COMP-573A Microcomputers

Intel x86 Architecture (I32)

slides by Alexandre Denault

General Description

- Most popular architecture for personal computers.
- Over 25 years old.
- One of the most complicated to work with.

A bit of history …

- 1968: Bob Noyce and Gordon Moore leave Fairchild Semiconductor to found their own company, “N M Electronics”. The company was latter renamed to Intel, which is short for “integrated electronic”.
- 1969: Intel engineer Ted Hoff designs a general-purpose logic chip that can be programmed to take instructions. A multi-chip project can now be handle by one chip. This chip was named the 4004.
- 1972: Intel releases the 8008 processor, which is the first 8-bit microprocessor.
- 1974: Intel debuts the 8080 processor. This processor features a 8-bit data bus, 16-bit addressing and runs at 2 Mhz.
- 1978: Intel introduces its first x86 chip, the 8086 microprocessor.

The IA-32 (x86) Architecture

- Contains both 16-bit and 32-bit processor.
- Programs written in 1978 for the x85 can still run on today’s IA-32 microprocessor.
- Contains very few general purpose registers.
- Very large instruction set.

Memory Architecture

- The 8088 could address 1 mega-byte of memory.
- However, it has no registers larger than 16-bits.

  2^{10} x 2^{10} Bytes = 1024 Bytes x 1024 Bytes = 1MB

  2^{10} x 2^{6} Bytes = 1024 Bytes x 64 Bytes = 64 KB
- How could memory addresses higher than 64 KB be accessed?

Memory Architecture (cont.)

- The 8088 broke up memory into 'segments' called paragraphs.

0000 0000 0000 0000 *0000*	Paragraph 0
0000 0000 0000 0000 *0001*	
0000 0000 0000 0000 *0010*	
…	
0000 0000 0000 0001 *0000*	Paragraph 1
…	
0000 0000 0000 0010 *0000*	Paragraph 2

Memory Architecture (cont.)

- The address of a specific byte in memory is the sum, after an appropriate shift, of two registers.

```
0010 0010 0111 0001      Segment
     0101 1000 1100 0101 Offset
-----------------------
0010 0111 1111 1101 0101 20-bit address
```

- Intel processors with 32 bit registers still store paragraph boundary using 16 bits, but uses a 32-bit offset to the address.

Maximum External Addr. Space

Processor	Date Introduced	Register Sizes (GP registers)	Max Ext. Addr. Space	Memory management
8086	1978	16	1 Mb	64 Kb paragraphs
286	1982	16	16 Mb	Segment registers point to descriptor table
386 DX	1985	32	4 Gb	Added flat memory model
486 DX	1989	32	4 Gb	-
Pentium	1993	32	4 Gb	-
Pentium Pro	1995	32	64 Gb	36 bit address bus
Pentium II	1997	32	64 Gb	-
Pentium III	1999	32	64 Gb	-
Pentium IV	2000	32	64 Gb	-

Segment Registers

- The 8088 has four segment registers allowing access four segments of the memory at the same time. These registers are still present in today’s Intel architectures.
  - **CS**: The Code Segment is used to address program instructions only.
  - **DS**: The Data Segment is used to address data.
  - **SS**: The Stack Segment is used by the PUSH, POP, CALL and RET instruction.
  - **ES**: The Extra Segment is used by resource intensive application to access other blocks of memory.

Offset Registers

- The 8088 has five registers used to store offsets from the segment registers.
  - **IP**: The Instruction Pointer contains the offset for the code segment. Together, they point to the instruction to be executed by the processor.
  - **SP**: The Stack Pointer contains the offset for the stack segment. Together, they point to the current stack position in memory.

Offset Registers (cont.)

- **BP**: The Base Pointer also contains an offset for the stack segment. Together, they point to another stack position in memory. (explained latter)
- **SI/DI** : The source index and the destination index usually offset for the data segment. However, in special situation, the destination index offsets the extra segment.

- These registers are still used today, but have been expanded to 32 bits. These registers have been respectively rename EIP, ESP, EBP, ESI and EDI.

Data Registers

- Four other 32-bit registers are each addressable as 32-bit registers, 16-bit registers (the lower 16 bits), or two 8-bit registers:

```
 7      0 7      0
 +-----------------+--------+--------+
 +             EAX | AX  AH |     AL | Accumulator
 +-----------------+--------+--------+
 +             EBX | BX  BH |     BL | Base register
 +-----------------+--------+--------+
 +             ECX | CX  CH |     CL | Counter
 +-----------------+--------+--------+
 +             EDX | DX  DH |     DL | Bit numbering
 +-----------------+--------+--------+
32                   15              0
```

Data Registers (cont.)

- The Accumulator is used with a few arithmetic instructions, such as MUL and DIV; it is also used for I/O and many instructions perform more efficiently if they use EAX, AX or AL rather than any other register.
- The Base register is the only one of these four that can be used to index into memory; EBX normally points to the Data Segment.
- The Counter is normally used to control the execution of loops.
- The Data register is used by a few instructions to extend the Accumulator to 64 bits.

Status Flag Register

- Set of status bits used to describe different “special” state.

Bit	Name	Function
0	CF	Carry Flag Set on high-order bit carry or borrow; cleared otherwise.
2	PF	Parity Flag -- Set if low-order eight bits of result contain an even number of 1 bits; cleared otherwise.
4	AF	Adjust flag -- Set on carry from or borrow to the low order four bits of AL; cleared otherwise. Used for decimal arithmetic.
6	ZF	Zero Flag -- Set if result is zero; cleared otherwise.
7	SF	Sign Flag -- Set equal to high-order bit of result (0 is positive, 1 if negative).
11	OF	Overflow Flag -- Set if result is too large a positive number or too small a negative number (excluding sign-bit) to fit in destination operand; cleared otherwise.

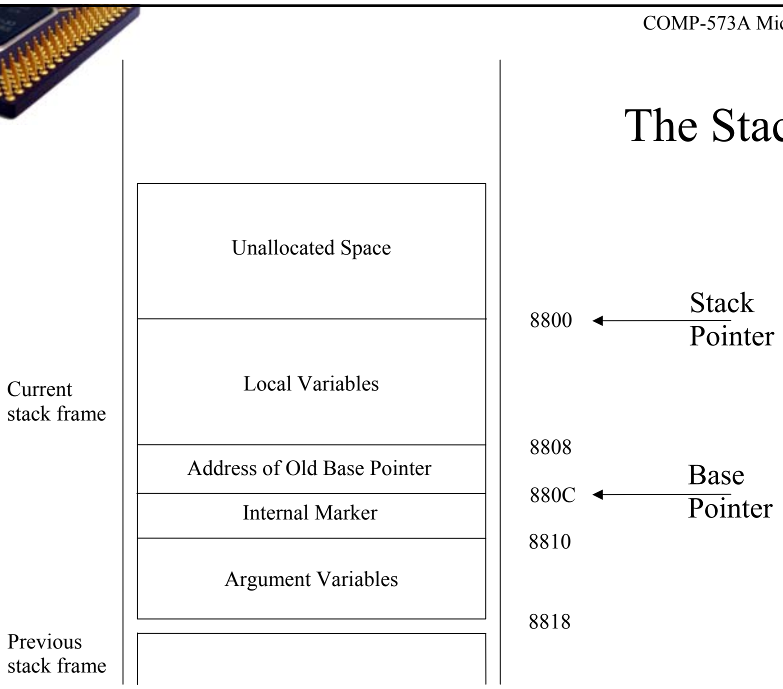
Status Flag Register (cont.)

- Complete schematic of flag register:

```
   31              23          17  15  13       9 8 7 6 5 4 3 2 1 0
  ++---------------+---------------+----------------+--------------+
  |                           |V|R| |N|I/O |O|D|I|T|S|Z| |A| |P| |C|
  |0 0 0 0 0 0 0 0 0 0 0 0 0 0| | |0| |    | | | | | | |0| |0| |1| |
  |                           |M|F| |T| PL |F|F|F|F|F|F| |F| |F| |F|
  +----------------------------------------------------------------+

                               | |   |  |   | | | | | |   |   |   |
  VIRTUAL 8086 MODE---X--------+ |   |  |   | | | | | |   |   |   |
        RESUME FLAG---X----------+   |  |   | | | | | |   |   |   |
   NESTED TASK FLAG---X--------------+  |   | | | | | |   |   |   |
I/O PRIVILEGE LEVEL---X-----------------+   | | | | | |   |   |   |
           OVERFLOW---S---------------------+ | | | | |   |   |   |
     DIRECTION FLAG---C-----------------------+ | | | |   |   |   |
   INTERRUPT ENABLE---X-------------------------+ | | |   |   |   |
          TRAP FLAG---S---------------------------+ | |   |   |   |
          SIGN FLAG---S-----------------------------+ |   |   |   |
          ZERO FLAG---S-------------------------------+   |   |   |
    AUXILIARY CARRY---S-----------------------------------+   |   |
        PARITY FLAG---S---------------------------------------+   |
         CARRY FLAG---S-------------------------------------------+
```

The Stack



Addressing Memory

- Direct Addressing

  Using a variable name to reference memory is called ‘direct’ addressing.

  ```
  MOVL %EDX,my_var
  ```

- Indirect Addressing

  Using an offset stored in a register to reference memory is called ‘indirect’ addressing.

  ```
  MOVL %EDX,(%EBX)

  MOVL %EAX,-8(%EBP)
  ```

  Only four registers can be used for indirect addressing: **EBX, EBP**, ESI, EDI

Instruction Set

- Most intructions come in three flavors:
  - Byte Instructions (b)
  - Word Instructions (w)
  - Long Instructions (l)
- These three categories denote the size of the operands given to the function.

  For example, the add<u>w</u> instruction will add two word (16 bits) values.

- Current CPU support other data types (such as FP or MMX). However, these will not be covered in this course.

Move Instructions

- Move instructions are available in three formats : movb, movw and movl
- They takes two argument, a source and a destination (in order)
- Operands must be of specific type:

Allowed	Not Allowed
• mov**x** register, register • mov**x** constant, register • mov**x** address, register • mov**x** register, address	• mov**x** register, constant • mov**x** address, address

Arithmetic Instructions

- Programmers are responsible for keeping track of which values are signed and unsigned.
- For addition and subtraction, no special operation are needed (since two’s complement is used).
- However, both signed and unsigned instructions are provided for some operations.

	Unsigned	**Signed**
byte	0 to 255	+127 to –128
word	0 to 65535	+32767 to –32768
long	0 to 4294967295	+2147483648 to –2147483648

Arithmetic Instructions (cont.)

- Simple operations such as add**x** and sub**x** can be executed over any registers or memory (some restriction applies).

```
ADDW %CX,%AX -> AX = AX + CX
```

- Increment (inc**x**) and decrement (dec**x**) operation are provided to increase or decrease a value by 1 without the need of a constant.

```
INCL %EBX
```

Arithmetic Instructions (cont.)

- Multiplication and division instructions require only one parameter. The other parameter is assume to be in EAX.
- Results of a multiplication are store in EAX (higher order bits are stored in EDX for long multiplications).
- Results of multiplication are store in EAX. The remainder is store in AH for byte division and in EDX for larger divisions.
- Even if the EDX is not used, it is usually a good idea to clear it before a multiplication or a division.

Arithmetic Instructions (cont.)

- Examples of multiplication and division operations:
  - Unsigned

```
MULB %CL            -> AX = AL * CL
MULW %CX            -> DXAX = AX * CX
MULL %ECX           -> EDXEAX = EAX * ECX
MULW my_byte        -> AX = AX * my_byte
DIVB %CL            -> AL = AL / CL
                    AH contains the remainder
```

  - Signed

```
IMULW %DI           -> DXAX = AX * DI
IMULL %EDI          -> EDXEAX = EAX * EDI
IMULW my_word       -> DXAX = AX * my_word
IDIVW my_word       -> AX = DXAX / myword
                    DX contains the remainder
```

Logical Operators Instructions

- Logical operators do not need to be called for a specific format (byte, word, etc). The type of instruction is determined by the size of the arguments.
- Some logical operators:
  - AND : Logical AND
  - OR : Logical inclusive OR
  - XOR : Logical eXclusive OR
  - NOT : Logical negation: form 1's complement (not the same as NEG : 2's complement)
  - TEST : Logical AND without affecting destination (not the same as CMP : substraction)

Jump Instructions

- The JMP instruction can be used to branch the execution of a program to a target label.
- All Jump instruction exists in several main format :
  - Short jump—A near jump where the jump range is limited to –128 to +127 from the current EIP value.
  - Near jump—A jump to an instruction within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment jump.
  - Far jump—A jump to an instruction located in a different segment than the current code segment but at the same privilege level, sometimes referred to as an intersegment jump.
  - Task switch—A jump to an instruction located in a different task.

Jump Instructions (cont.)

- The compiler (GCC) will automatically determine which jump instruction format should be used.
- Conditional jump instructions will branch conditionally on a value in the status register. Different types of conditional jumps have different criteria for branching.
- Some conditional jump instructions are specifically designed for signed or unsigned numbers.
- The Intel reference guide contains a complete list of available jump instruction.
- The compare (cmp) instruction should be use to set the status flag used by conditional jump instructions.

Stack Instructions

- The Push and Pop instructions can be used to add and remove data on the stack.
- Like the move instruction, these instructions are available in three formats:

  pushb, pushw, pushl, popb, popw, popl
- These operations will change the value of the stack pointer (sp).

Functions Instructions

- The **call** instruction is use to branch the execution of the program to a new function.
- Since call is also a branch instruction, there are several formats for this instruction.
- The **ret** instruction is used to return the execution of rhe program to the previous function.

Instruction Reference

- Here are some of the most common Intel assembly instructions.
- The descriptions were taken from “the IA-32 Intel Architecture Software Developers Manual Volume 2 Instruction Set Reference”
- The instruction format used in the Intel IA-32 Manual and the instruction format used by Linux is different. Some arguments (such as destination and source) may be switched.
- The instructions in the following sections are all in the “Linux” format.

Move

Syntax:

MOVx SRC, DEST

Operation:

DEST ← SRC

Description:

Copies the first operand (source operand) **to the second operand** (destination operand). The source operand can be an immediate value, general-purpose register, segment register, or memory location; the destination register can be a general-purpose register, segment register, or memory location. Both operands must be the same size, which can be a byte, a word, or a doubleword.

The MOV instruction cannot be used to load the CS register. Attempting to do so results in an invalid opcode exception (#UD). To load the CS register, use the far JMP, CALL, or RET instruction.

Addition

Syntax:

ADDx SRC, DEST

Operation:

DEST ← DEST + SRC

Description:

Adds the first operand (source operand) **and the second operand** (destination operand) and **stores the result in the destination operand**. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADD instruction performs integer addition. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate a carry (overflow) in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

Increment

Syntax:

INCx DEST

Operation:

DEST ← DEST +1

Description:

Adds 1 to the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (Use a ADD instruction with an immediate operand of 1 to perform an increment operation that does updates the CF flag.)

Unsigned Multiplication

Syntax:

MULx SRC

Operation:

Byte: AX ← AL * SRC

Word: DX:AX ← AX ∗ SRC

Long: EDX:EAX ← EAX ∗ SRC

Description:

Performs an unsigned multiplication of the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand is an implied operand located in register AL, AX or EAX (depending on the size of the operand); the source operand is located in a general-purpose register or a memory location.

Signed Multiplication

Syntax: (first form, multiple forms of this operation exists)

IMOVx SRC

Operation:

Byte: AX ← AL * SRC

Word: DX:AX ← AX ∗ SRC

Long: EDX:EAX ← EAX ∗ SRC

Description:

Performs a signed multiplication of two operands. This instruction has three forms, depending on the number of operands.

- One-operand form: This form is identical to that used by the MUL instruction. Here, the source operand (in a general-purpose register or memory location) is multiplied by the value in the AL, AX, or EAX register (depending on the operand size) and the product is stored in the AX, DX:AX, or EDX:EAX registers, respectively.

Unsigned Division

Syntax:

DIVx SRC

Operation:

Byte:	Quotient: AL Remainder: AH ← AX / SRC
Word:	Quotient: AX Remainder: DX ← DX:AX / SRC
Long:	Quotient: EAX Remainder: EDX ← EDX:EAX / SRC

Description:

Divides (unsigned) the value in the AX, DX:AX, or EDX:EAX registers (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, or EDX:EAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor),

Logical AND

Syntax:

ANDx SRC, DEST

Operation:

DEST ← DEST AND SRC

Description:

Performs a bitwise AND operation on the destination (second) and source (first) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is set to 1 if both corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

Logical OR

Syntax:

ORx SRC, DEST

Operation:

DEST ← DEST OR SRC

Description:

Performs a bitwise inclusive OR operation between the destination (second) and source (first) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result of the OR instruction is set to 0 if both corresponding bits of the first and second operands are 0; otherwise, each bit is set to 1.

Two's Complement Negation

Syntax:

NEGx DEST

Operation:

DEST ← - DEST

Description:

Replaces the value of operand (the destination operand) **with its two's complement.** (This operation is equivalent to subtracting the operand from 0.) The destination operand is located in a general-purpose register or a memory location.

One’s Complement Negation

Syntax:

NOTx DEST

Operation:

DEST ← NOT DEST

Description:

Performs a bitwise NOT operation (each 1 is cleared to 0, and each 0 is set to 1) on the destination operand and stores the result in the destination operand location. The destination operand can be a register or a memory location.

Compare Two Operands

Syntax:

CMPx SRC1, SRC2

Operation:

TEMP ← SRC1 − SignExtend(SRC2)

MODIFY STATUS FLAGS

Description:

Compares the first source operand with the second source operand and **sets the status flags in the EFLAGS register** according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand.

The CMP instruction is typically used in conjunction with a conditional jump (J*cc*), condition move (CMOV*cc*), or SET*cc* instruction. The condition codes used by the J*cc*, CMOV*cc*, and SET*cc* instructions are based on the results of a CMP instruction.

Logical Compare

Syntax:

TESTx SRC1, SRC2

Operation:

TEMP ← SRC1 AND SRC2

MODIFY STATUS FLAGS

Description:

Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.

Jump

Syntax:

JMP DEST

Operation:

Short Jump: EIP ← EIP + DEST

Near Jump: EIP ← DEST

Description:

Transfers program control to a different point in the instruction stream without recording return information. The destination (target) operand specifies the address of the instruction being jumped to. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of jumps.

Conditional Jump

Syntax:

```
Jcc DEST
```

Operation:

```
IF condition
THEN EIP ← DEST
```

Description:

Checks the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and, if the flags are in the specified state (condition), **performs a jump to the target instruction** specified by the destination operand. A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, the jump is not performed and execution continues with the instruction following the J*cc* instruction.

Call Procedure

Syntax:

```
CALL DEST
```

Operation:

```
PUSH EIP
EIP ← DEST
```

Description:

Saves procedure linking information on the stack and **branches to the procedure** (called procedure) specified with the destination (target) operand. The target operand specifies the address of the first instruction in the called procedure. This operand can be an immediate value, a general purpose register, or a memory location.

Return from Procedure

Syntax:

RET OPTARG

Operation:

POP EIP

POP to nothing OPTARG times

Description:

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed.

Push on the Stack

Syntax:

PUSHx SRC

Operation:

ESP ← ESP – x

SS:ESP ← SRC

Description:

Decrements the stack pointer and then **stores the source operand on the top of the stack**. The address-size attribute of the stack segment determines the stack pointer size (16 bits or 32 bits), and the operand-size attribute of the current code segment determines the amount the stack pointer is decremented (2 bytes or 4 bytes). For example, if these address- and operand-size attributes are 32, the 32-bit ESP register (stack pointer) is decremented by 4 and, if they are 16, the 16-bit SP register is decremented by 2.

Pop from Stack

Syntax:

POPx DEST

Operation:

DEST ← SS:ESP

ESP ← ESP + x

Description:

Loads the value from the top of the stack to the location specified with the destination operand and then **increments the stack pointer**. The destination operand can be a general-purpose register, memory location, or segment register.

No Operation

Syntax:

NOP

Operation:

None

Description:

Performs no operation. This instruction is a one-byte instruction that takes up space in the instruction stream but does not affect the machine context, except the EIP register.

The NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

Struct. of an Assembly Prog.

```
        .section        .rodata
.LC0:
        .string "Hello World\n"
```

Constants Declaration

```
        .align 4
         .text
```

Code Alignment

```
.globl main
        .type   main,@function
```

Function Declaration

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        subl    $12, %esp
        pushl   $.LC0
        call    printf
        leave
        ret
```

Function Code

Constants Declaration

- Constants should be declare at the beginning of the file (though gcc sometimes puts them at the end).
- Most common type of constants are
  - string
  - byte
  - word
  - long
- Can create array of constants (see gas manual).

Code Alignment

- Some architecture require specific aligning of instructions in memory (x86 requires an alignment of a least 3).
- Code takes a little more space in memory (because of the padding).
- Makes code easier to debug by aligning instructions in memory.

Hello Word

```
.section        .rodata              /* Constant Declaration */
.LC0:   .string "Hello World\n"       /* Hello World string   */

        .text

.globl main
        .type   main,@function        /* Make main global so the shell can
                                         execute it */
main:
        pushl   %ebp                  /* Save the old base pointer */
        movl    %esp, %ebp            /* Set the base pointer to the current
                                          position of the stack */
        pushl   $.LC0                 /* Pushes the string on the stack */
        call    printf                /* Calls the printf function */
        addl    $4, %esp              /* Removes the printf arguments from
                                          the stack */
        movl    %ebp, %esp            /* Restores the old stack pointer */
        popl    %ebp                  /* Restores the old base pointer */

        ret                           /* Return from this function */
```

Allocating space on stack

```
...
 pushl   %ebp                    /* Save the old base pointer */
 movl    %esp, %ebp              /* Set the base pointer to the current
                                     position of the stack */

 subl    $12, %esp               /* Allocate  3*4 bytes on stack for
                                     x,y,z */

 movl    $3, -4(%ebp)            /* x = 3 */
 movl    $4, -8(%ebp)            /* y = 4 */
 movl    $5, -12(%ebp)           /* x = 5 */
...

 addl    $12, %esp               /* Restore space on the stack */

 movl    %ebp, %esp              /* Restores the old stack pointer */
 popl    %ebp                    /* Restores the old base pointer */

...
```

Calling a C function

```
        ...
.LC0:    .string "Print test x=%i, y=%i, z=%i\n"
        ...

         movl    $3, -4(%ebp)            /* x = 3 */
         movl    $4, -8(%ebp)            /* y = 4 */
         movl    $5, -12(%ebp)           /* z = 5 */

         pushl   -12(%ebp)               /* Push arguments on stack */
         pushl   -8(%ebp)                /* Notice that arguments are */
         pushl   -4(%ebp)                /* pushed in inverse order */
         pushl   $.LC0
         call    printf                  /* Calls printf */
         addl    $16, %esp               /* Erase arguments from stack */

        ...
```

Retrieving Arguments

```
 .type   add,@function
add:
        pushl   %ebp                    /* Save the old base pointer */
        movl    %esp, %ebp              /* Set the base pointer to the current
                                            position of the stack */
        subl    $12, %esp               /* Allocate space for a,b,c */

                                        /* Since we can't move from memory
                                            to memory, we use eax as our
                                            temp register */
        movl    8(%ebp), %eax           /* temp = x */
        movl    %eax, -4(%ebp)          /* a = temp */
        movl    12(%ebp), %eax          /* temp = y */
        movl    %eax, -8(%ebp)          /* b = temp */

        movl    -8(%ebp), %eax          /* temp = a */
        addl    -4(%ebp), %eax          /* temp = b + temp */
        movl    %eax, -12(%ebp)         /* c = temp */

        ...
```

If statement

C Code:

```
if( i == 0 ) {
/* Inside if */
}
/* Outside if */
```

Assembler:

```
    cmpl    $0, -4(%ebp)
    jne     .notif
    /* Inside if */
.notif:
    /* Outside if */
```

For statement

C Code:

```
for (j = 0; j++; j < 15) {
/* Inside for */
}
/* Outside for */
```

Assembler:

```
 movl    $0, -8(%ebp)
.start_for:
        cmpl    $14, -8(%ebp)
        jle     .in_for
        jmp     .exit_for
.in_for:
        /* Inside for */
        incl    (%eax)
        jmp .start_for
.exit_for:
        /* Outside for */
```

Evolution of the x86

- **Features introduce in the 286 :**
  - Protected mode operations (descriptor tables)
  - Virtual memory management
  - New protection mecanism (segment limit checking, etc)
- **Features introduce in the 386 :**
  - First 32-bit processor of the x86 family
  - New virtual mode for executing 8086 and 8088 applicaiotn
  - Flat memory model
  - Paging (fixed 4-Kb page size)
  - Limited parallele stages (6 stages)

Evolution of the x86 (cont.)

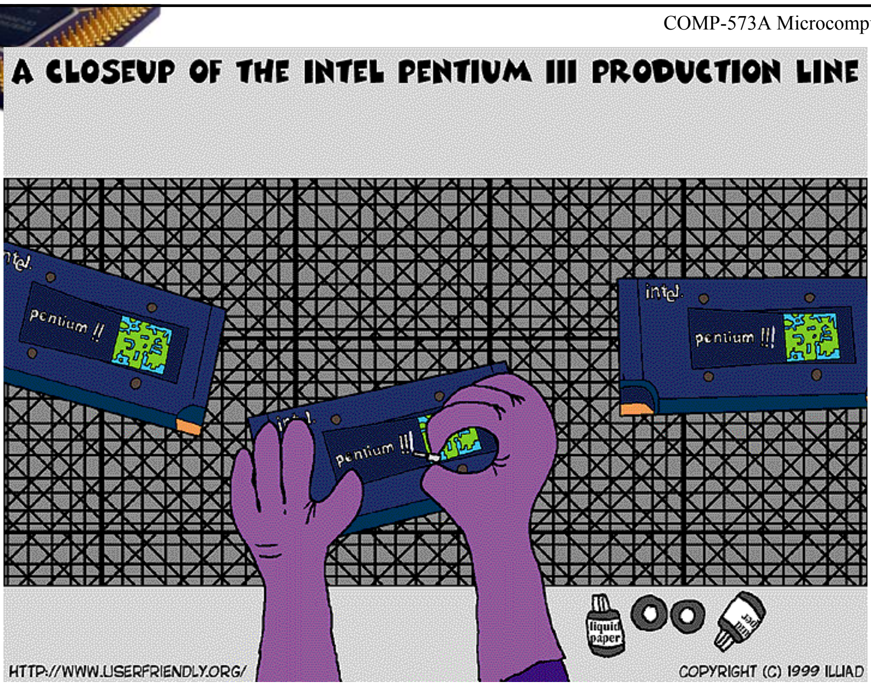
- **Features introduce in the 486 :**
  - Improved parallele execution
  - Integrated FPU (Floating Point Unit)
  - On-chip level 1 cache
  - Power management (late in the series)
- **Features introduce in the Pentium :**
  - Addition of another execution pipeline (making the architecture superscaler)
  - Branch prediction
  - Built-in multi-processor support
  - MMX (Multimedia Extensions)

Evolution of the x86 (cont.)

- **Features introduce in the Pentium Pro:**
  - Addition of another execution pipeline (bringing the total up to 3 pipelines)
  - Dynamic Execution (out-of-order execution, superior branch prediction, flow analysis, etc)
  - On-chip level 2 cache
  - No MMX
- **Features introduce in the Pentium II:**
  - Similar to Pentium Pro, but with MMX
  - Cartridge design
  - Improved power management (sleep, etc)

Evolution of the x86 (cont.)

- **Features introduce in the Pentium III:**
  - Streaming SIMD Extensions (SSE)
- **Features introduce in the Pentium VI:**
  - First Implementation of NetBurst micro-architecture
    - Rapid Execution Engine
    - Hyper Pipeline Technology
  - Streaming SIMD Extensions 2 (SSE2)


A CLOSEUP OF THE INTEL PENTIUM III PRODUCTION LINE
intel
pentium II
intel
pentium III
intel
pentium III
liquid paper
HTTP://WWW.USERFRIENDLY.ORG/
COPYRIGHT (C) 1999 ILLIAD

Pentium 4 vs Athlon XP+

- New net-burst architecture
- 2 double-speed ALU and 1 FPU
- Streaming SIMD Extensions 2 (SSE)
- 20 levels of pipelines
- L1 cache replaced by 8kb dual cache

- Tradditional x86 architecture
- 3 ALU and 3 FPU
- Streaming SIMD Extensions (SSE)
- 11 levels of pipelines
- 64 kb L1 cache

References

- The IA-32 Intel Architecture Software Developers Manual Volume 1,2 and 3
- The Floppy Textbook (1999 edition)
- Intel Assembler 80x86 CodeTable http://www.jegerlehner.ch/intel/
- UserFriendly Comics http://ars.userfriendly.org/cartoons/?id=19990328