COSC 511: Computer Architecture
The Processor (Part 2)

Week 5

Last Week

- The Processor (Part 1)
  - Generic architecture of CPUs
  - The CPU execution cycle
  - Improving performance by pipelining the CPU execution cycle
  - Use of caching at multiple levels to improve performance
  - Creating "faster" CPUs by adding more cores
  - Understanding tradeoffs between higher clock speeds and more cores
  - Simultaneous Multithreading
  - Branch Prediction
    - Spectre: A security vulnerability in most CPUs made before 2019 caused by how branch prediction was implemented

The Processor

- Loading and Executing a Program
  - Programs are stored on the system's Hard Drive/Solid State Drive
  - To load and execute a program:
    1. OS tells the hard drive to locate and retrieve the application
    2. OS finds free RAM and copies the application to it
    3. OS creates a process that has the job of executing the program starting at its entry point
       - The entry point is the memory address where the program begins
    4. The process begins executing the application at its entry point
       - The process will take care of running the program
       - The OS has the job of tracking the execution process and responding to requests for system resources (memory allocation, file I/O, etc.)
    5. When the process ends, it is removed from memory

The Processor

- Allowing applications to have unrestricted access to memory can be a security risk.
  - Example:
    - My web browser is using memory between addresses **`0x500000`** and **`0x700000`**.
    - My e-mail client is using memory between addresses **`0x700000`** and **`0x900000`**.
    - I accidentally install malware on my computer that is loaded in memory between **`0x910000`** and **`0x950000`**.
    - Malware accesses memory allocated to my web browser and e-mail client.
    - With this access, it can:
      - Read private data from other applications' memory
      - Deliberately cause other applications to crash
      - Inject code into other applications' memory, causing these applications to execute other malicious code on its behalf
- Some ISAs implement different memory access modes to prevent this.

The Processor

- x86/amd64 Memory Access Modes
  - Real Address Mode
    - This is the mode described in the prior example.
    - Applications running in real address mode have direct access to all memory.
    - Memory addresses that a program "sees" correspond to real, physical memory locations.
  - Protected Mode
    - In protected mode, applications are not allowed to access memory that is not allocated to them.
    - Programs executed in protected mode are allocated a section of linear address space in memory and cannot access anything outside of it.
    - Memory addresses that a program "sees" are virtual addresses that are mapped to physical memory addresses under the hood.
    - Segmentation Faults!

The Processor

- Q: If real address mode is less secure, why do we still have it?
  - A: Some applications need it!
  - Examples:
    - Operating System components
    - Hardware Drivers
    - Antivirus Software

- Applications running in real address mode can switch to protected.
  - Switching from protected to real address mode is not possible.

The Processor

- Real Address Mode Pros
  - Easier to implement than protected mode
  - No additional memory management overhead

- Real Address Mode Cons
  - Huge security risk
  - Does not allow for any memory management

The Processor

- Protected Mode Pros
  - Allows for extensive control over memory management
  - Prevents deliberate or accidental tampering with other applications' memory
    - Exception: Malware that targets applications running in real address mode can circumvent this.

- Protected Mode Cons
  - Harder to implement than real address mode
  - Added complexity due to need for memory management

The Processor

- Memory Management
  - Maps real memory addresses to virtual memory addresses
  - Paging
    - Computer uses secondary storage to offload memory contents and reload them as needed
    - Paged data is either written to a page file or a dedicated drive partition
    - This is useful for when the system needs more memory than it has
    - Problem: Although paging is useful, it causes a noticeable performance bottleneck
      - The secondary storage is slower than RAM, so performance will be degraded
      - Performance degradation is less bad for computers with SSDs, but still present
      - For mechanical hard drives, performance degradation is worse because of higher seek times
        - HDD average seek time: 9ms
        - SSD average seek time: 0.16ms

The Processor

- Another Paging Problem
  - Paging causes lots of read and write operations to your hard drive
  - Old hard drives allowed for an infinite number of write cycles
  - Modern SSDs have a finite number of write cycles
    - An SSD that stores a single data bit per cell typically supports up to 100,000 write cycles.
    - Paging is a very write-heavy operation
  - Frequent paging can kill your SSD prematurely!

The Processor

- Paging: The SSD Killer – Solutions
  - Don't page.
    - Instead of swapping data to internal storage, Android and iOS start killing processes.
  - Add more RAM.
  - Use memory compression!
    - First version of Windows to support memory compression: Windows 10
    - First version of macOS to support memory compression: OS X Mavericks
    - zram, Linux kernel module for memory compression, released in 2014.

The Processor

- Memory Compression Pros
  - Faster than swapping
  - For systems with SSDs, it prevents premature death
- Memory Compression Cons
  - Compressing and decompressing memory contents takes CPU time
- Memory compression does not replace swapping. It supplements it.

The Processor

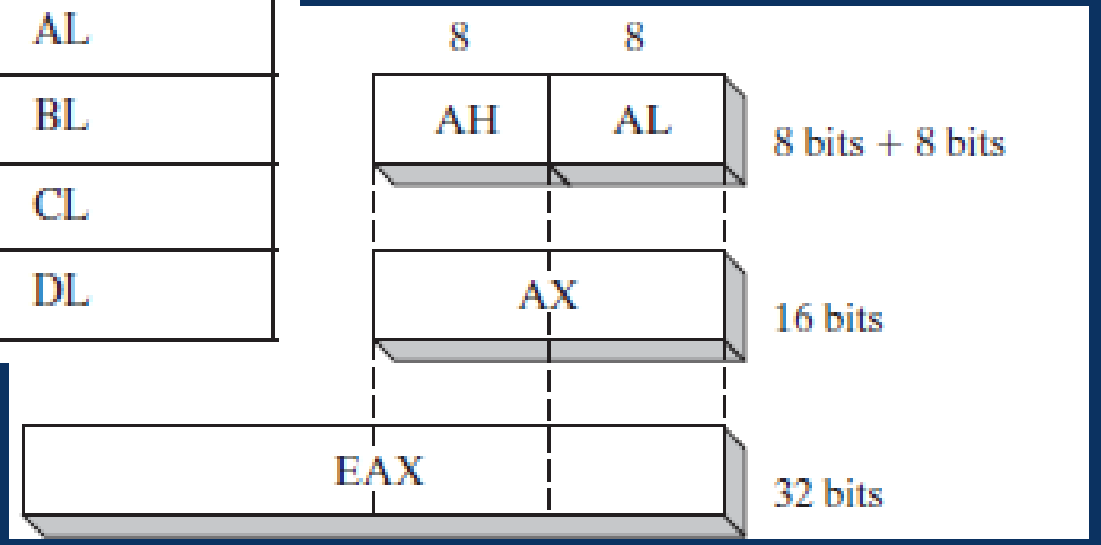
- x86 General Purpose 32-bit Registers
  - EAX
    - Extended Accumulator Register
    - Used for I/O port access, arithmetic operations, interrupts, etc.
  - EBX
    - Extended Base Register
    - Used as a base pointer for memory access
  - ECX
    - Extended Count Register
    - Used as a loop counter
  - EDX
    - Extended Data Register
    - Used to store various data

In practice, these registers can be—and often are—used for storing whatever data the executing program needs.

The Processor

- x86 General Purpose 32-bit Registers
  - Some registers can be accessed in segments

32-Bit	16-Bit	8-Bit (High)	8-Bit (Low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL



The Processor

- x86 General Purpose 32-bit Registers
  - ESP
    - Extended Stack Pointer Register
    - Used as the stack pointer
  - EBP
    - Extended Base Pointer Register
    - Used to reference the bottom of the stack
  - ESI
    - Extended Source Index
    - Used for high-speed memory transfer
  - EDI
    - Extended Data Index
    - Used for high-speed memory transfer

These registers can be used for storing arbitrary data as well, but it's considered bad practice.

The Processor

- More x86 Registers
  - Segment Registers
    - In protected mode, these hold pointers to segment descriptor tables.
    - Segment descriptor tables store information about where the process's segment information is stored.
  - Instruction Pointer
    - Stores the memory address of the next instruction waiting to be executed.
    - Certain instructions modify this value, causing the program to jump to instructions located elsewhere.

The Processor

Instruction Jump Example

```
int x = (some value);
if (x == 4)
{
    x += 1;
}
else
{
    x -= 1;
}
…
```

Memory Location	Instruction
01	Assign (###) to ‘x’
02	Compare ‘x’ with 4
03	If true, change EIP to 04, otherwise change EIP to 06
04	Add 1 to ‘x’
05	Change EIP to 07
06	Subtract 1 from ‘x’
07	(next instruction)
…	(rest of program)

The Processor

- Flag Registers
  - These registers used each of their bits to store a flag that either controls an operation or reflects the outcome of an operation.
    - 1 = set
    - 0 = clear/reset
- Flags
  - Control Flags
    - These control the CPU's operation
    - Can be used to break after each instruction, stop execution when an overflow occurs, etc.
  - Status Flags
    - These are set based on the outcome of an arithmetic or logical operation

The Processor

- Status Flags
  - Carry Flag: Set when the last arithmetic operation (unsigned) resulted in a carry bit that was beyond the size of the destination.

Unsigned 32-bit example:

```
  1 1 … 1 1 1 1 1 1
+ 0 0 … 0 0 0 0 0 1
-------------------
  0 0 … 0 0 0 0 0 0
```

CF will be set to 1 to indicate the overflow.

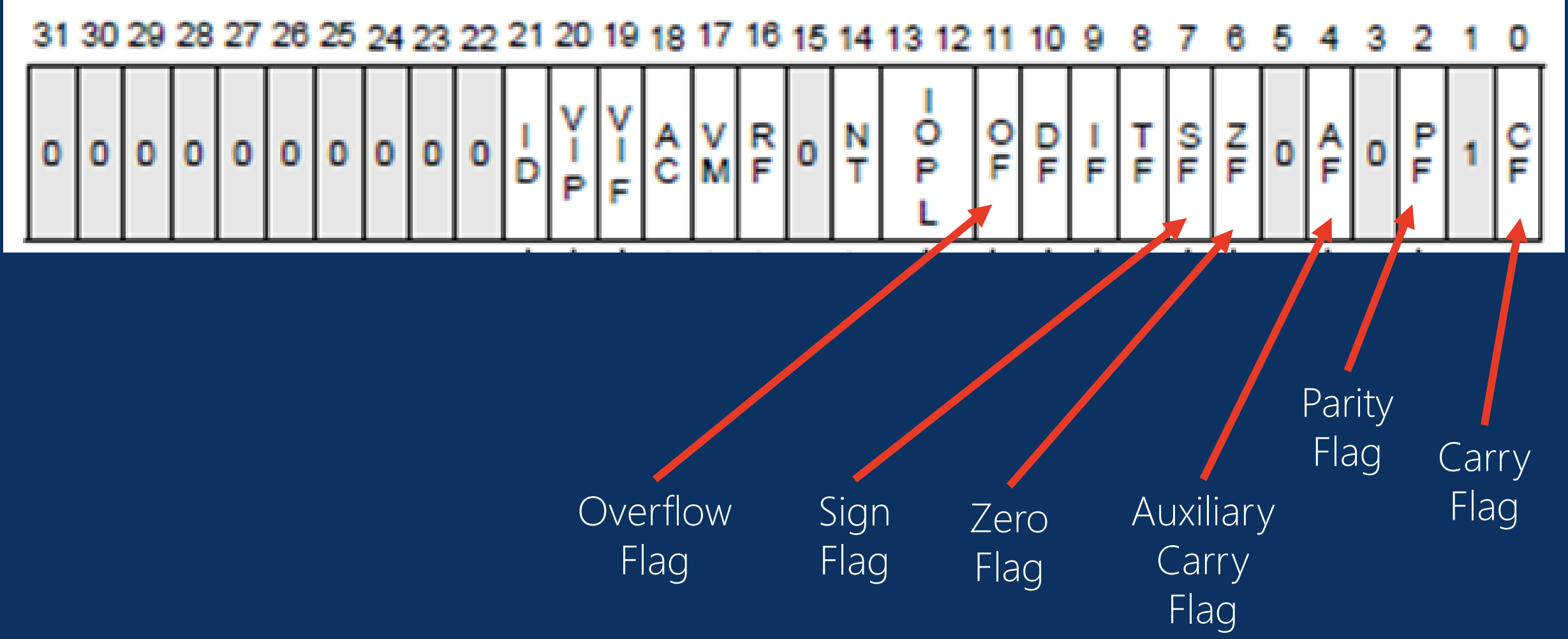
The Processor

- Status Flags
  - Overflow Flag
    - Set when the last arithmetic operation (signed) resulted in a carry bit that was beyond the size of the destination.
  - Sign Flag
    - Set when the last arithmetic operation resulted in a negative value.
  - Zero Flag
    - Set when the last arithmetic operation resulted in a zero result.

The Processor

- Status Flags
  - Auxiliary Carry Flag
    - Set when the last arithmetic operation resulted in a carry bit from the low nibble (lowest four bits) to the high nibble (highest four bits).
  - Parity Flag
    - Set when the least significant byte results in an even number of 1 bits.
    - Used for data validation.

The Processor



The Processor

- Additional Registers – Used to improve performance for certain tasks.
  - MMX Registers (Multi-Media Extension)
    - A set of 8 64-bit registers that are useful for improving performance of multimedia and communications applications
  - Float Point Unit (FPU)
    - Registers used for high-speed floating point arithmetic.
  - XMM Registers
    - 128-bit registers for doing work that is performance-critical

Next Week

- The CPU: Part 3
  - Floating Point Units
  - A little more about x86 and amd64 processors
  - Components of a computer
  - Memory Types
  - Input/Output Systems
  - I/O Access Levels

- Homework #1 due on Canvas at the start of class.