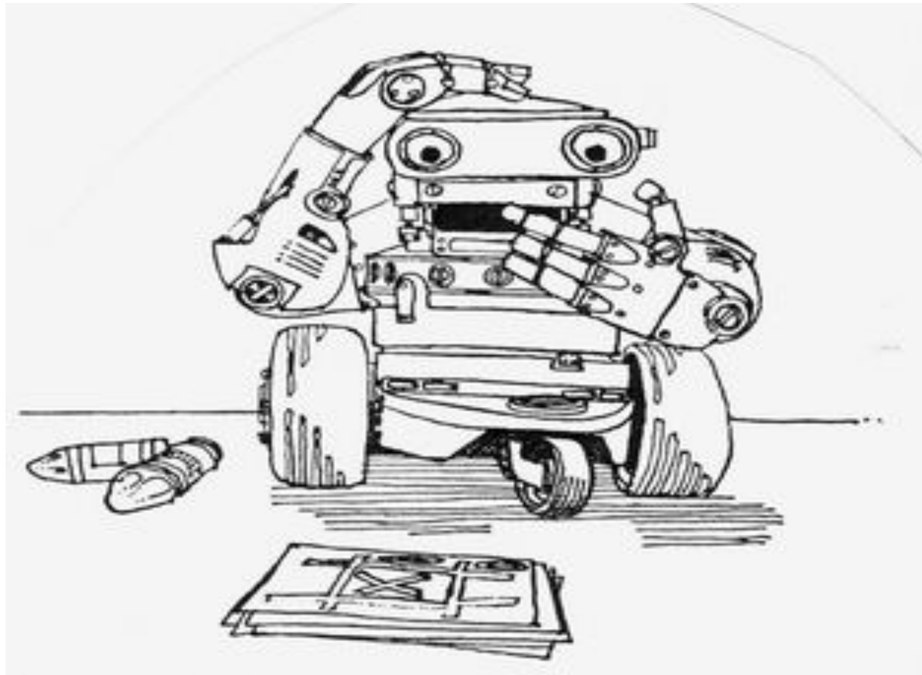


Intel Architecture

Objectives

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

History

8086 (1978)	- 16 bit registers - 20 bit addressing (1MB address space)
80286 (1982)	- Protected mode - 24 bit addressing (16 MB address space) - various memory protection mechanisms
80386 (1985)	- 32 bit registers - introduced paging - 32 bit addressing bus (4 GB address space)
80486 (1989)	- Parallel execution capability
Pentium (1993)	- Increased performance
P6 (1993)	- Increased performance

History

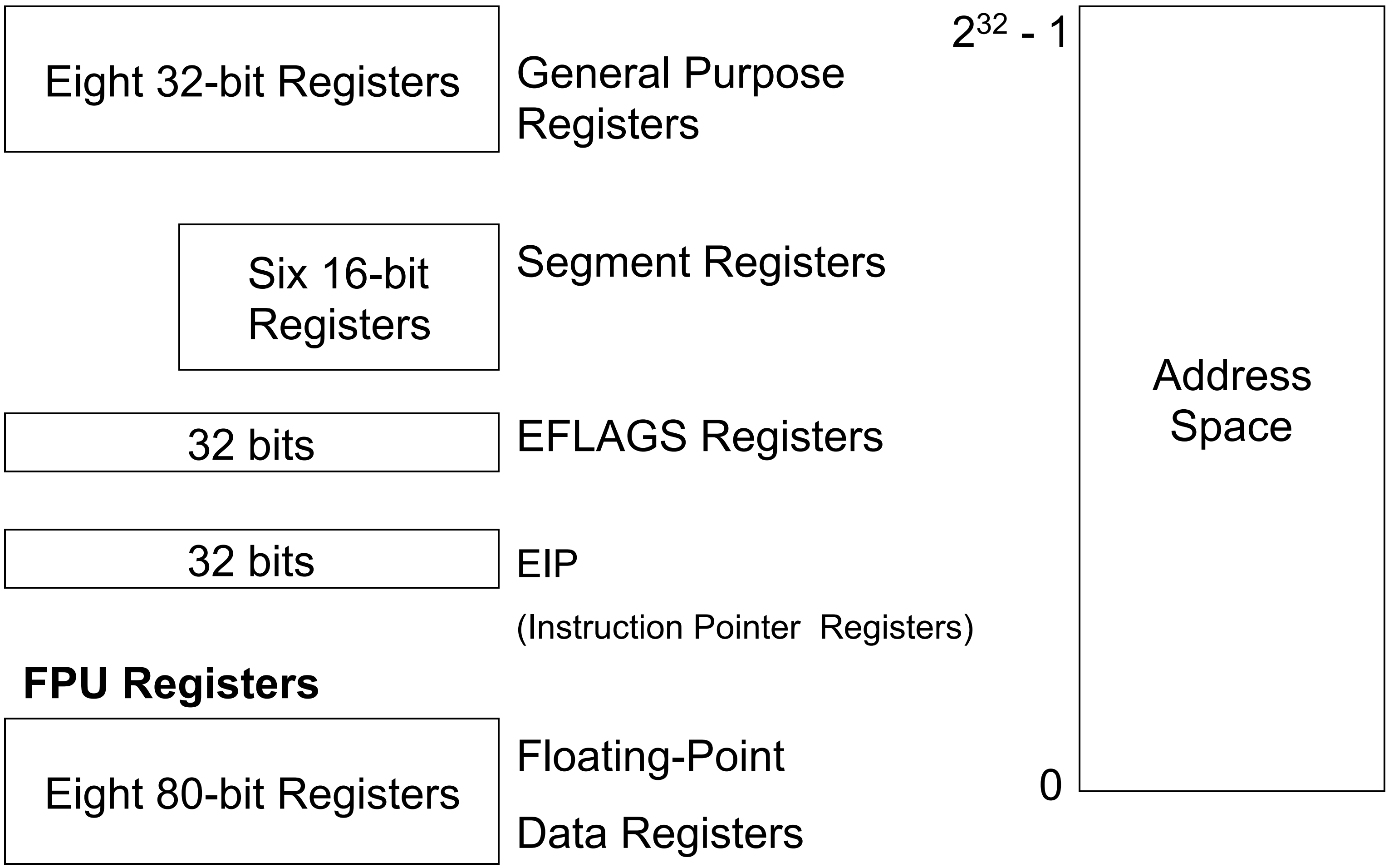
Code created for CPUs released in 1978 still executes on latest CPUs!

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

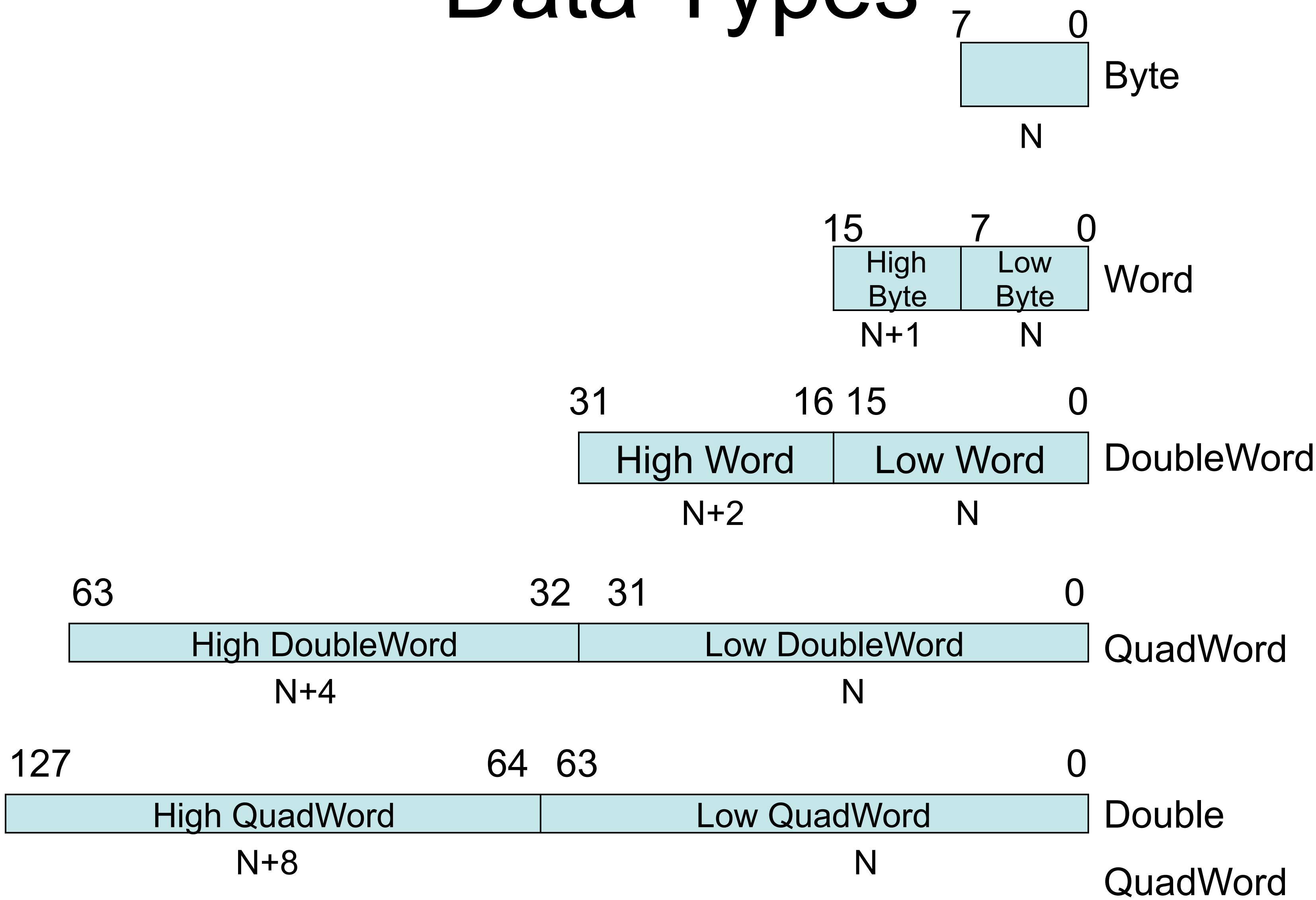
Moore's Law (Named after Intel cofounder Gordon Moore):

"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



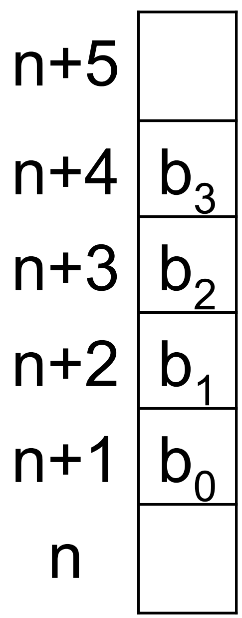
Data Types


7
0
Byte
N
15
7
0
High Byte
Low Byte
Word
N+1
N
31
16
15
0
High Word
Low Word
DoubleWord
N+2
N
63
32
31
0
High DoubleWord
Low DoubleWord
QuadWord
N+4
N
127
64
63
0
High QuadWord
Low QuadWord
Double QuadWord
N+8
N

Little/Big Endian (1)

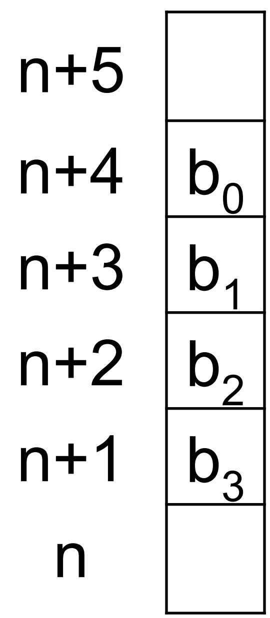
Different computer architectures order information in different ways.

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



Little Endian

(e.g. Intel x86)



Big Endian

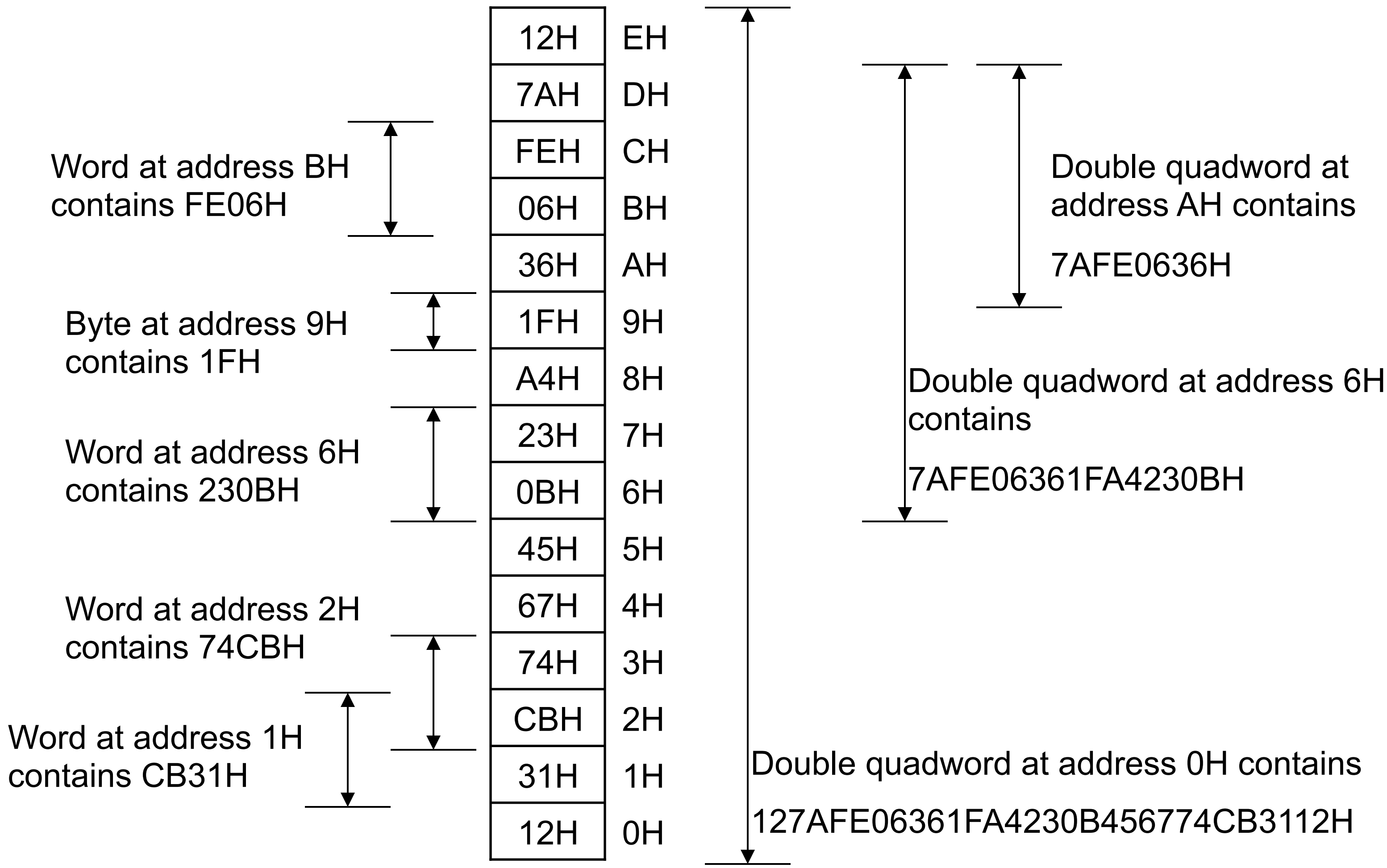
(e.g. Sun SPARC)

Little/Big Endian (2)

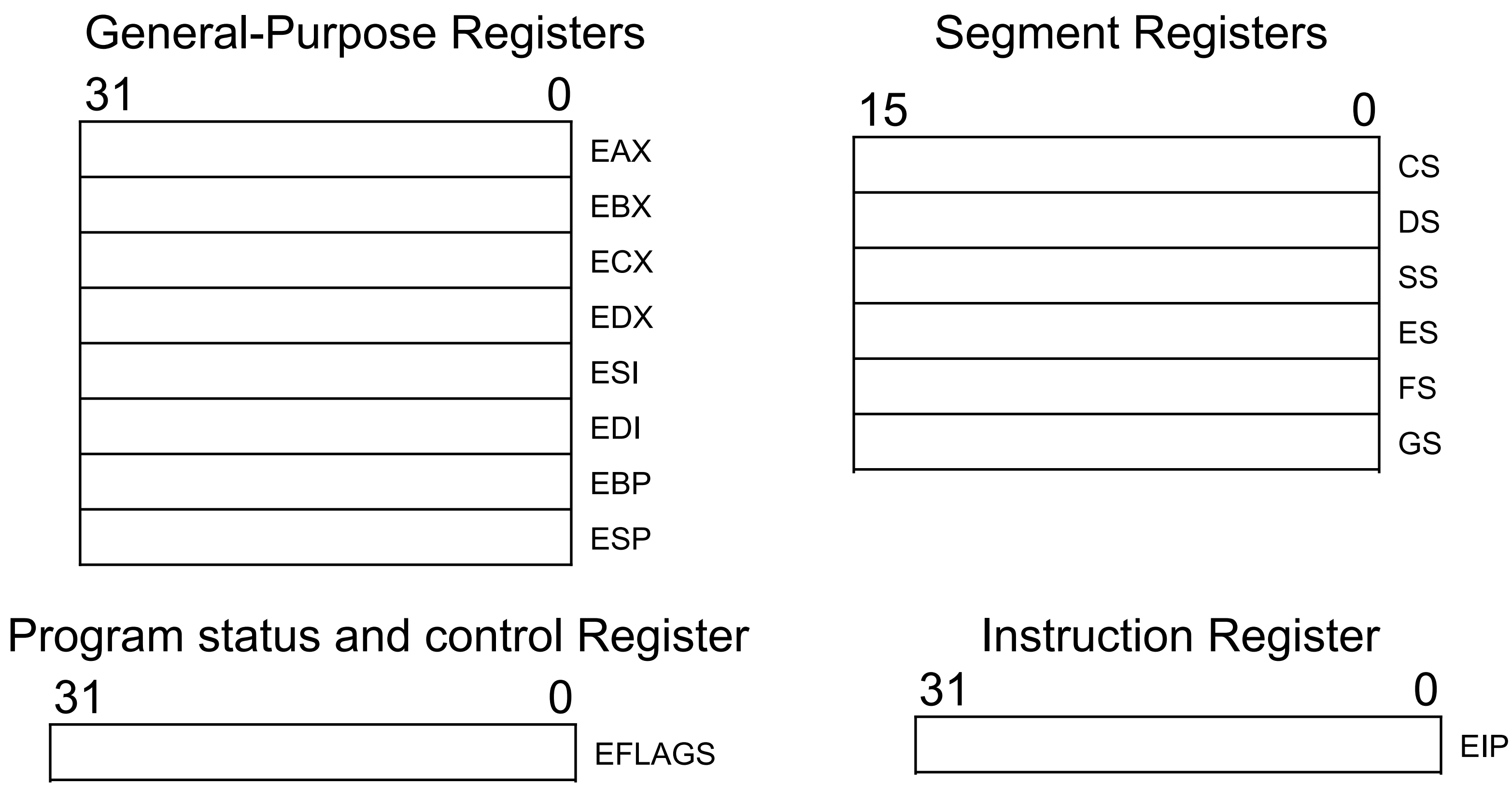
The following C program tests if the machine it is running on has a little or big endian architecture.

```
#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


12H EH
7AH DH
FEH CH
06H BH
36H AH
1FH 9H
A4H 8H
23H 7H
0BH 6H
45H 5H
67H 4H
74H 3H
CBH 2H
31H 1H
12H 0H
Word at address BH contains FE06H
Byte at address 9H contains 1FH
Word at address 6H contains 230BH
Word at address 2H contains 74CBH
Word at address 1H contains CB31H
Double quadword at address AH contains
7AFE0636H
Double quadword at address 6H contains
7AFE06361FA4230BH
Double quadword at address 0H contains
127AFE06361FA4230B456774CB3112H

Registers



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

General Purpose Registers

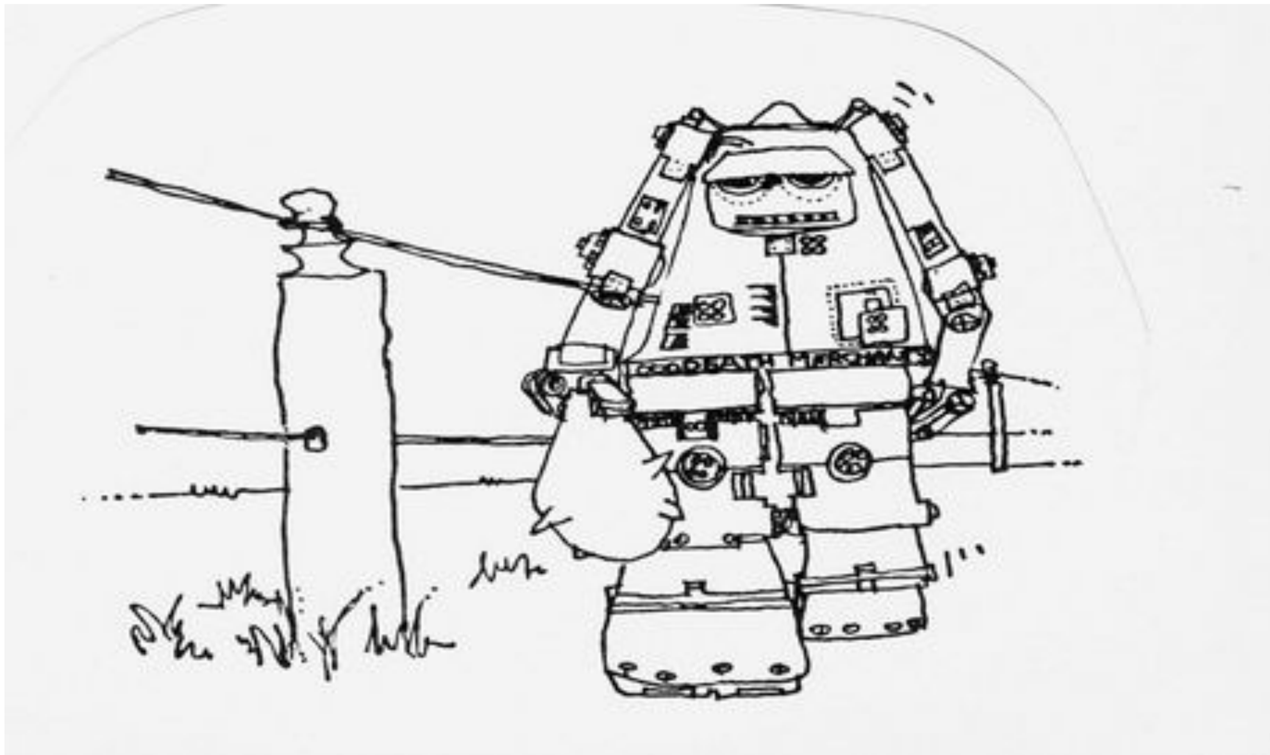
31 16	8	7 0
EAX	AH	AL
	AX	
EBX	BH	BL
	BX	
ECX	CH	CL
	CX	
EDX	DH	DL
	DX	
EBP	BP	
ESI	SI	
EDI	GI	
ESP	SP	

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

- 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

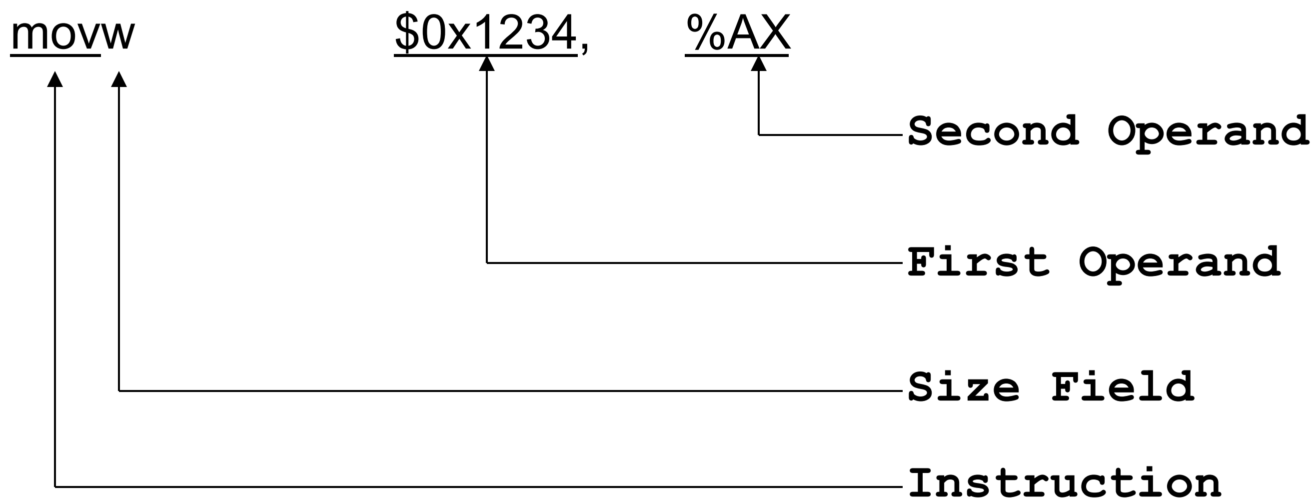
Logical and arithmetic operations

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



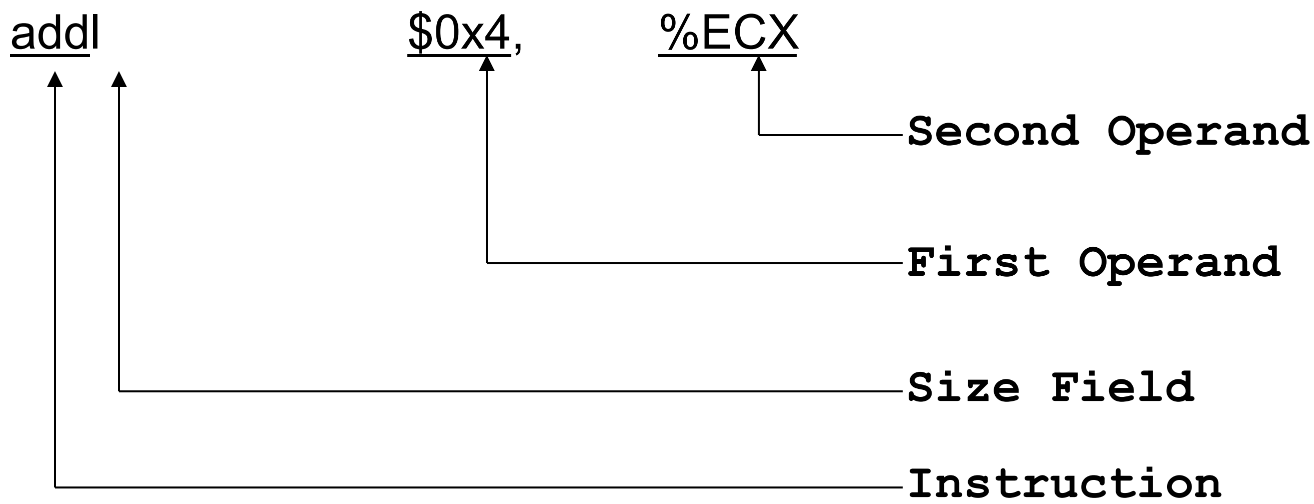
- This instruction will move the value 0x1234 into register %AX
- General format of move instruction: mov src, dest
- NOTE: We assume AT&T assembly syntax!

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
```

EIP	**EAX**	**EBX**
0	00000012	00000000
2	00003412	00000000
4	00005678	00000000
8	0000AB78	00000000
A	0000AB78	DEADBEEF
F	0000BEEF	DEADBEEF
12	0000BEEF	DEADBEBE

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
```

EIP	AX	BX
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
```

EIP	***AX***	***CX***	***Z-Flag***
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.

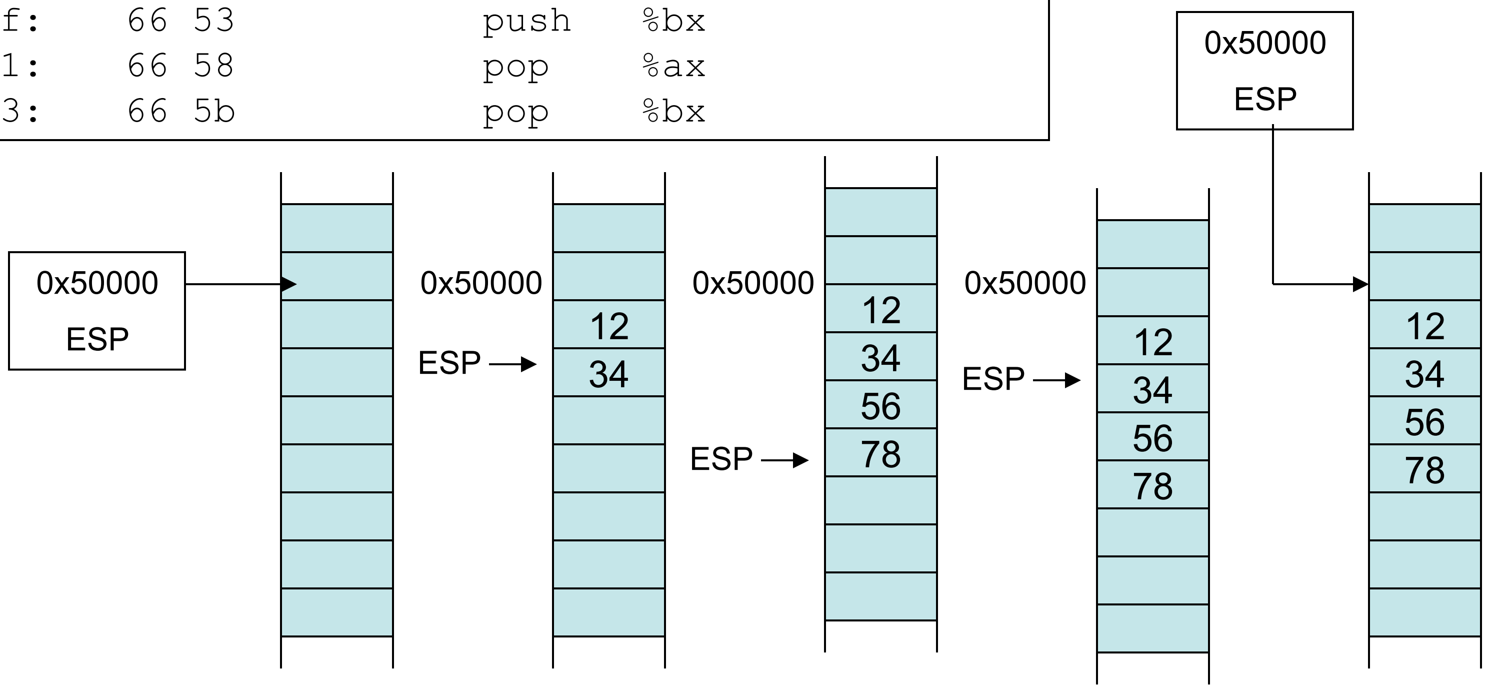
0xb8003	
0xb8002	
0xb8001	
0xb8000	41

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
```

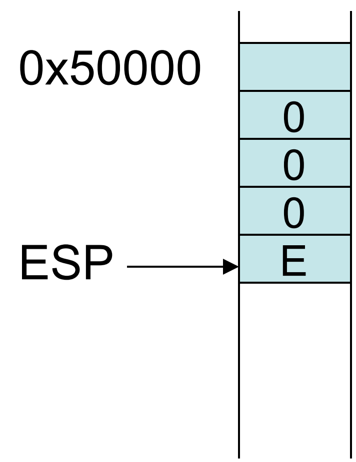


EIP	0	0xD	0XF	0x11	0X13
ESP	0x50000	0x4FFFE	0x4FFFC	0x4FFFE	0x50000
AX	-	0x1234	0x1234	0x5678	0x5678
BX	-	0x5678	0x5678	0x5678	0x1234

Subroutines

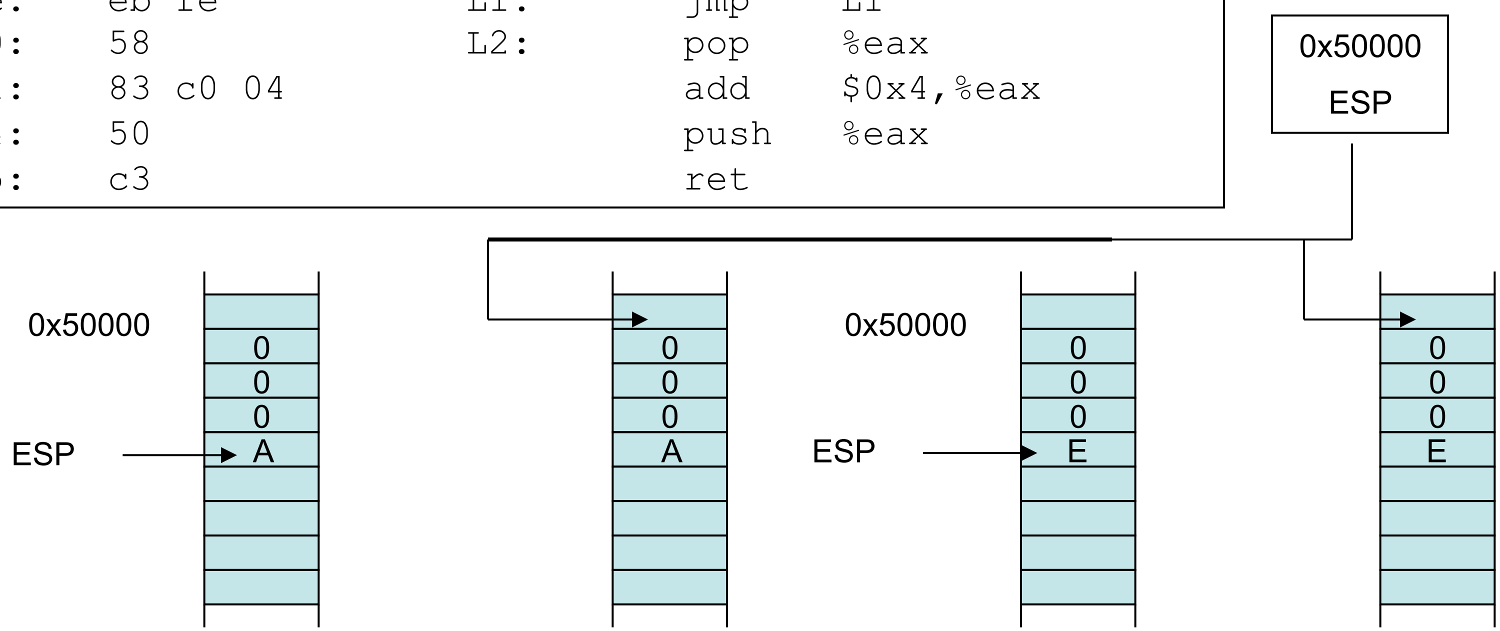
```
Addr     Machine code     Assembly
--------------------------------------------------
  0:     bc 00 00 05 00                  mov    $0x50000,%esp
  5:     66 b8 34 12                     mov    $0x1234,%ax
  9:     e8 02 00 00 00                  call   L2
  e:     eb fe              L1:      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

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



EIP	0x10	0x11	0x14	0xE
ESP	0x4FFFC	0x50000	0x4FFFC	0x50000
EAX	-	0xA	0xE	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);
}
```

Compile with: gcc –fomit-frame-pointer -01 –S add.c

```
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
```

C and Assembly

Observations:

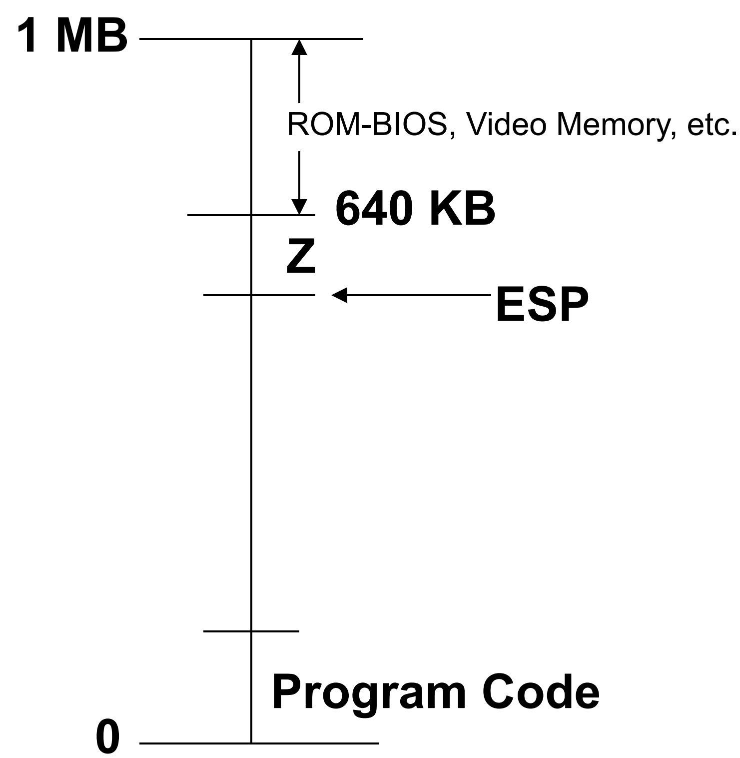
- C-functions are called via the x86 call instruction.
- The caller pushes the actual parameters onto the stack.
- The actual parameters are pushed from right to left.
- The callee accesses the parameters as offset to the current %ESP.
- After the function call returns, the caller has to clean up the 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

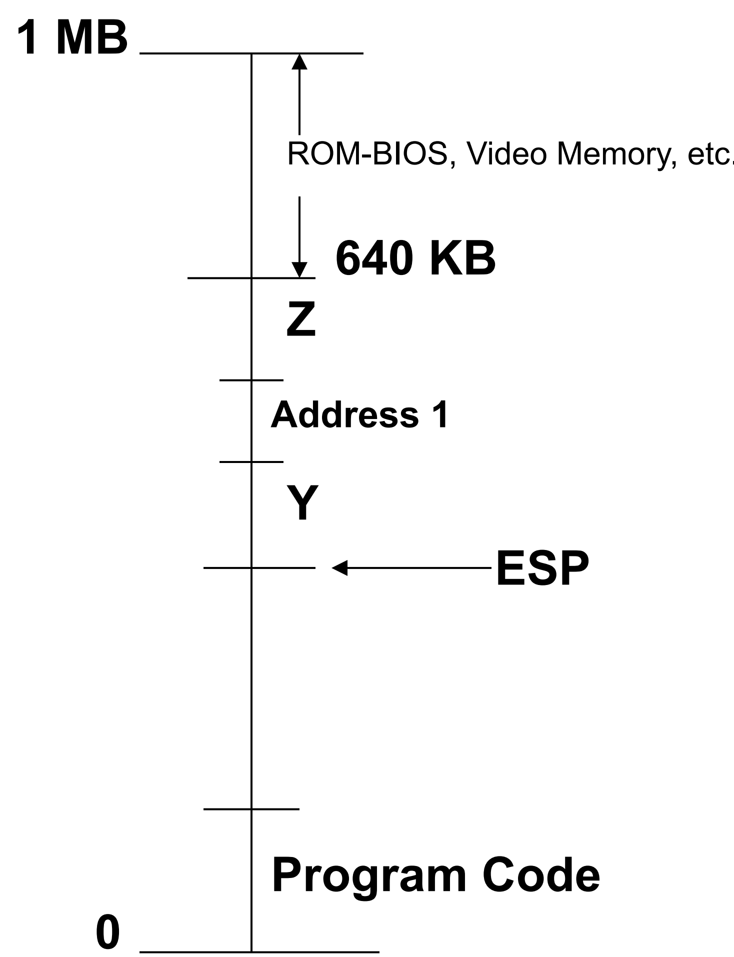
Stack Layout (1)

```
void f()
{
     int x;
}

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



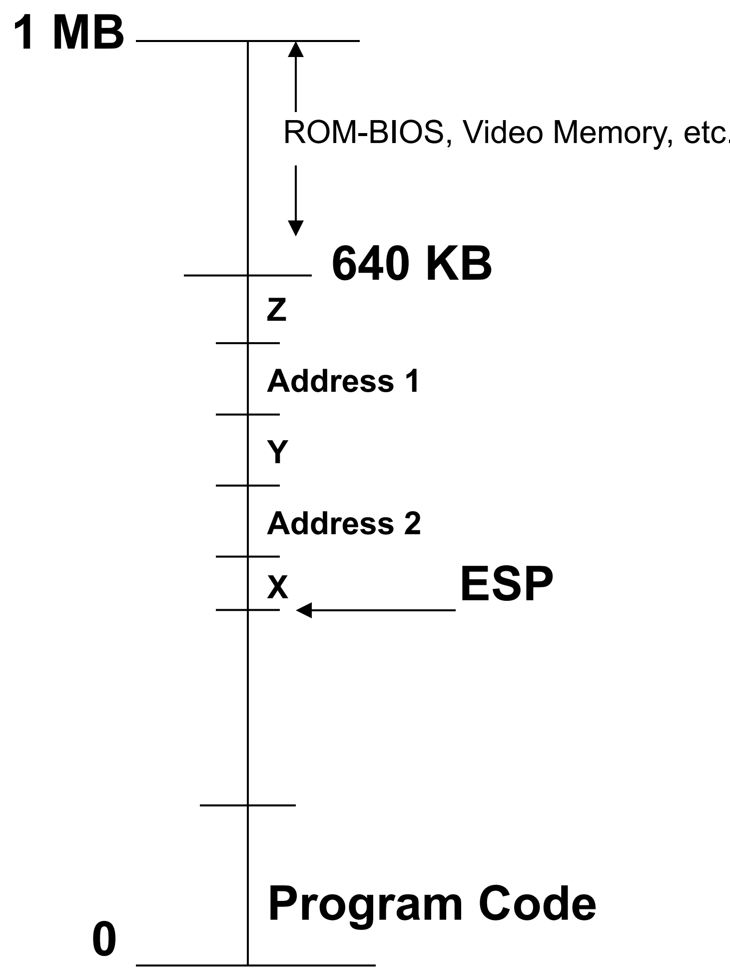
Stack Layout (2)

```
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
```

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

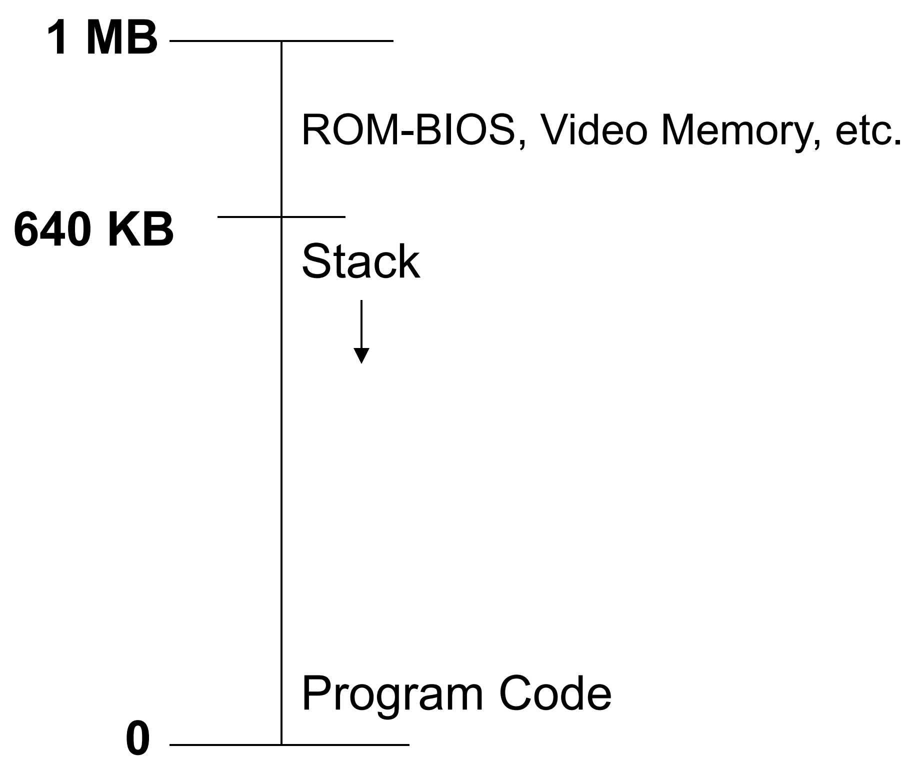
- Due to constraints of the x86 architecture, booting TOS happens in two stages.
- Code for the two boot loader stages is in:
  - `tos/tools/boot/boot.s`
  - `tos/tools/boot/second-stage.s`
- This code is a mess, you don’t need to understand it!
- But, it is important to understand the environment that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