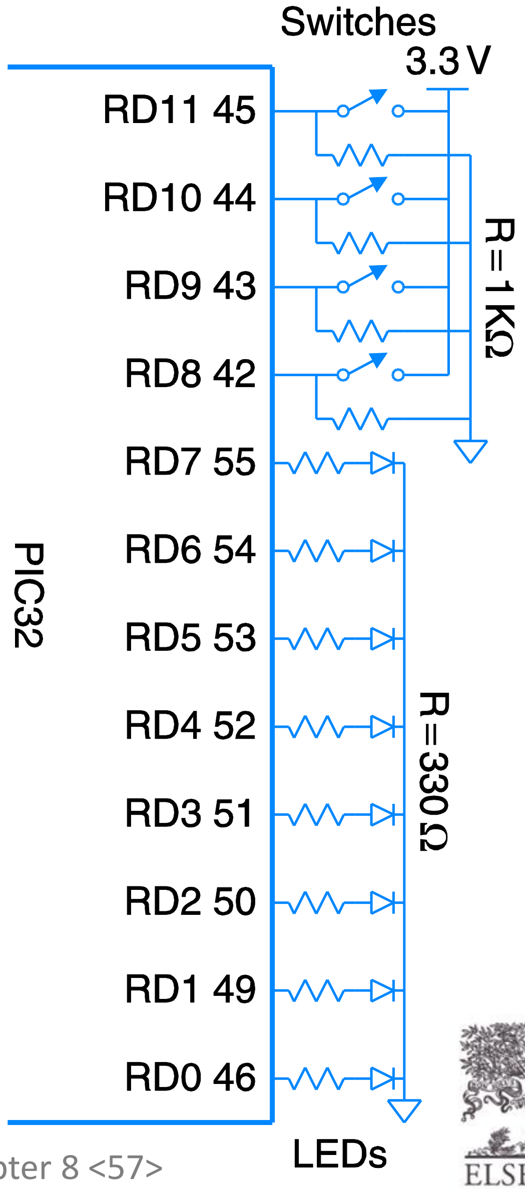
- Embedded I/O Systems
  - Toasters, LEDs, etc.
- PC I/O Systems

Embedded I/O Systems

- Example microcontroller: PIC32
  - microcontroller
  - 32-bit MIPS processor
  - low-level peripherals include:
    - serial ports
    - timers
    - A/D converters

Digital I/O

```
// C Code
#include <p3xxxx.h>

int main(void) {
  int switches;
  TRISD = 0xFF00;        // RD[7:0] outputs
                         // RD[11:8] inputs
  while (1) {
    // read & mask switches, RD[11:8]
    switches = (PORTD >> 8) & 0xF;
    PORTD = switches;  // display on LEDs
  }
}
```

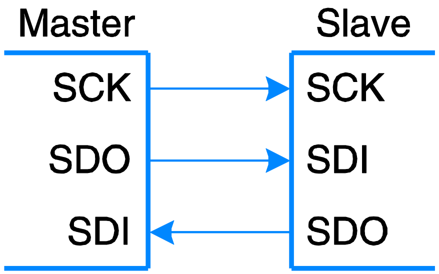


Serial I/O

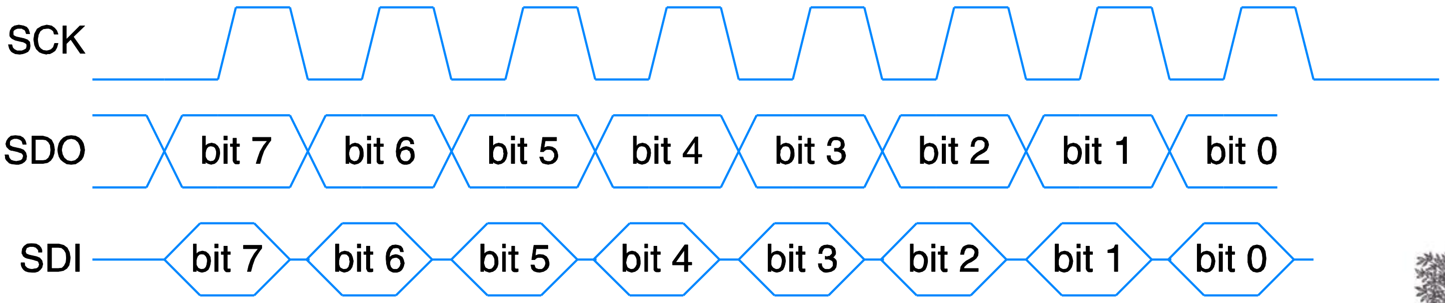
- Example serial protocols
  - **SPI:** Serial Peripheral Interface
  - **UART:** Universal Asynchronous Receiver/Transmitter
  - Also: I^2C, USB, Ethernet, etc.

SPI: Serial Peripheral Interface

- Master initiates communication to slave by sending pulses on SCK
- Master sends SDO (Serial Data Out) to slave, msb first
- Slave may send data (SDI) to master, msb first



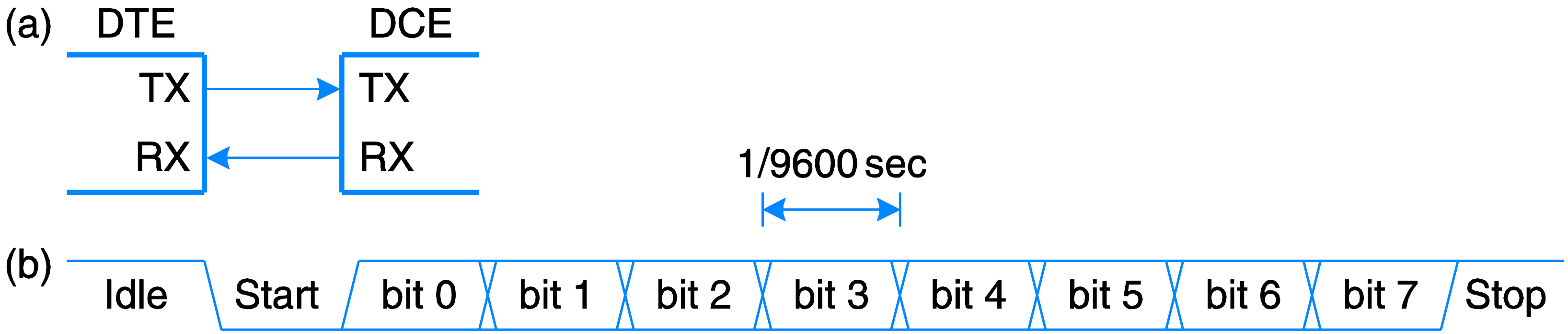
(a)



(b)

UART: Universal Asynchronous Rx/Tx

- Configuration:
  - start bit (0), 7-8 data bits, parity bit (optional), 1+ stop bits (1)
  - data rate: 300, 1200, 2400, 9600, …115200 baud
- Line idles HIGH (1)
- Common configuration:
  - 8 data bits, no parity, 1 stop bit, 9600 baud



Timers

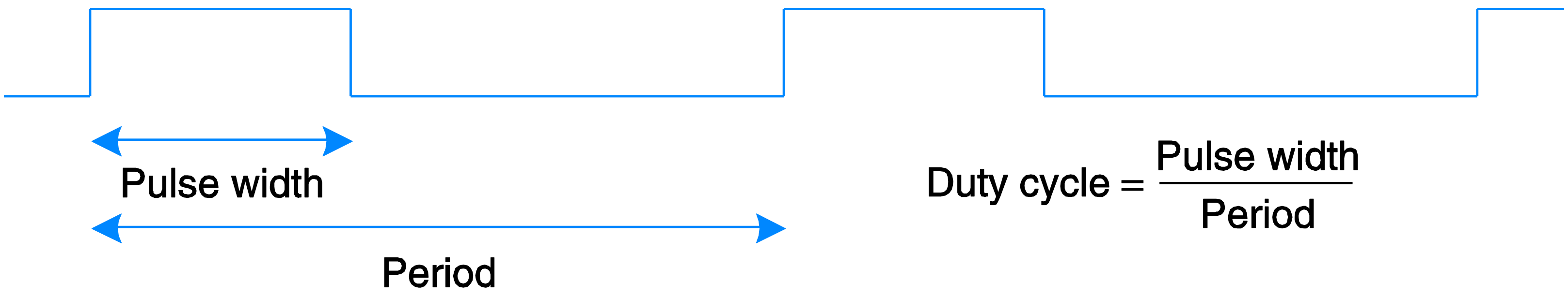
```
// Create specified ms/us of delay using built-in timer
#include <P32xxxx.h>

void delaymicros(int micros) {
  if (micros > 1000) {        // avoid timer overflow
    delaymicros(1000);
    delaymicros(micros-1000);
  }
  else if (micros > 6){
    TMR1 = 0;                 // reset timer to 0
    T1CONbits.ON = 1;         // turn timer on
    PR1 = (micros-6)*20;      // 20 clocks per microsecond
                              // Function has overhead of ~6 us
    IFS0bits.T1IF = 0;        // clear overflow flag
    while (!IFS0bits.T1IF);   // wait until overflow flag set
  }
}

void delaymillis(int millis) {
  while (millis--) delaymicros(1000); // repeatedly delay 1 ms
}                                     // until done
```

Analog I/O

- Needed to interface with outside world
- **Analog input:** Analog-to-digital (A/D) conversion
  - Often included in microcontroller
  - *N*-bit: converts analog input from V_{ref-}-V_{ref+} to 0-2^{N-1}
- **Analog output:**
  - Digital-to-analog (D/A) conversion
    - Typically need external chip (e.g., AD558 or LTC1257)
    - *N*-bit: converts digital signal from 0-2^{N-1} to V_{ref-}-V_{ref+}
  - Pulse-width modulation

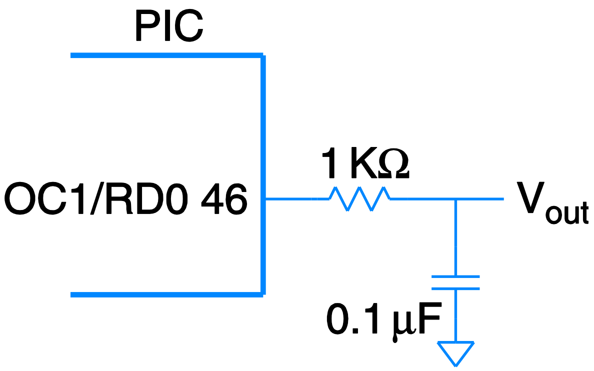
Pulse-Width Modulation (PWM)

- Average value proportional to duty cycle



$$\text{Duty cycle} = \frac{\text{Pulse width}}{\text{Period}}$$

- Add high-pass filter on output to deliver average value



Other Microcontroller Peripherals

- Examples
  - Character LCD
  - VGA monitor
  - Bluetooth wireless
  - Motors

Personal Computer (PC) I/O Systems

- USB: Universal Serial Bus
  - USB 1.0 released in 1996
  - standardized cables/software for peripherals
- PCI/PCIe: Peripheral Component Interconnect/PCI Express
  - developed by Intel, widespread around 1994
  - 32-bit parallel bus
  - used for expansion cards (i.e., sound cards, video cards, etc.)
- DDR: double-data rate memory

Personal Computer (PC) I/O Systems

- TCP/IP: Transmission Control Protocol and Internet Protocol
  - physical connection: Ethernet cable or Wi-Fi
- SATA: hard drive interface
- Input/Output (sensors, actuators, microcontrollers, etc.)
  - Data Acquisition Systems (DAQs)
  - USB Links