

# *Intel Architecture*

# Objectives

- History of the Intel x86 processor family
- Architecture of the x86 CPU

# History

8086 (1978)	<ul style="list-style-type: none"><li>- 16 bit registers</li><li>- 20 bit addressing (1MB address space)</li></ul>
80286 (1982)	<ul style="list-style-type: none"><li>- Protected mode</li><li>- 24 bit addressing (16 MB address space)</li><li>- various memory protection mechanisms</li></ul>
80386 (1985)	<ul style="list-style-type: none"><li>- 32 bit registers</li><li>- introduced paging</li><li>- 32 bit addressing bus (4 GB address space)</li></ul>
80486 (1989)	<ul style="list-style-type: none"><li>- Parallel execution capability</li></ul>
Pentium (1993)	<ul style="list-style-type: none"><li>- Increased performance</li></ul>
AMD K8 (2000)	<ul style="list-style-type: none"><li>- 64 bit architecture (x86_64)</li></ul>

# History

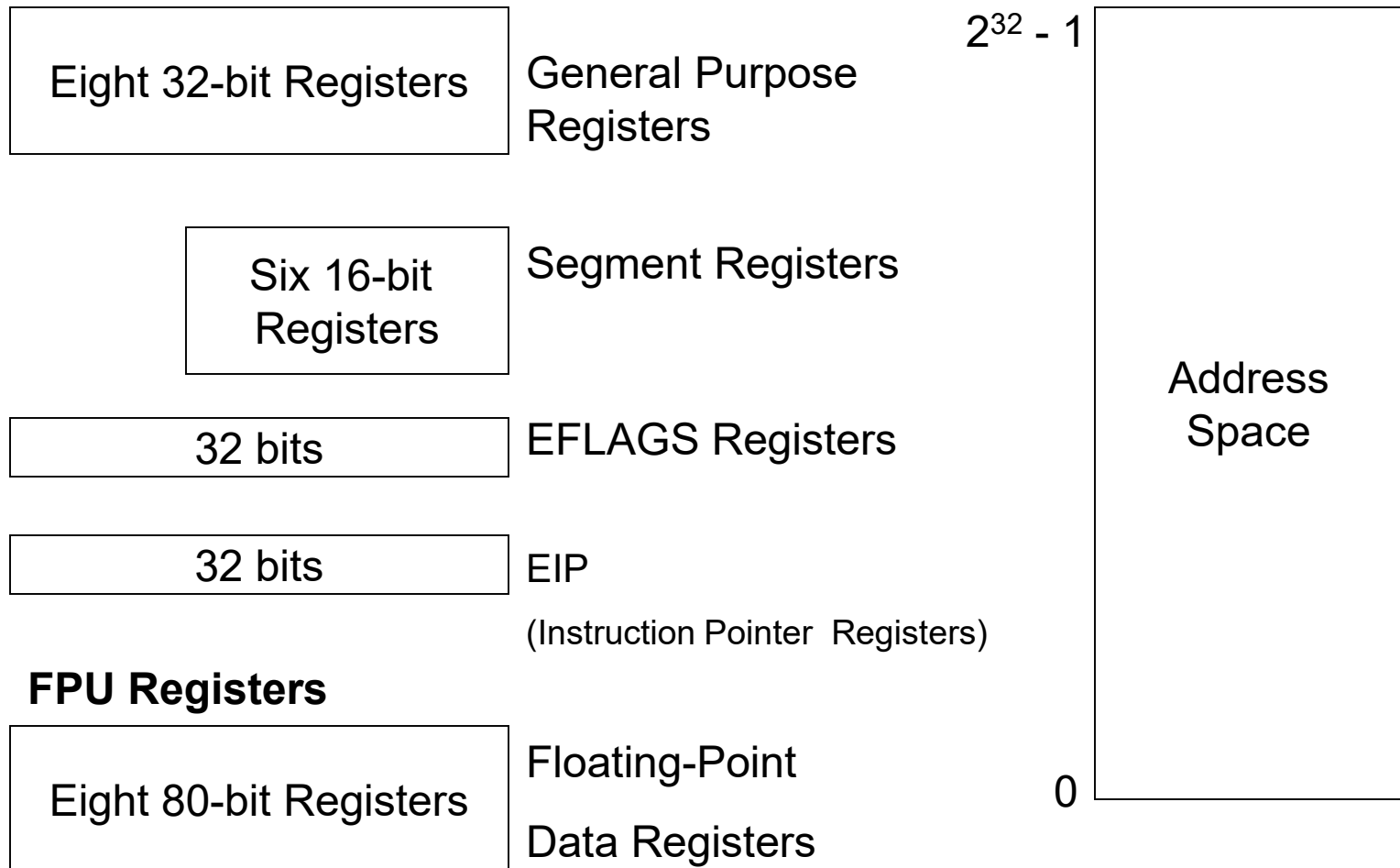
Code created for CPUs released in 1978 still executes on latest CPUs!

<b>CPU</b>	<b>Clock Frequency</b>	<b>Transistors per die</b>	<b>Address space</b>
<b>8086</b>	8 MHz	29 K	1 MB
<b>80486</b>	25 MHz	1.2 M	4 GB
<b>Pentium 4</b>	1.5 GHz	42 M	64 GB

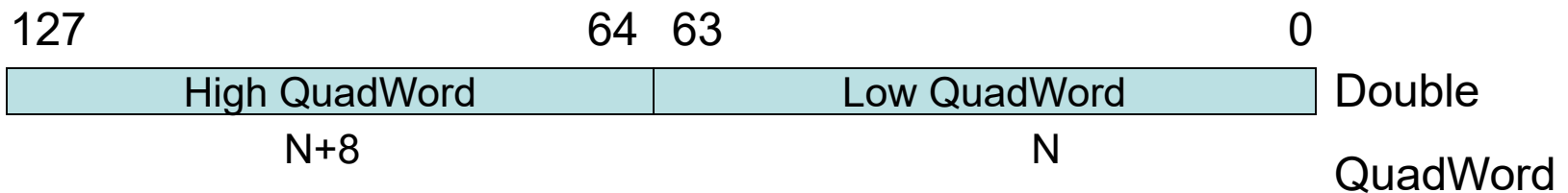
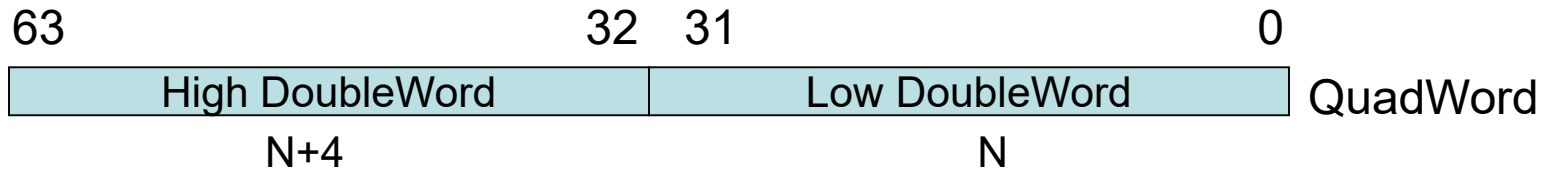
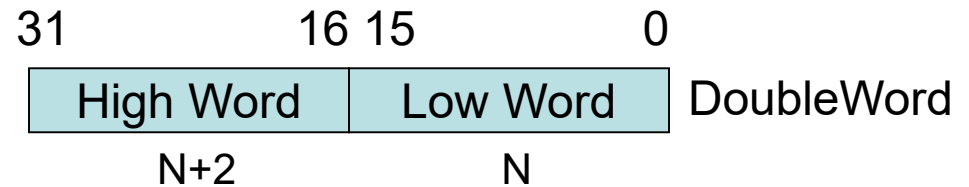
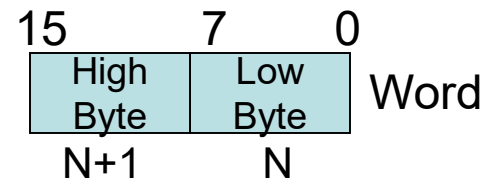
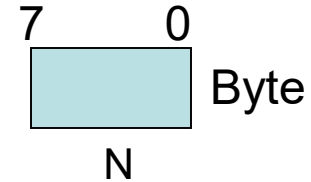
**Moore's Law (Named after Intel cofounder Gordon Moore):**

“The number of transistors that would be incorporated on a silicon die would double every 18 months for the next several years.”

# Intel CPU Architecture (32 bit)



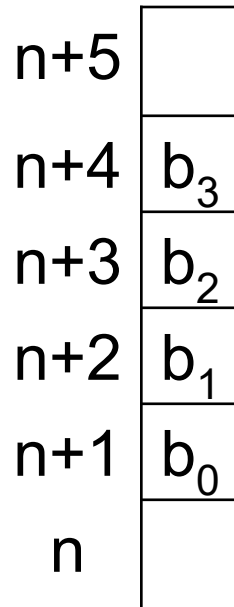
# Data Types



# Little/Big Endian (1)

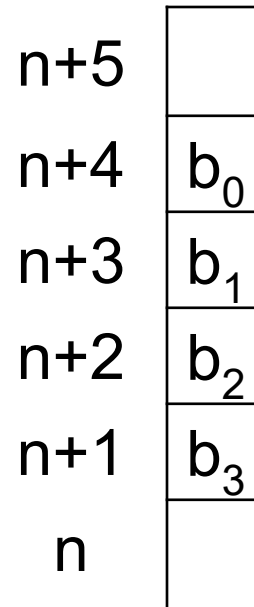
Different computer architectures order information in different ways.

$$X = b_3 * 2^{24} + b_2 * 2^{16} + b_1 * 2^8 + b_0 * 2^0$$



**Little Endian**

**(e.g. Intel x86)**



**Big Endian**

**(e.g. Sun SPARC)**

# Little/Big Endian (2)

The following C program tests if the machine it is running on has a little or big endian architecture.

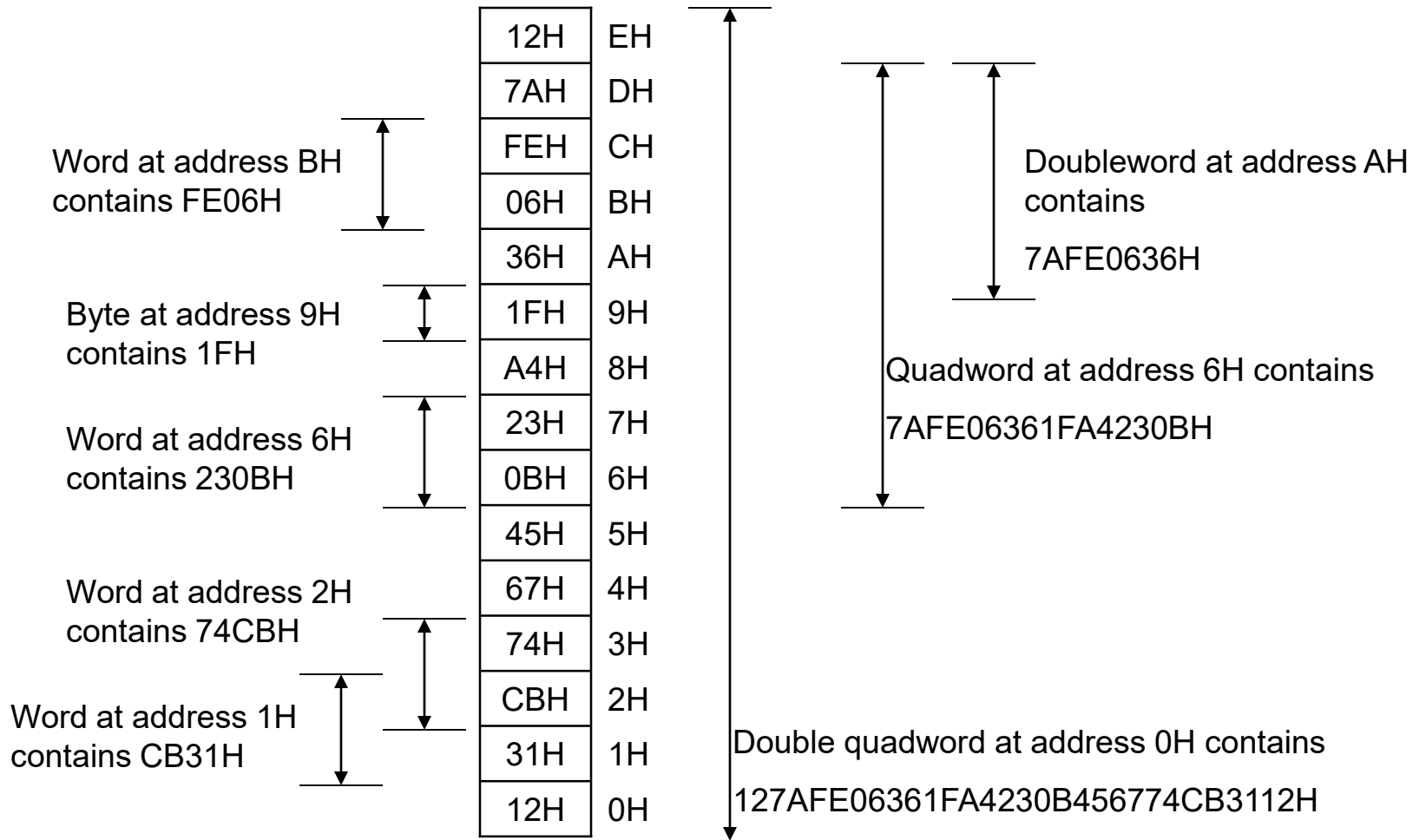
```
#include <stdio.h>

union {
    int x;
    char c[sizeof(int)];
} u;

void main()
{
    u.x = 1;
    if (u.c[0] == 1)
        printf("Little Endian\n");
    else
        printf("Big Endian\n");
}
```

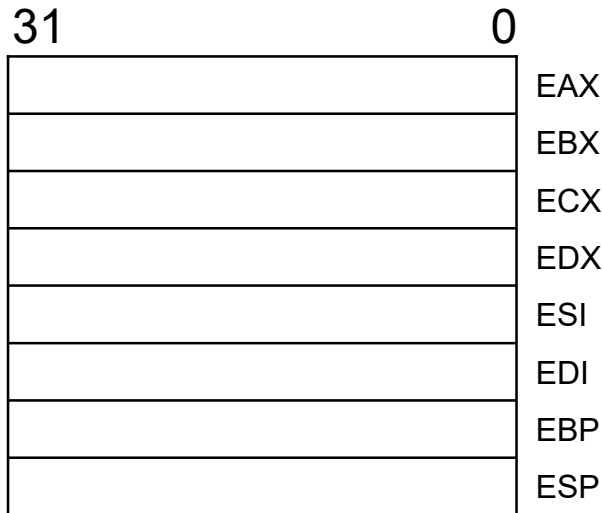


# Data Types



# Registers

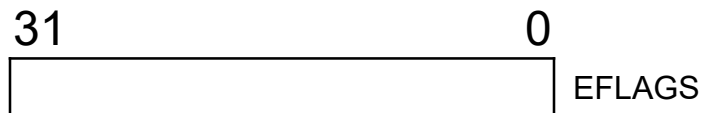
## General-Purpose Registers



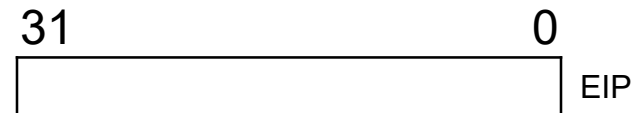
## Segment Registers



## Program status and control Register



## Instruction Register



- Registers are like variables of C, but there exist only a finite amount.
- We will look at Segment Registers later.

# General Purpose Registers

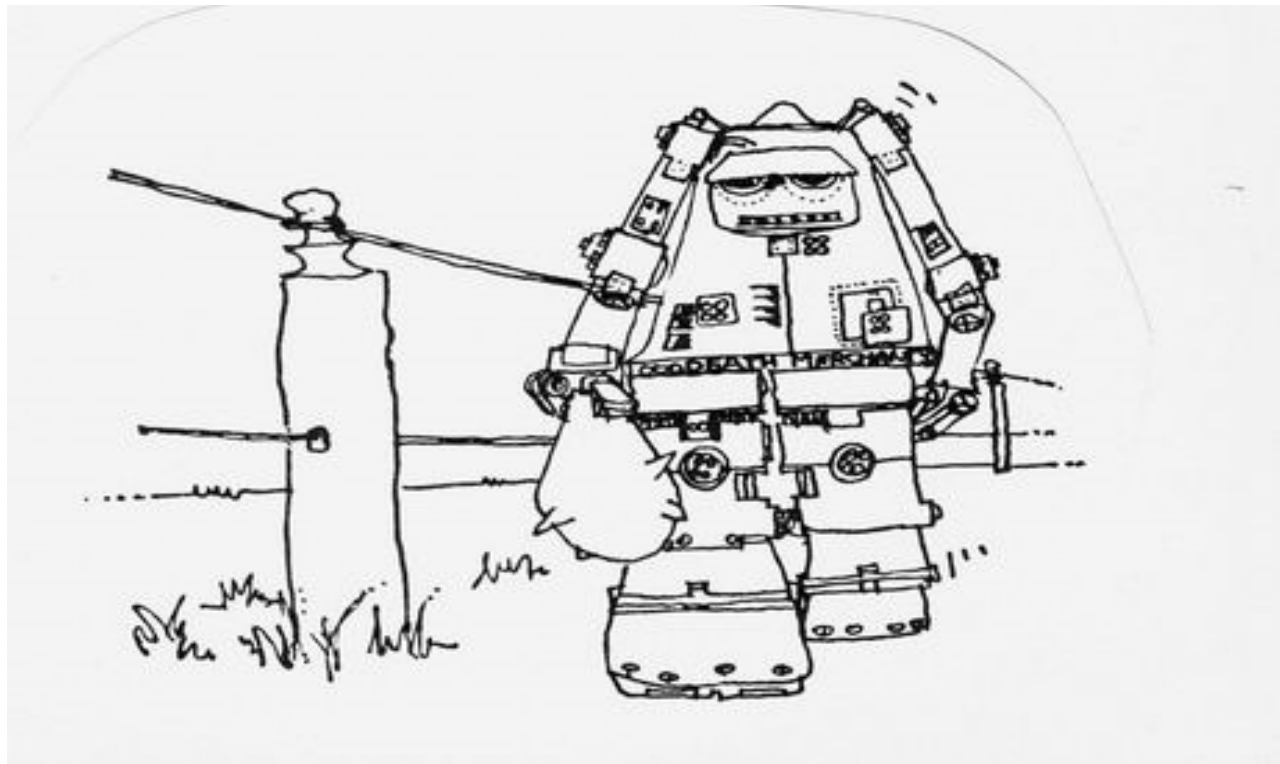
31	16	8	7	0
EAX		AH	AL	
		AX		
EBX		BH	BL	
		BX		
ECX		CH	CL	
		CX		
EDX		DH	DL	
		DX		
EBP		BP		
ESI		SI		
EDI		DI		
ESP		SP		

Some registers are only available for certain machine instructions.

# EFLAGS Register

- EFLAGS = Extended Flags
- 32 bit register, where each bit indicates a certain status
- Machine instructions such as ADD, SUB, MUL, DIV modify the EFLAGS Register.

Flag	Bit	Description
CF	0	Carry Flag: Indicates an overflow condition for unsigned integer arithmetic.
ZF	6	Zero Flag: Set if the result is zero; cleared otherwise.
SF	7	Sign Flag: Set equal to the most significant bit of the result.
OF	11	Overflow Flag: Set if the integer result is too large to fit in the destination operand.
IF	9	Interrupt Flag: If set, enables the recognition of external interrupts.



# *x86 Instruction Overview*

# Objectives

- Quick introduction to x86 assembly
- Provide examples for common use cases
- Show how C is mapped to assembly
- Show how to embed assembly into C-code

# Assembly Syntax

- **IMPORTANT:** all examples will use the 32-bit version of the x86 (TOS runs in 32-bit mode)
- Two major syntaxes for writing x86 assembly code:
- Intel format:
  - destination, source
  - Exists in boot loader (`tools/boot/*.s`)
- AT&T syntax:
  - source, destination
  - Produced by gcc, used in all class slides

# x86 Instruction Overview

## **Memory Operations**

- MOV - move data
- Push - push data onto stack
- Pop - Pop data off the stack

## **Logical and arithmetic operations**

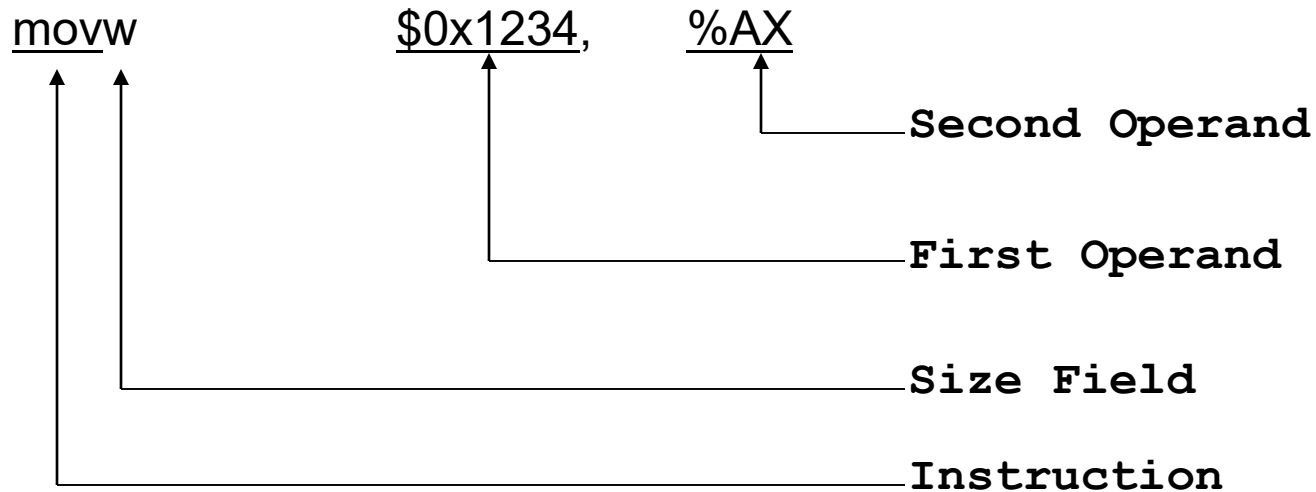
- AND - Bitwise and
- OR - Bitwise or
- XOR - Bitwise exclusive or
- ADD - Addition
- SUB - Subtraction

## **Control flow operatoins**

- JMP - Jump
- JZ - Jump if Zero
- JNZ - Jump if NOT Zero
- CALL - Call Subroutine
- RET - Return from subroutine



# Anatomy of a Move Instruction



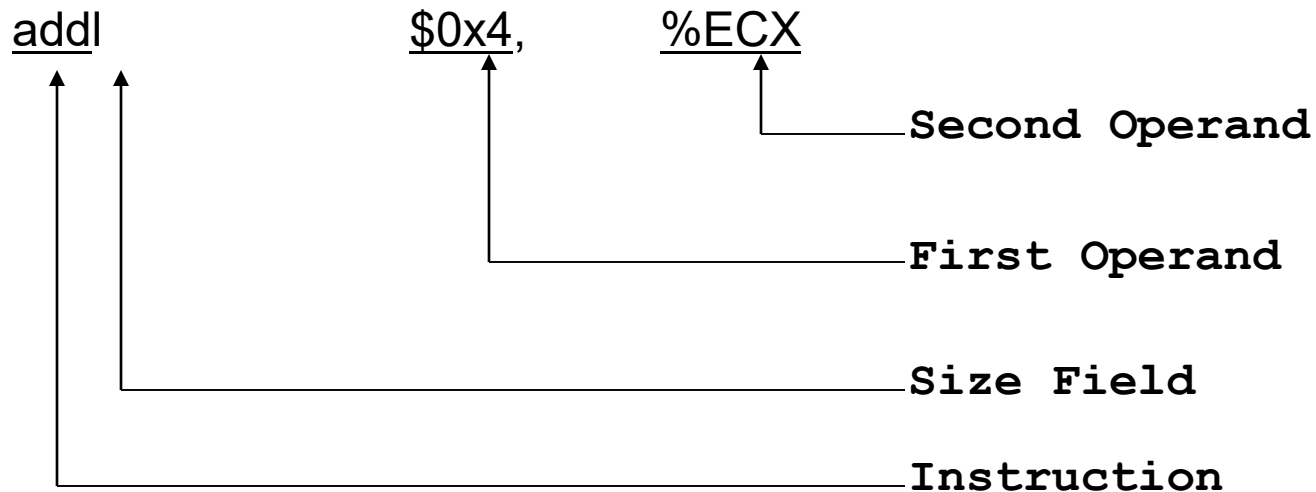
- This instruction will move the value 0x1234 into register %AX
- General format of move instruction: `mov src, dest`
- NOTE: We assume AT&T assembly syntax!

# Move Operations

Addr	Machine code	Assembly
0:	b0 12	mov \$0x12,%al
2:	b4 34	mov \$0x34,%ah
4:	66 b8 78 56	mov \$0x5678,%ax
8:	b4 ab	mov \$0xab,%ah
a:	bb ef be ad de	mov \$0xdeadbeef,%ebx
f:	66 89 d8	mov %bx,%ax
12:	88 e3	mov %ah,%bl

<b>EIP</b>	<b>EAX</b>	<b>EBX</b>
0	00000012	00000000
2	00003412	00000000
4	00005678	00000000
8	0000AB78	00000000
A	0000AB78	DEADBEEF
F	0000BEEF	DEADBEEF
12	0000BEEF	DEADBEFE

# Logic / Arithmetic Instructions (1)



- Adds 4 to the value in register `%ECX`
- Second operand (`%ECX` in this example) is also the destination
- `ADD`, `SUB` for arithmetic
- `AND`, `OR`, `XOR` for Boolean logic

# Logical / Arithmetic Instructions (2)

Addr	Machine code	Assembly
0:	66 b8 20 00	mov \$0x20,%ax
4:	66 bb 0a 00	mov \$0xa,%bx
8:	66 01 d8	add %bx,%ax
b:	66 0d 00 34	or \$0x3400,%ax
f:	66 25 00 ff	and \$0xff00,%ax
13:	66 31 db	xor %bx,%bx
16:	66 43	inc %bx

<b>EIP</b>	<b>AX</b>	<b>BX</b>
0	0020	0000
4	0020	000A
8	002A	000A
B	342A	000A
F	3400	000A
13	3400	0000
16	3400	0001

# Jump Instructions

- Jump instruction changes %EIP to modify flow of control
  - Used to implement if statements and loops
- Target of a jump is the address of an instruction (like a C pointer-to-function)
- Assembler labels reference an address
- Instructions:
  - JMP (unconditional jump)
  - JZ (Jump if Zero)
  - JNZ (Jump if Not Zero)

# EFLAGS

Addr	Machine code	Assembly
0:	66 31 c9	xor %cx,%cx
3:	66 b8 03 00	mov \$0x3,%ax
7:	66 01 c1	L1: add %ax,%cx
a:	66 48	dec %ax
c:	75 f9	jnz L1
e:	66 89 c8	mov %cx,%ax

<i>EIP</i>	<i>AX</i>	<i>CX</i>	<i>Z-Flag</i>
0	-	0000	1
3	0003	0000	1
7	0003	0003	0
A	0002	0003	0
C	0002	0003	0
7	0002	0005	0
A	0001	0005	0
C	0001	0005	0
7	0001	0006	0
A	0000	0006	1
C	0000	0006	1
E	0006	0006	1

# Indirect Addressing

- Assembly equivalent of dereferencing a pointer
- General format:  
offset(register)
- Examples:  
  (%ecx)  
  4 (%esp)

# Indirect Addressing

Addr	Machine code	Assembly
0:	b8 00 80 0b 00	mov \$0xb8000,%eax
5:	66 bb 34 12	mov \$0x41,%bl
9:	66 89 18	mov %bl,(%eax)

*Before last mov-instr.*

0xb8003	00
0xb8002	00
0xb8001	00
0xb8000	00

*After last mov-instr.*

0xb8003	
0xb8002	
0xb8001	
0xb8000	41

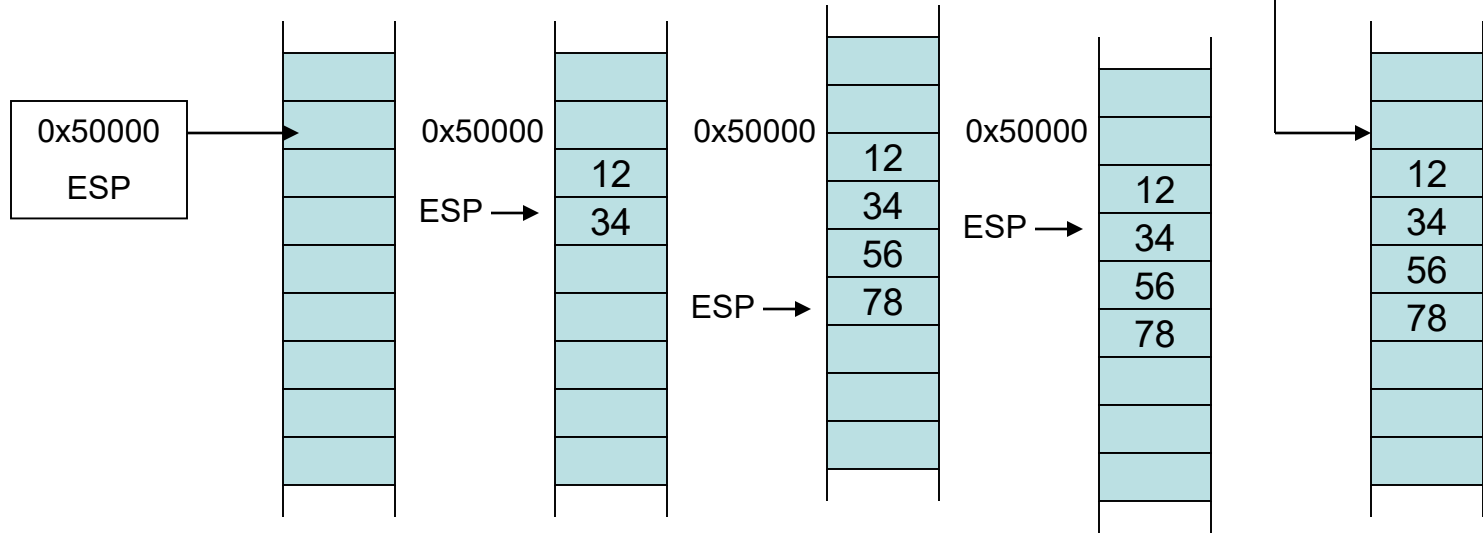
Equivalent to the following C-code:

```
char* screen_base = (char *) 0xb8000;  
*screen_base = 'A';
```



# Pushing and Popping

Addr	Machine code	Assembly
0:	bc 00 00 05 00	mov \$0x50000,%esp
5:	66 b8 34 12	mov \$0x1234,%ax
9:	66 bb 78 56	mov \$0x5678,%bx
d:	66 50	push %ax
f:	66 53	push %bx
11:	66 58	pop %ax
13:	66 5b	pop %bx

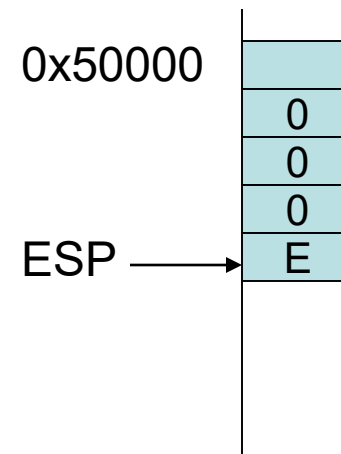


<b>EIP</b>	0	0xD	0xF	0x11	0x13
<b>ESP</b>	0x50000	0x4FFFE	0x4FFFC	0x4FFFE	0x50000
<b>AX</b>	-	0x1234	0x1234	0x5678	0x5678
<b>BX</b>	-	0x5678	0x5678	0x5678	0x1234

# Subroutines

Addr	Machine code	Assembly
0:	bc 00 00 05 00	mov \$0x50000,%esp
5:	66 b8 34 12	mov \$0x1234,%ax
9:	e8 02 00 00 00	call L2
e:	eb fe	L1: jmp L1
10:	66 40	L2: inc %ax
12:	c3	ret

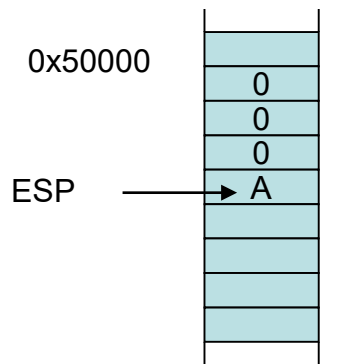
EIP	ESP	AX
0	0x50000	-
5	0x50000	0x1234
9	0x4FFFC	0x1234
0x10	0x4FFFC	0x1235
0x12	0x50000	0x1235
0xE	0x50000	0x1235



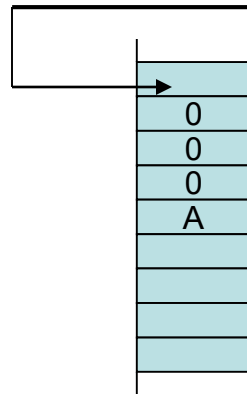
# Subroutines

Addr	Machine code	Assembly
0:	bc 00 00 05 00	mov \$0x50000,%esp
5:	e8 06 00 00 00	call L2
a:	66 b8 34 12	mov \$0x1234,%ax
e:	eb fe	L1: jmp L1
10:	58	L2: pop %eax
11:	83 c0 04	add \$0x4,%eax
14:	50	push %eax
15:	c3	ret

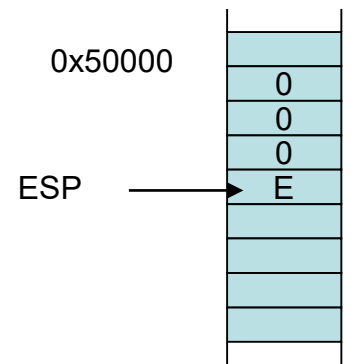
0x50000  
ESP



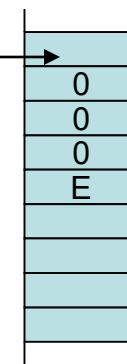
EIP 0x5  
ESP 0x4FFFC  
EAX -



EIP 0x10  
ESP 0x50000  
EAX 0xA



EIP 0x14  
ESP 0x4FFFC  
EAX 0xE



EIP 0xE  
ESP 0x50000  
EAX 0xE

# C and Assembly

```
int add (int x, int y)
{
    return x + y;
}

void main()
{
    int sum = add (3, 4);
    printf ("3 + 4 = %d\n", sum);
}
```

```
add:
    movl    8(%esp), %eax
    addl    4(%esp), %eax
    ret

.LC0:
    .string "3 + 4 = %d\n"

main:
    subl    $20, %esp
    pushl   $4
    pushl   $3
    call    add
    addl    $8, %esp
    pushl   %eax
    pushl   $.LC0
    call    printf
    addl    $28, %esp
    ret
```

**Compile with: gcc -fomit-frame-pointer -O1 -S add.c**

# C and Assembly

## Observations:

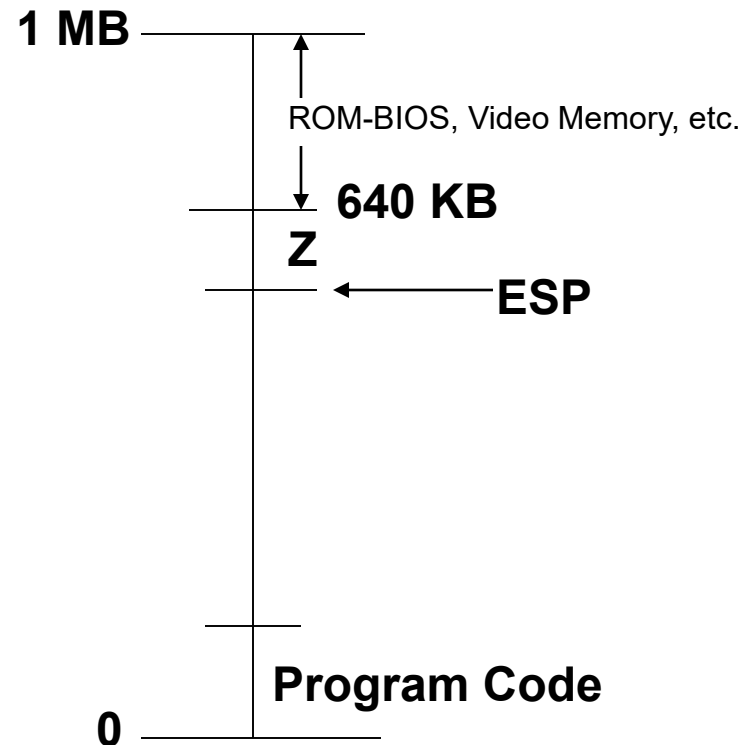
- C-functions are called via the x86 `call` instruction.
- The caller pushes the actual parameters onto the stack.
- The actual parameters are pushed from right to left.
- The callee accesses the parameters as offset to the current value of the `%ESP` register.
- After the function call returns, the caller has to clean up the stack.
- `printf()` and `main()` are treated just like any other function.
- Return values are placed in `%EAX`

# Memory Layout

- When running a program under a modern OS, memory is divided into code, heap, stack
- This environment is set up by the OS, when writing the OS we don't have such an environment.
- Memory layout managed manually by the OS

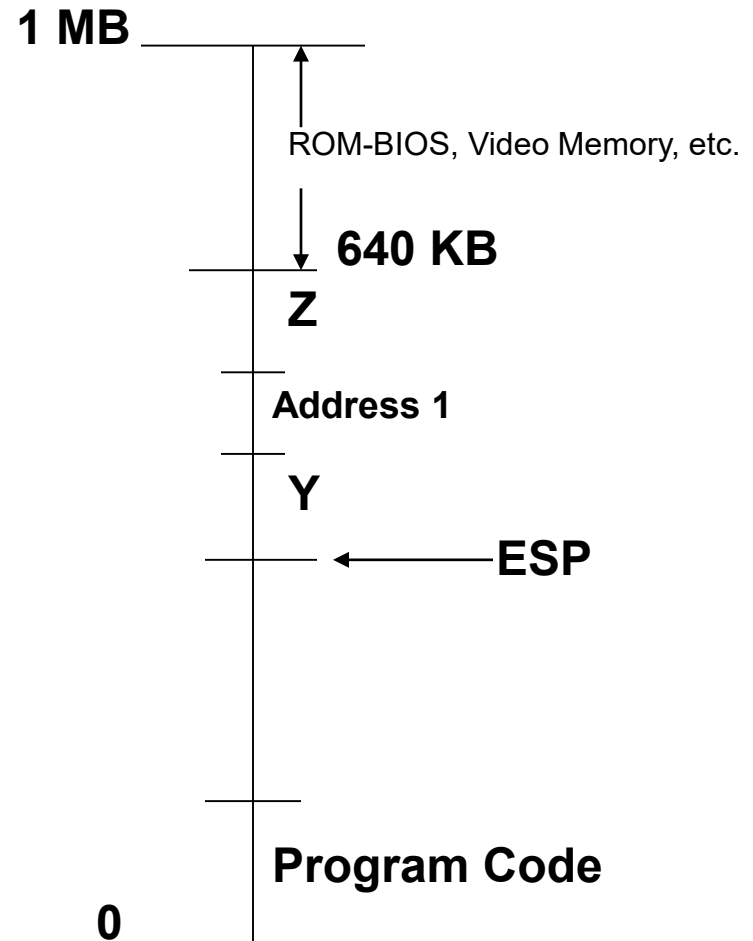
# Stack Layout (1)

```
void f()  
{  
    int x;  
}  
  
void g()  
{  
    int y;  
    f(); /* Address 2 */  
}  
  
void main()  
{  
    int z;  
    g(); /* Address 1 */  
}
```



# Stack Layout (2)

```
void f()  
{  
    int x;  
}  
  
void g()  
{  
    int y;  
    f(); /* Address 2 */  
}  
  
void main()  
{  
    int z;  
    g(); /* Address 1 */  
}
```



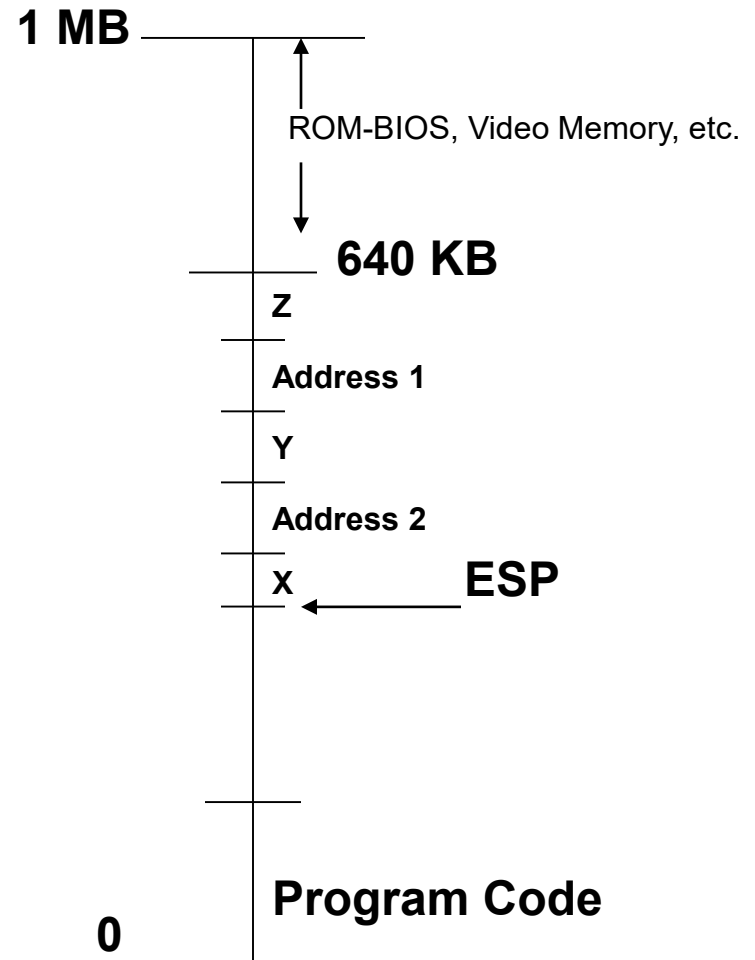


# Stack Layout (3)

```
void f()  
{  
  → int x;  
}
```

```
void g()  
{  
  int y;  
  f(); /* Address 2 */  
}
```

```
void main()  
{  
  int z;  
  g(); /* Address 1 */  
}
```



# Embedding Assembly into C

## test.c

```
void enable_interrupts()
{
    asm( "sti" );
}
```

## test.s

```
enable_interrupts:
    #APP
        sti
    #NO_APP
    ret
```

- Assembly instructions can be embedded anywhere C-statements are allowed.
- This is done with the `asm`-instruction (gcc-specific!)
- Application specific assembly is surrounded by `#APP` and `#NO_APP`.
- Note: `sti` – instruction for enabling the interrupts.

# Embedding Assembly in C

```
int add (int x, int y)
{
    int sum = x;
    asm ("add %1, %0" : "=r"
        (sum) : "m" (y) );
    return sum;
}
```

```
add:    subl    $4, %esp
        movl   8(%esp), %eax
        movl   %eax, (%esp)
#APP
        add   12(%esp), %eax
#NO_APP
        addl  $4, %esp
        ret
```

# EFLAGS - Revisited

Addr	Machine code	Assembly
0:	31 c9	xor %ecx,%ecx
2:	b8 03 00 00 00	mov \$0x3,%eax
7:	01 c1	L1: add %eax,%ecx
9:	ff c8	dec %eax
b:	75 fa	jnz L1
d:	89 c8	mov %ecx,%eax
f:	c3	ret

This example is mostly identical from an earlier slide entitled “EFLAGS”. Differences are that it uses 32-bit registers and is assembled for a 64-bit architecture (note that TOS is compiled for 32-bit architectures). This subroutine will compute the sum of  $1 + 2 + 3 = 6$  which is stored in %eax.

# Just In Time (JIT)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>

void main() {
    unsigned char code[] = {
        0x31, 0xC9, 0xB8, 0x03, 0x00, 0x00, 0x00, 0x01,
        0xC1, 0xFF, 0xC8, 0x75, 0xFA, 0x89, 0xC8, 0xC3 };
    void* mem = mmap(NULL, sizeof(code), PROT_WRITE | PROT_EXEC,
                    MAP_ANON | MAP_PRIVATE, -1, 0);
    memcpy(mem, code, sizeof(code));
    int (*func)() = mem; int result = func();
    printf("Result: %d\n", result);
}
```

- This code needs to be compiled on a Linux 64-bit x86 machine. It will print 6.

# Booting TOS

- When the computer is turned on, several things must happen:
  - TOS kernel gets loaded into memory
  - Execution stack (%esp) is established
  - Kernel starts by jumping to `kernel_main()`
- For a regular program, this process is done by the *OS loader*.
- In the case of the kernel, we have no OS so this work is done by the *boot loader*.

# Booting TOS

- Due to constraints of the x86 architecture, booting TOS happens in two stages.
- Code for the two boot loader stages is in:
  - `tos/tools/boot/boot.s`
  - `tos/tools/boot/second-stage.s`
- This code is a mess, you don't need to understand it!
- But, it is important to understand the environment that the boot loader sets up for the kernel!

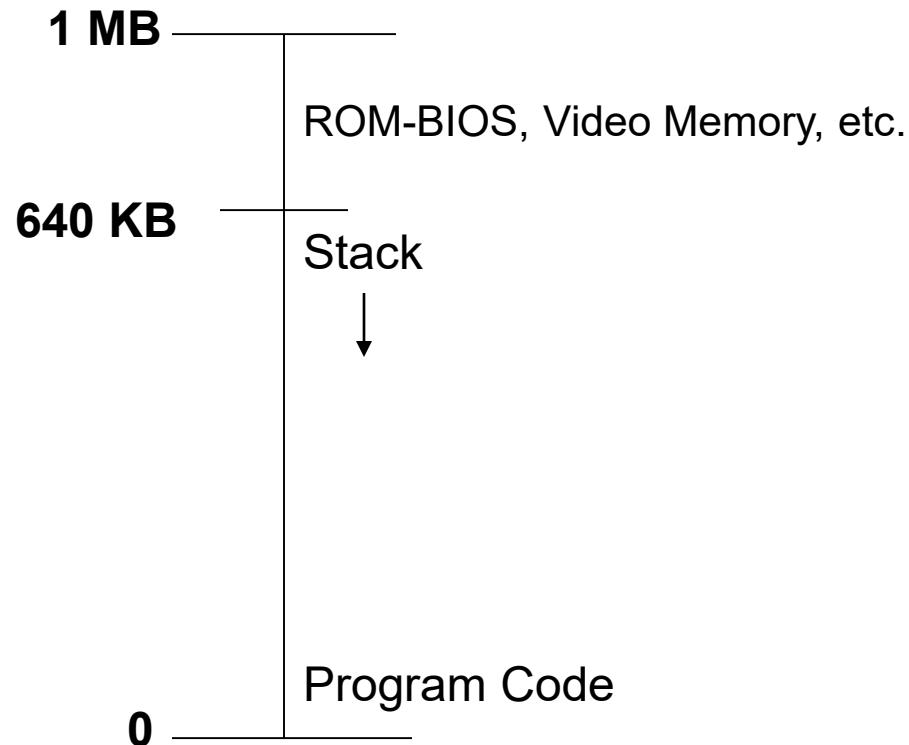
# Memory Layout

- The memory visible to a “regular” program is divided into code, heap, and stack
- For a “regular” program, the OS (kernel) can do things such as:
  - Prevent the program from modifying its code
  - Allow the heap to grow (i.e., as a result of a call to `malloc()`)
- But, we are writing the kernel so we don't have these conveniences!



# TOS Memory Layout

- No heap in TOS, just code and stack
- The only usable addresses are 0-640KB



# Hardware Protection

- OS needs to protect applications from each other
  - Want protection from malicious programs as well as from programs that are just buggy
- We will discuss the details of implementing protection for specific hardware resources (memory, CPU, etc.) throughout the semester.

# Hardware Protection

- Regardless of the hardware, the kernel needs to run with more “privileges” than other programs.
- Modern hardware can switch between two modes with different privileges:
  - Kernel mode, in which the processor may do anything
  - User mode, in which some operations are restricted

# (Lack of) Protection in TOS

- Protection is complex in the x86 architecture.
- We will not implement protection in TOS (a malicious or badly-written program can crash the OS, interfere with other programs, etc.)
- Nevertheless, we will study how protection is implemented in “real” operating systems.