CMSC 313 Lecture 14

- Announcement:
  Project 4 due date extended to Fri 10/17
- Reminder:
  Midterm Exam next Monday 10/20
- Project 4 Questions
- Cache Memory
- Interrupts
- Review for midterm exam

Project 4: C Functions

Due: Tue 10/14/03, Section 0101 (Chang) & Section 0301 (Macneil)
Wed 10/15/03, Section 0201 (Patel & Bourner)

Objective

The objective of this programming exercise is to practice writing assembly language programs that use the C function call conventions.

Assignment

Convert your assembly language program from Project 3 as follows:

1. Convert the program into one that follows the C function call convention, so it may be called from a C program. Your program should work with the following function prototype:
   The intention here is that the first parameter is a pointer to the records array and the second parameter has the number of items in that array.

   ```
   void report (void *, unsigned int) ;
   ```

   The intention here is that the first parameter is a pointer to the records array and the second parameter has the number of items in that array.

2. Modify your program so it uses the strncmp() function from the C library to compare the nicknames of two records. The function prototype of `strncmp()` is:

   ```
   int strncmp(const char *s1, const char *s2, size_t n) ;
   ```

   The function returns an integer less than, equal to, or greater than zero if s1 (or the first n bytes thereof) is found, respectively, to be less than, to match, or be greater than s2.

3. Modify your program so that it prints out the entire record (not just the `realname` field) of the record with the least number of points and the record with the alphabetically first nickname. You must use the `printf()` function from the C library to produce this output. The output of your program would look something like:

   ```
   Lowest Points: James Pressman (jamieboy)
     Alignment: Lawful Neutral
     Role: Fighter
     Points: 57
     Level: 1
   First Nickname: Dan Gannett (danmeister)
     Alignment: True Neutral
     Role: Ranger
     Points: 7502
     Level: 3
   ```

A sample C program that should work with your assembly language implementation of the `report()` function is available on the GL file system: `/afs/umbc.edu/users/c/h/chang/pub/cs313/records2.c`

Implementation Notes

- Documentation for the printf() and strncmp() functions are available on the Unix system by typing `man -S 3 printf` and `man -S 3 strncmp`.
- Note that the strncmp() function takes 3 parameters, not 2. It is good programming practice to use `strncmp()` instead of `strcmp()` since this prevents runaway loops if the strings are not properly null terminated. The third argument should be 16, the length of the `nickname` field.

- As in Project 3, you must also make your own test cases. The example in `records2.c` does not fully exercise your program. As before, your program will be graded based upon other test cases. If you have good examples in Project 3, you can just reuse those.
- Use `gcc` to link and load your assembly language program with the C program. This way, `gcc` will call `ld` with the appropriate options:

```
nasm -f elf report2.asm
gcc records2.c report2.o
```

- Notes on the C function call conventions are available on the web:

```
http://www.csee.umbc.edu/~chang/cs313.f03/stack.shtml
```

- Your program should be reasonably robust and report errors encountered (e.g., empty array) rather than crashing.

Turning in your program

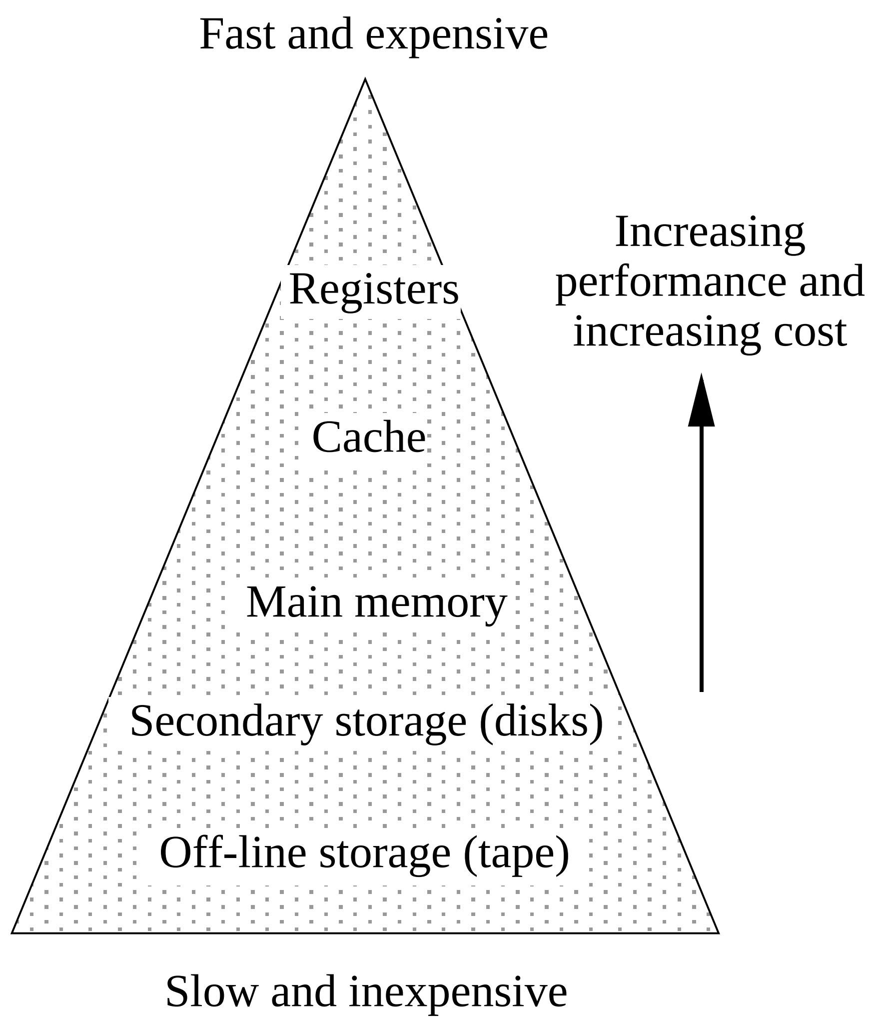
Use the UNIX `submit` command on the GL system to turn in your project. You should submit at least 4 files: your assembly language program, at least 2 of your own test cases and a typescript file of sample runs of your program. The class name for submit is `cs313_0101`, `cs313_0102` or `cs313_0103` for respectively sections 0101 (Chang), 0201 (Patel & Bourner) or 0301 (Macneil). The name of the assignment name is `proj4`. The UNIX command to do this should look something like:

```
submit cs313_0103 proj4 report2.asm myrec1.c myrec2.c typescript
```

Last Time: Virtual Memory

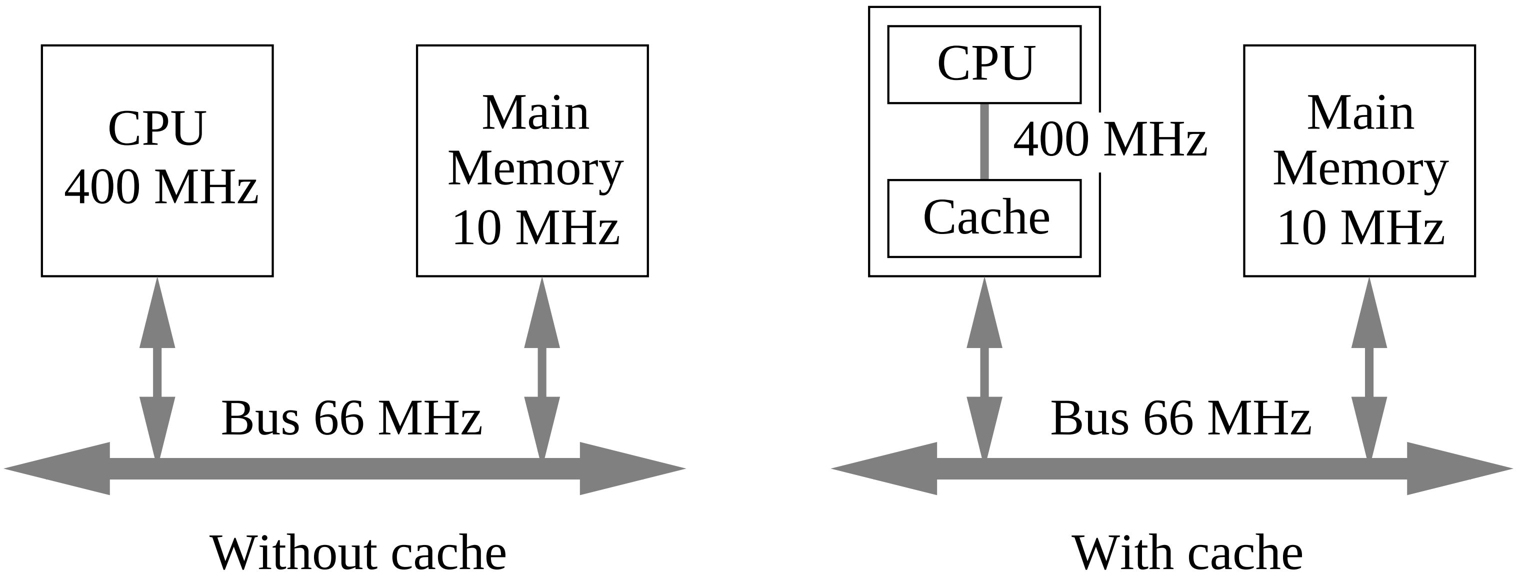
- **Not enough physical memory**
  - Uses disk space to simulate extra memory
  - Pages not being used can be swapped out
    (how and when you'll learn in CMSC 421 Operating Systems)
  - Thrashing: pages constantly written to and retrieved from disk
    (time to buy more RAM)
- **Fragmentation**
  - Contiguous blocks of virtual memory do not have to map to contiguous sections of real memory
- **Memory protection**
  - Each process has its own page table
  - Shared pages are read-only
  - User processes cannot alter the page table (must be supervisor)

The Memory Hierarchy



© 1999 M. Murdocca and V. Heuring

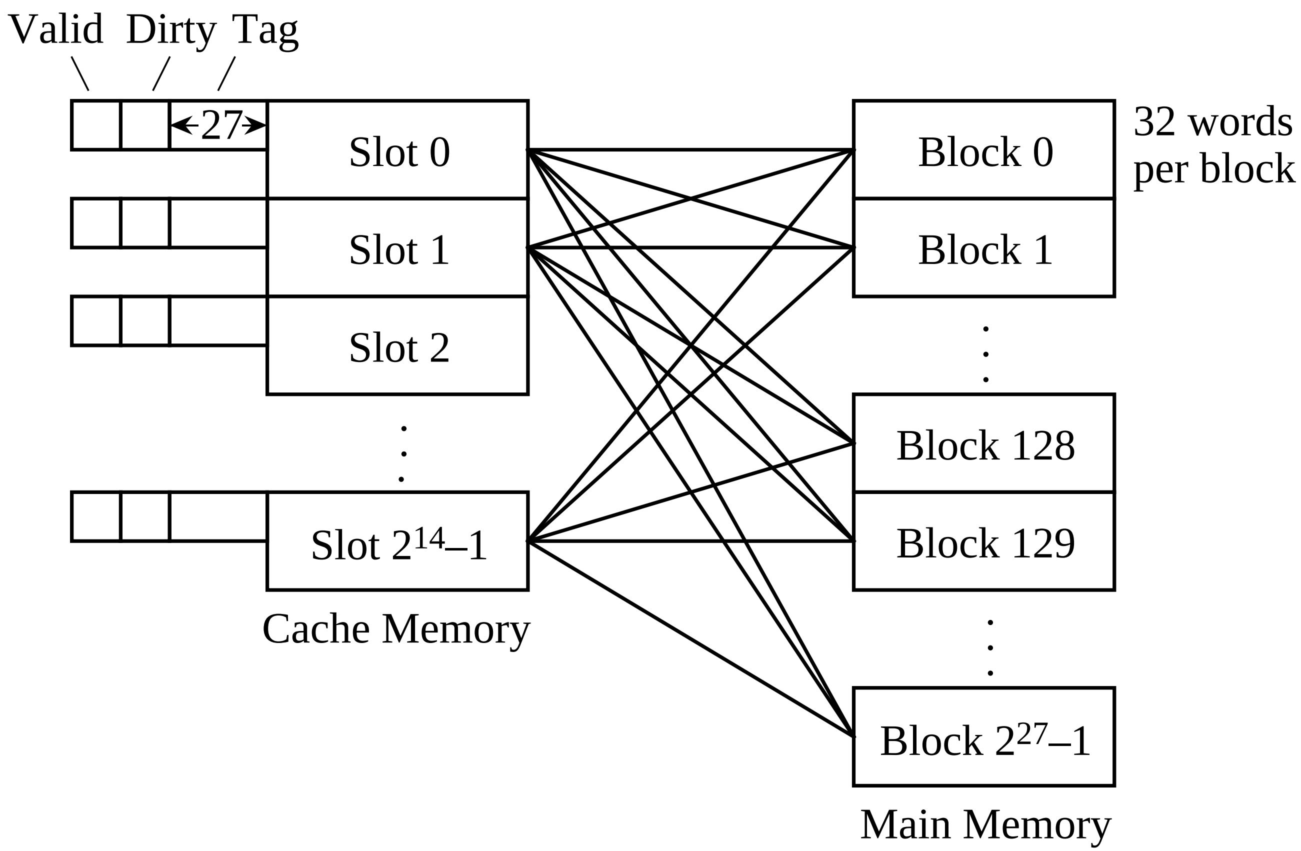
Placement of Cache in a Computer System



- **The *locality principle*: a recently referenced memory location is likely to be referenced again (*temporal locality*); a neighbor of a recently referenced memory location is likely to be referenced (*spatial locality*).**

 © 1999 M. Murdocca and V. Heuring

An Associative Mapping Scheme for a Cache Memory



© 1999 M. Murdocca and V. Heuring

Associative Mapping Example

- **Consider how an access to memory location $(A035F014)_{16}$ is mapped to the cache for a 2^{32} word memory. The memory is divided into 2^{27} blocks of $2^5 = 32$ words per block, and the cache consists of 2^{14} slots:**

Tag	Word
27 bits	5 bits

- **If the addressed word is in the cache, it will be found in word $(14)_{16}$ of a slot that has tag $(501AF80)_{16}$, which is made up of the 27 most significant bits of the address. If the addressed word is not in the cache, then the block corresponding to tag field $(501AF80)_{16}$ is brought into an available slot in the cache from the main memory, and the memory reference is then satisfied from the cache.**

Tag	Word
1 0 1 0 0 0 0 0 0 0 1 1 0 1 0 1 1 1 1 1 0 0 0 0 0 0 0	1 0 1 0 0

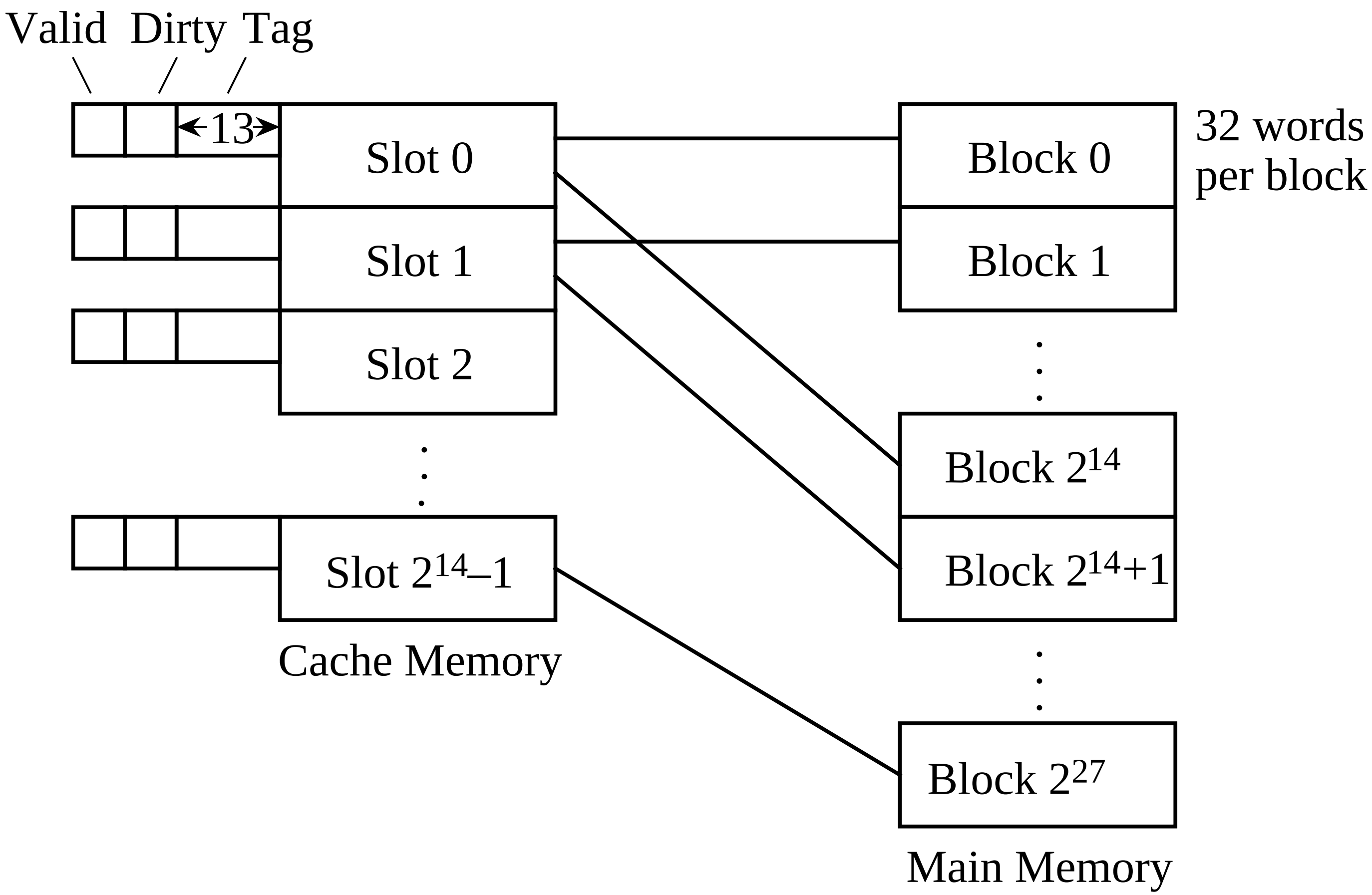
© 1999 M. Murdocca and V. Heuring

Replacement Policies

- When there are no available slots in which to place a block, a *replacement policy* is implemented. The replacement policy governs the choice of which slot is freed up for the new block.
- Replacement policies are used for associative and set-associative mapping schemes, and also for virtual memory.
- Least recently used (LRU)
- First-in/first-out (FIFO)
- Least frequently used (LFU)
- Random
- Optimal (used for analysis only – look backward in time and reverse-engineer the best possible strategy for a particular sequence of memory references.)

© 1999 M. Murdocca and V. Heuring

A Direct Mapping Scheme for Cache Memory



© 1999 M. Murdocca and V. Heuring

Direct Mapping Example

- **For a direct mapped cache, each main memory block can be mapped to only one slot, but each slot can receive more than one block. Consider how an access to memory location $(A035F014)_{16}$ is mapped to the cache for a 2^{32} word memory. The memory is divided into 2^{27} blocks of $2^5 = 32$ words per block, and the cache consists of 2^{14} slots:**

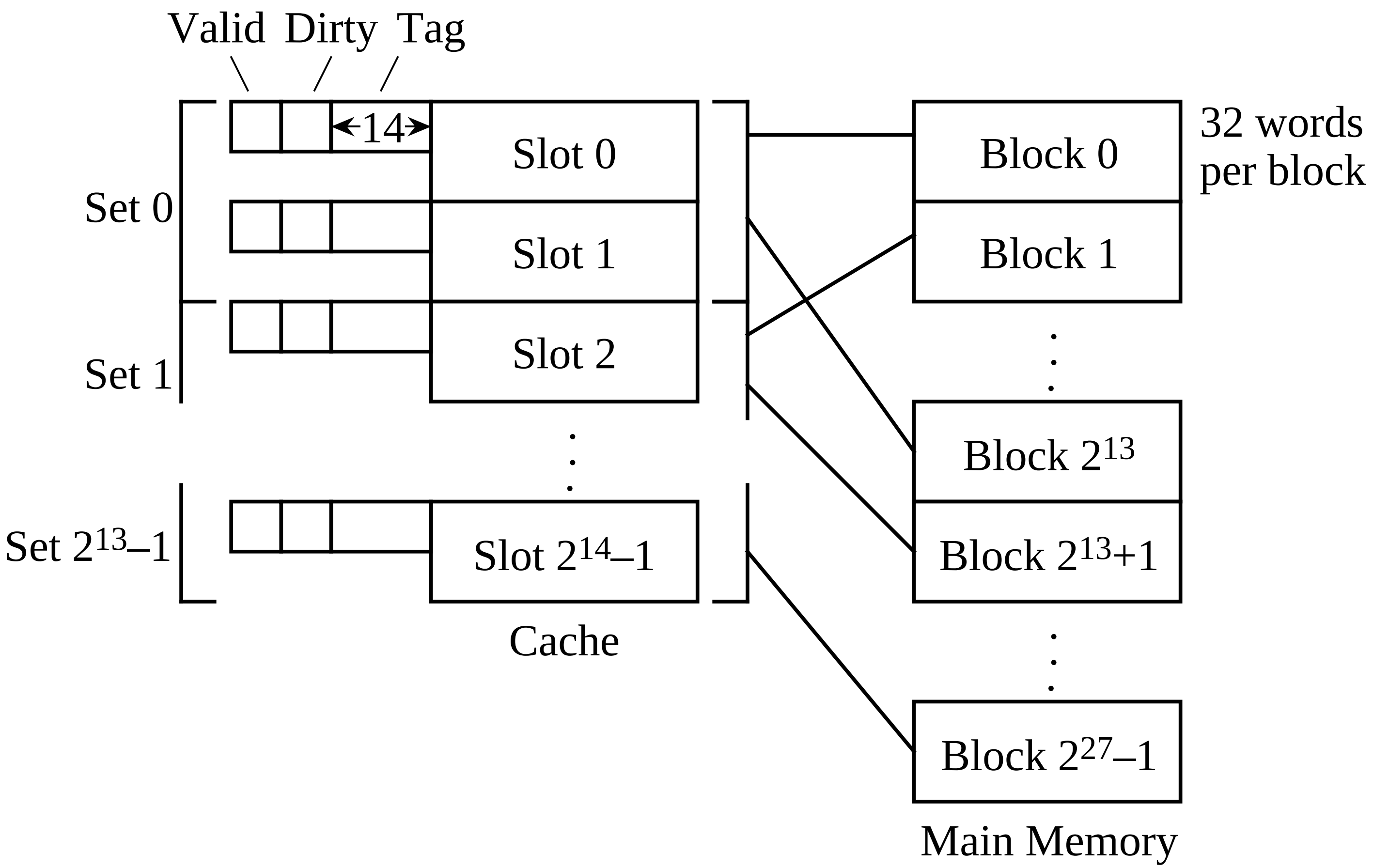
Tag	Slot	Word
13 bits	14 bits	5 bits

- **If the addressed word is in the cache, it will be found in word $(14)_{16}$ of slot $(2F80)_{16}$, which will have a tag of $(1406)_{16}$.**

Tag	Slot	Word
1 0 1 0 0 0 0 0 0 0 1 1 0	1 0 1 1 1 1 1 0 0 0 0 0 0 0	1 0 1 0 0

© 1999 M. Murdocca and V. Heuring

A Set Associative Mapping Scheme for a Cache Memory



© 1999 M. Murdocca and V. Heuring

Set-Associative Mapping Example

- **Consider how an access to memory location $(A035F014)_{16}$ is mapped to the cache for a 2^{32} word memory. The memory is divided into 2^{27} blocks of $2^5 = 32$ words per block, there are two blocks per set, and the cache consists of 2^{14} slots:**

Tag	Set	Word
14 bits	13 bits	5 bits

- **The leftmost 14 bits form the tag field, followed by 13 bits for the set field, followed by five bits for the word field:**

Tag	Set	Word
1 0 1 0 0 0 0 0 0 0 1 1 0 1	0 1 1 1 1 1 0 0 0 0 0 0 0	1 0 1 0 0

 © 1999 M. Murdocca and V. Heuring

Cache Read and Write Policies

Cache Read

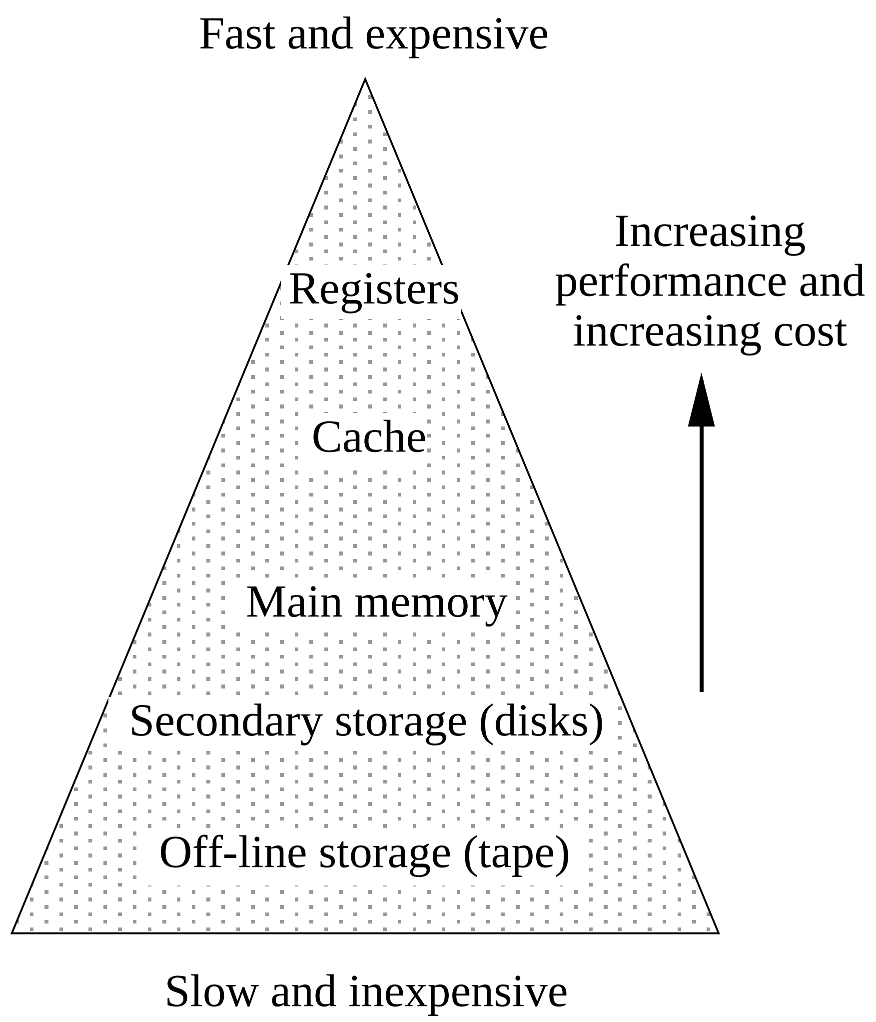
- Data is in the cache → Forward to CPU.
- Data is not in the cache → **Load Through**: Forward the word as cache line is filled,
  -or-
  Fill cache line and then forward word.

Cache Write

- Data is in the cache → **Write Through**: Write data to both cache and main memory,
  -or-
  Write Back: Write data to cache only. Defer main memory write until block is flushed.
- Data is not in the cache → **Write Allocate**: Bring line into cache, then update it,
  -or-
  `Write No-Allocat`
  Update main memory only.

© 1999 M. Murdocca and V. Heuring

The Memory Hierarchy



© 1999 M. Murdocca and V. Heuring

INTERRUPTS

Motivating Example

```
; An Assembly language program for printing data
        MOV EDX, 378H          ;Printer Data Port
        MOV ECX, 0             ;Use ECX as the loop counter
XYZ:    MOV AL, [ABC + ECX]    ;ABC is the beginning of the memory area
                               ; that characters are being printed from
        OUT [DX], AL           ;Send a character to the printer
        INC ECX
        CMP ECX, 100000        ; print this many characters
        JL XYZ
```

Issues:

- ❑ What about difference in speed between the processor and printer?
- ❑ What about the buffer size of the printer?
  - ➢ Small buffer can lead to some lost data that will not get printed

Communication with input/output devices needs handshaking protocols

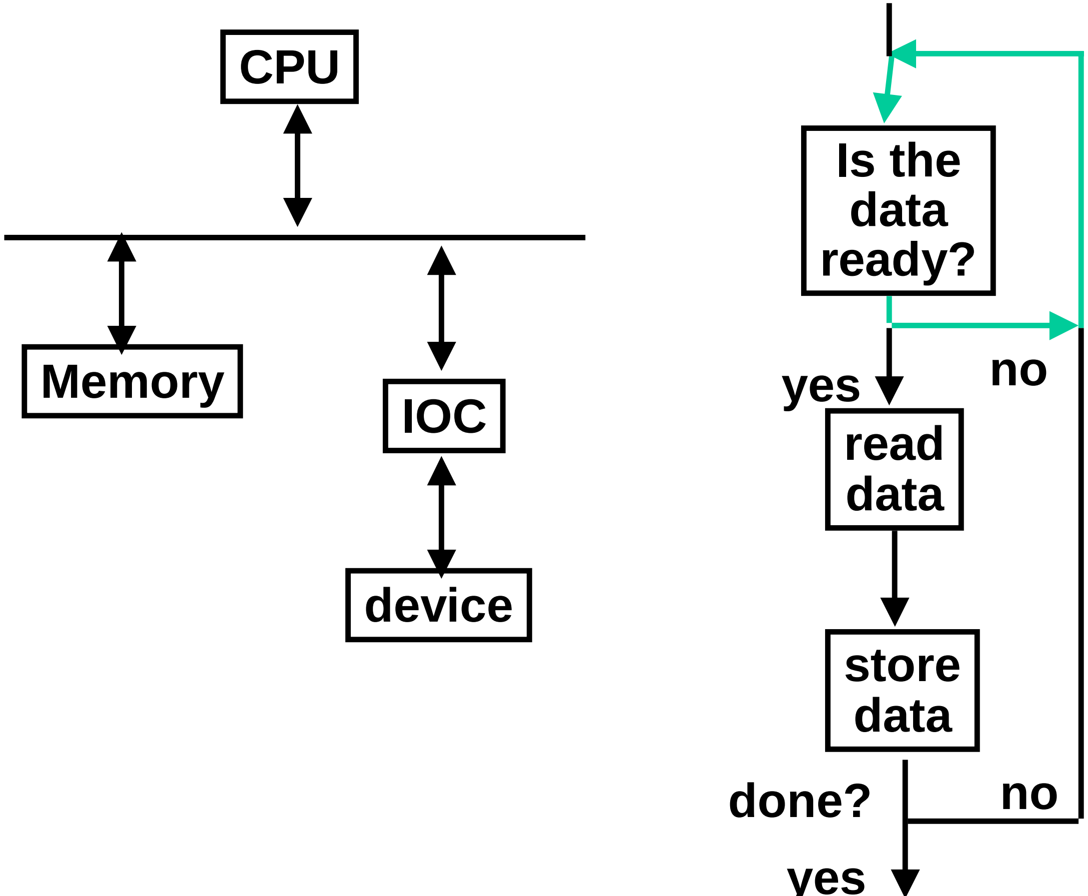
Communicating with I/O Devices

- The OS needs to know when:
  - The I/O device has completed an operation
  - The I/O operation has encountered an error
- This can be accomplished in two different ways:
  - Polling:
    - The I/O device put information in a status register
    - The OS periodically check the status register
  - I/O Interrupt:
    - An I/O interrupt is an externally stimulated event, asynchronous to instruction execution but does NOT prevent instruction completion
    - Whenever an I/O device needs attention from the processor, it interrupts the processor from what it is currently doing
    - Some processors deals with interrupts as special exceptions

These schemes requires heavy processor's involvement and suitable only for low bandwidth devices such as the keyboard

* Slide is partially a courtesy of Dave Patterson

Polling: Programmed I/O



busy wait loop not an efficient way to use the CPU unless the device is very fast!

but checks for I/O completion can be dispersed among computation intensive code

- ❑ Advantage:
  - ➢ Simple: the processor is totally in control and does all the work
- ❑ Disadvantage:
  - ➢ Polling overhead can consume a lot of CPU time

* Slide is courtesy of Dave Patterson

Polling in 80386

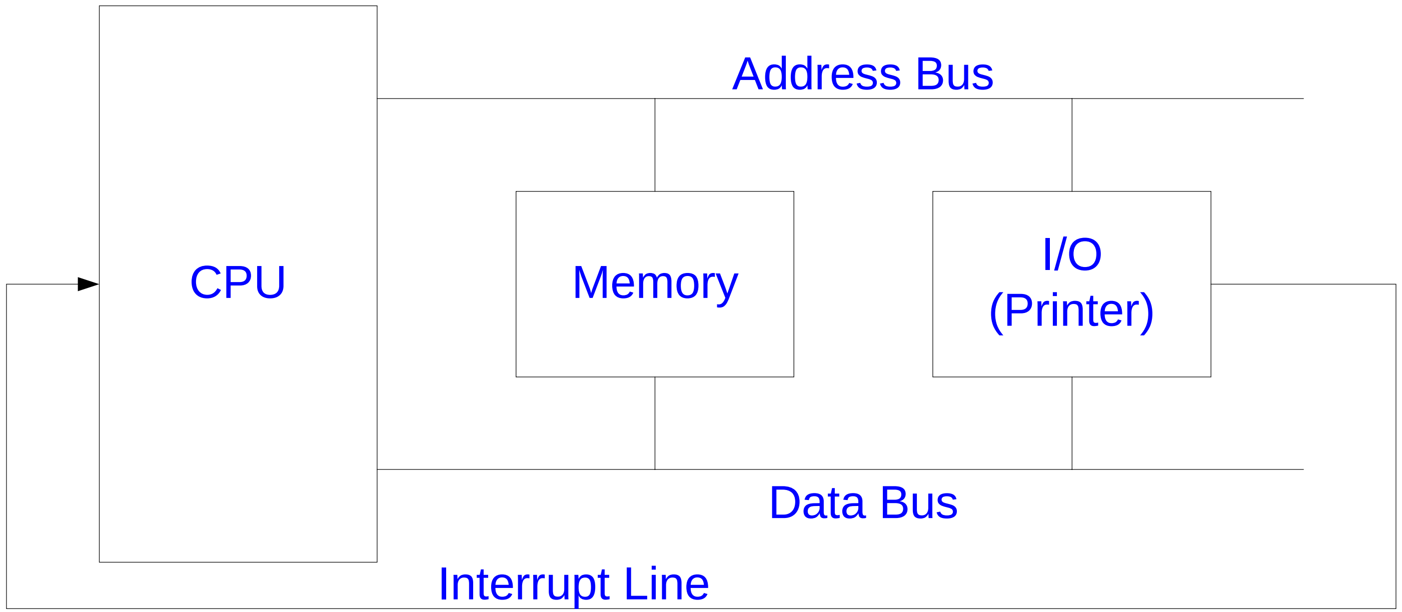
```
        MOV EDX, 379H           ;Printer status port
        MOV ECX, 0
XYZ:    IN AL, [DX]             ;Ask the printer if it is ready
        CMP AL, 1               ;1 means it's ready
        JNE XYZ                 ;If not try again
        MOV AL, [ABC + ECX]
        DEC EDX                 ;Data port is 378H
        OUT [DX], AL            ;Send one byte
        INC ECX
        INC EDX                 ;Put back the status port
        CMP ECX, 100000
        JL XYZ
```

Issues:

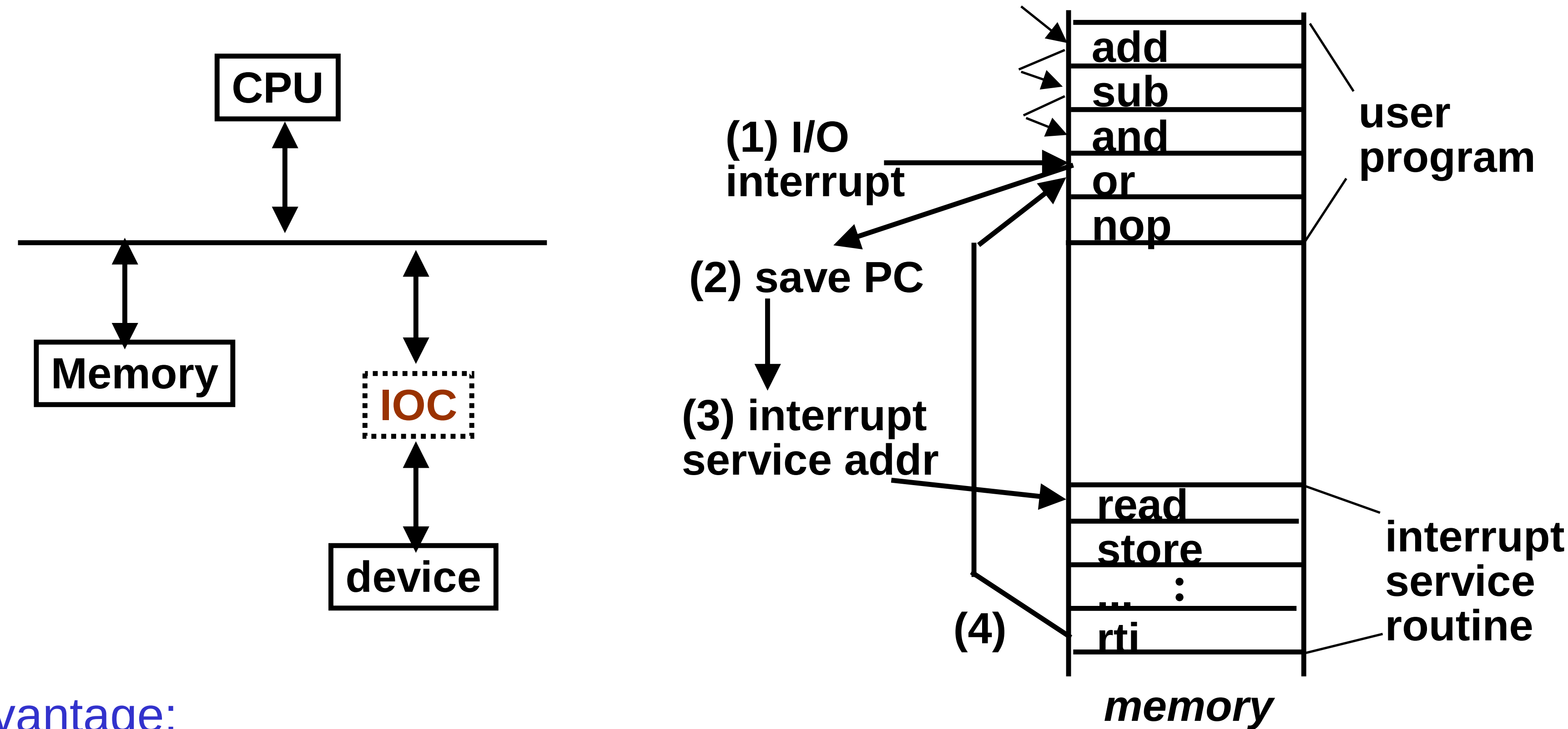
- Status registers (ports) allows handshaking between CPU and I/O devices
- Device status ports are accessible through the use of typical I/O instructions
- CPU is running at the speed of the printer (what a waste!!)

External Interrupt

- The fetch-execute cycle is a program-driven model of computation
- Computers are not totally program driven as they are also hardware driven
- An I/O interrupt is an externally stimulated event, asynchronous to instruction execution but does NOT prevent instruction completion
- Whenever an I/O device needs attention from the processor, it interrupts the processor from what it is currently doing
- Processors typically have one or multiple interrupt pins for device interface



Interrupt Driven Data Transfer

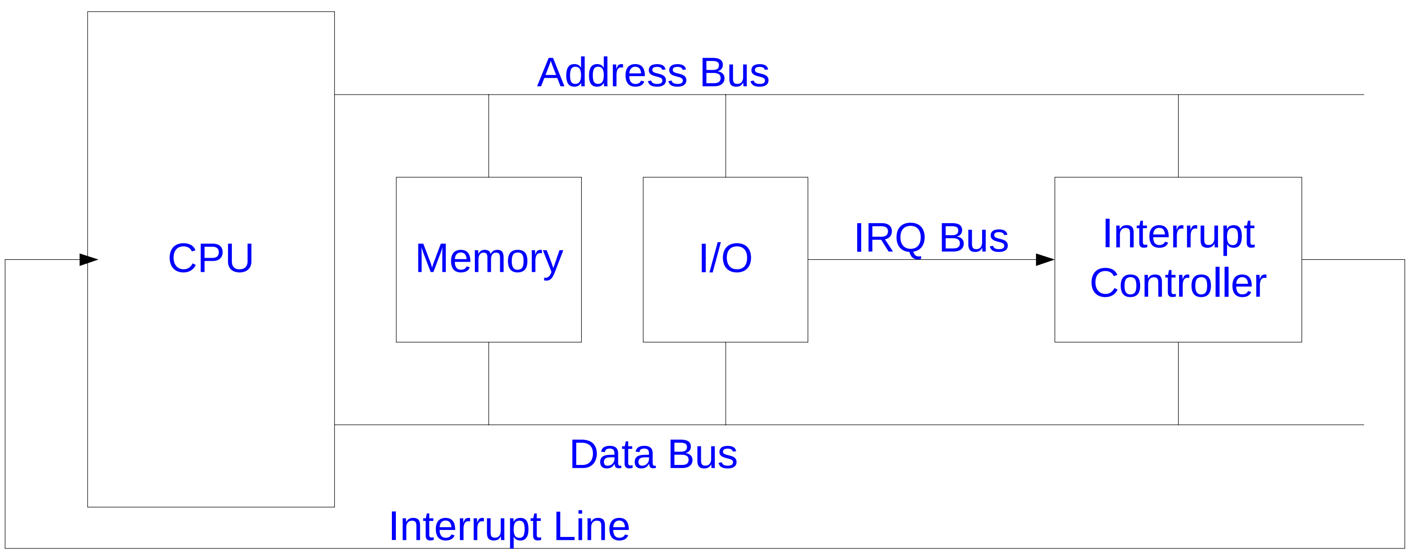


- ❑ Advantage:
  - ➢ User program progress is only halted during actual transfer
- ❑ Disadvantage: special hardware is needed to:
  - ➢ Cause an interrupt (I/O device)
  - ➢ Detect an interrupt (processor)
  - ➢ Save the proper states to resume after the interrupt (processor)

* Slide is courtesy of Dave Patterson

80386 Interrupt Handling

- The 80386 has only one interrupt pin and relies on an interrupt controller to interface and prioritize the different I/O devices
- Interrupt handling follows the following steps:
  - ❶ Complete current instruction
  - ❷ Save current program counter and flags into the stack
  - ❸ Get interrupt number responsible for the signal from interrupt controller
  - ❹ Find the address of the appropriate interrupt service routine
  - ❺ Transfer control to interrupt service routine
- A special interrupt acknowledge bus cycle is used to read interrupt number
- Interrupt controller has ports that are accessible through IN and OUT



Interrupt Descriptor Table

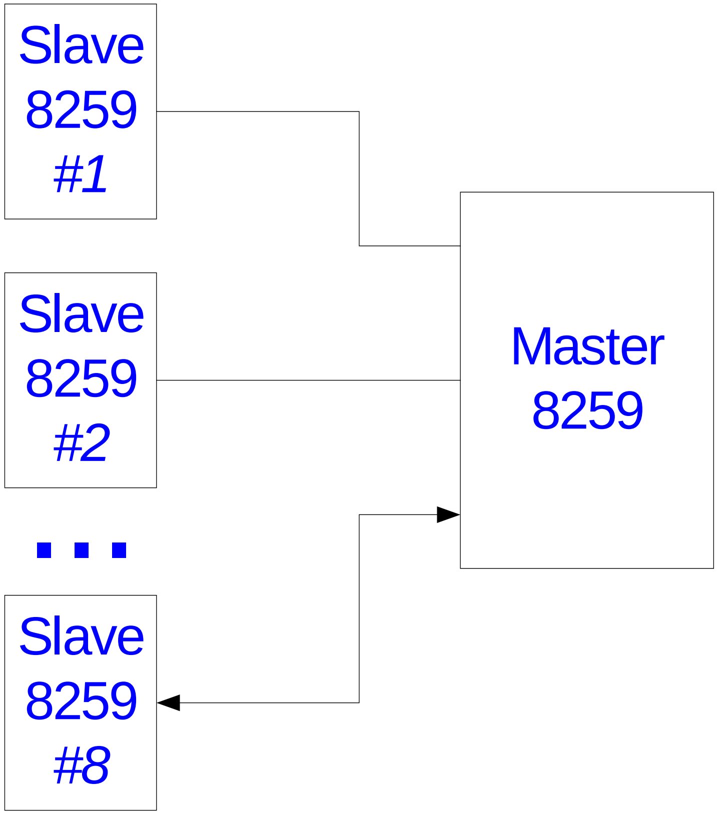
Address	
b	Gate #0
b + 8	Gate #1
b + 16	Gate #2
b + 24	Gate #3
b + 32	Gate #4
b + 40	Gate #5
	. . .
b + 2040	Gate #255

- The address of an ISR is fetched from an interrupt descriptor table
- IDT register is loaded by operating system and points to the interrupt descriptor table
- Each entry is 8 bytes indicating address of ISR and type of interrupt (trap, fault etc.)
- RESET and non-maskable (NMI) interrupts use distinct processor pins
- NMI is used to for parity error or power supply problems and thus cannot be disables

63 48	47 44	43 40	39 16	15 0
ISR Address Upper 2 Bytes		Type		ISR Address Lower 2 Bytes

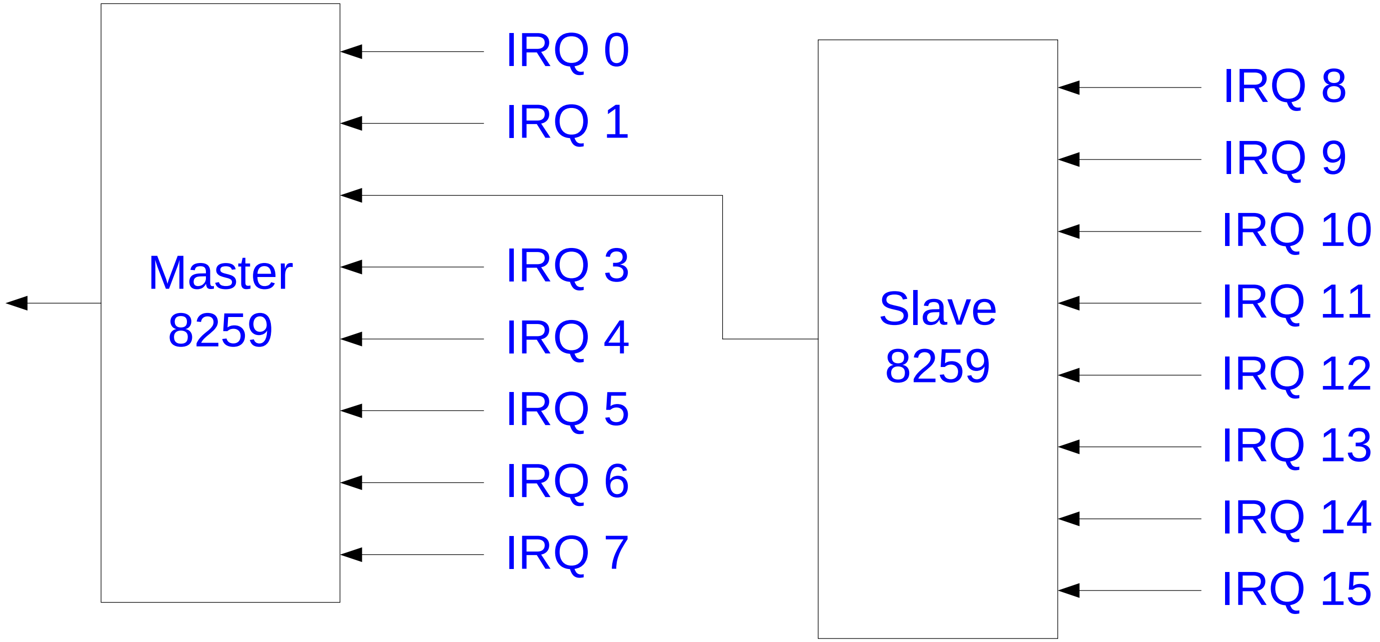
The 8259 Interrupt Controller

- Since the 80386 has one interrupt pin, an interrupt controller is needed to handle multiple input and output devices
- The Intel 8259 is a programmable interrupt controller that can be used either singly or in a two-tier configuration
- When used as a master, the 8259 can interface with up to 8 slaves
- Since the 8259 controller can be a master or a slave, the interrupt request lines must be programmable
- Programming the 8259 chips takes place at boot time using the OUT commands
- The order of the interrupt lines reflects the priority assigned to them



The ISA Architecture

- The ISA architecture is set by IBM competitors and standardizes:
  - The interrupt controller circuitry
  - Many IRQ assignments
  - Many I/O port assignments
  - The signals and connections made available to expansion cards
- A one-master-one-slave configuration is the norm for ISA architecture



- Priority is assigned in the following order:

  IRQ 0, IRQ 1, IRQ 8, …, IRQ 15, IRQ 3, …, IRQ 7

ISA Interrupt Routings

IRQ	ALLOCATION	INTRRUPT NUMBER
IRQ0	System Timer	08H
IRQ1	Keyboard	09H
IRQ3	Serial Port #2	OBH
IRQ4	Serial Port # 1	OCH
IRQ5	Parallel Port #2	ODH
IRQ6	Floppy Controller	OEH
IRQ7	Parallel Port # 1	OFH
IRQ8	Real time clock	70H
IRQ9	available	71 H
IRQ10	available	72H
IRQ11	available	73H
IRQ12	Mouse	74H
IRQ13	87 ERROR line	75H
IRQ14	Hard drive controller	76H
IRQ15	available	77H

linux1$ cat /proc/interrupts

I/O Interrupt vs. Exception

- An I/O interrupt is just like the exceptions except:
  - An I/O interrupt is asynchronous
  - Further information needs to be conveyed
  - Typically exceptions are more urgent than interrupts
- An I/O interrupt is asynchronous with respect to instruction execution:
  - I/O interrupt is not associated with any instruction
  - I/O interrupt does not prevent any instruction from completion
    - You can pick your own convenient point to take an interrupt
- I/O interrupt is more complicated than exception:
  - Needs to convey the identity of the device generating the interrupt
  - Interrupt requests can have different urgencies:
    - Interrupt request needs to be prioritized
    - Priority indicates urgency of dealing with the interrupt
    - High speed devices usually receive highest priority

* Slide is courtesy of Dave Patterson

Internal and Software Interrupt

Exceptions:

- Exceptions do not use the interrupt acknowledge bus cycle but are still handled by a numbered ISR
- Examples: divide by zero, unknown instruction code, access violation, …

Software Interrupts:

- The INT instruction makes interrupt service routines accessible to programmers
- Syntax: "INT imm" with *imm* indicating interrupt number
- Returning from an ISR is like RET, except it enables interrupts

	Ordinary subroutine	**Interrupt service routine**
Invoke	CALL	INT
Terminate	RET	IRET

Fault and Traps:

- When an instruction causes an exception and is retried after handling it, the exception is called faults (e.g. page fault)
- When control is passed to the next instruction after handling an exception or interrupt, such exception is called a trap (e.g. division overflow)

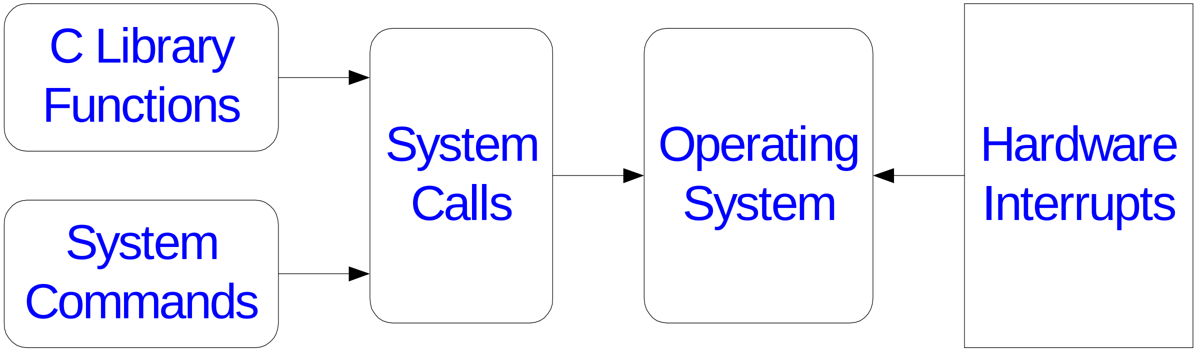
Built-in Hardware Exceptions

Allocation	Int #
Division Overflow	00H
Single Step	01H
NMI	02H
Breakpoint	03H
Interrupt on Overflow	04H
BOUND out of range	05H
Invalid Machine Code	06H
87 not available	07H
Double Fault	08H
87 Segment Overrun	09H
Invalid Task State Segment	0AH
Segment Not Present	0BH
Stack Overflow	0CH
General Protection Error	0DH
Page Fault	0EH
(reserved)	0FH
87 Error	10H

System Calls

- Linux conventions: parameters are stored left to right order in registers EBX, ECX, EDX, EDI and ESI respectively

```
main() {
  char s[] = "Hello world!\n";
  write(1,s,13);
}
```




```
;This program makes a system call
;
          global main
main:     MOV EAX, 4          ;Write is system call #4
          MOV EBX, 1          ;1 is number for standard output
          MOV ECX, ABC        ;ABC is the string pointer
          MOV EDX, 13         ;Write 13 bytes
          INT 80H             ;System call interrupt
          RET
 ABC: db "Hello world!", 0AH,0
```

Privileged Mode

Privilege Levels

- The difference between kernel mode and user mode is in the privilege level
- The 80386 has 4 privilege levels, two of them are used in Linux
  - Level 0: system level (Linux kernel)
  - Level 3: user level (user processes)
- The CPL register stores the current privilege level and is reset during the execution of system calls
- Privileged instructions, such as LIDT that set interrupt tables can execute only when CPL = 0

Stack Issues

- System calls have to use different stack since the user processes will have write access to them (imagine a process passing the stack pointer as a parameter forcing the system call to overwrite its own stack
- There is a different stack pointer for every privilege level stored in the task state segment

Summary: Types of Interrupts

- **Hardware vs Software**
  - ◇ Hardware: I/O, clock tick, power failure, exceptions
  - ◇ Software: INT instruction
- **External vs Internal Hardware Interrupts**
  - ◇ External interrupts are generated by CPU's interrupt pin
  - ◇ Internal interrupts (exceptions): div by zero, single step, page fault, bad opcode, stack overflow, protection, ...
- **Synchronous vs Asynchronous Hardware Int.**
  - ◇ Synchronous interrupts occur at exactly the same place every time the program is executed. E.g., bad opcode, div by zero, illegal memory address.
  - ◇ Asynchronous interrupts occur at unpredictable times relative to the program. E.g., I/O, clock ticks.

Summary: Interrupt Sequence

- Device sends signal to interrupt controller.
- Controller uses IRQ# for interrupt # and priority.
- Controller sends signal to CPU if the CPU is not already processing an interrupt with higher priority.
- CPU finishes executing the current instruction
- CPU saves EFLAGS & return address on the stack.
- CPU gets interrupt # from controller using I/O ops.
- CPU finds “gate” in Interrupt Description Table.
- CPU switches to Interrupt Service Routine (ISR). This may include a change in privilege level. IF cleared.

Interrupt Sequence (cont.)

- ISR saves registers if necessary.
- ISR, after initial processing, sets IF to allow interrupts.
- ISR processes the interrupt.
- ISR restores registers if necessary.
- ISR sends End of Interrupt (EOI) to controller.
- ISR returns from interrupt using IRET. EFLAGS (inlcuding IF) & return address restored.
- CPU executes the next instruction.
- Interrupt controller waits for next interrupt and manages pending interrupts.

Next

- Mon 10/20: Midterm Exam

- Wed 10/22: Introduction to Digital Logic