The Intel Microprocessor Family

- The Intel family owes its origins to the 8080, an 8-bit processor which could only access 64 kilobytes of memory.
- The 8086 (1978) had 16-bit registers, a 16-bit data bus, 20-bit memory using segmented memory. The IBM PC used the 8088, which was identical except it used an 8-bit data bus.
- 8087 - a math co-processor that worked together with the 8086/8088. Without it, floating point arithmetic require complex software routines.
- 80286 - ran in real mode (like the 8086/8088) or in protected mode could access up tp 16MB using 24-bit addressing with a clock speed between 12 and 25 MHz. Its math co-processor was the 80287.
The Intel Microprocessor Family (continued)

- 80386 or **i386** (1985) - used 32-bit registers and a 32-bit data bus. It could operate in real, protected or virtual mode. In virtual mode, multiple real-mode programs could be run.
- **i486** - The instruction set was implemented with up to 5 instructions fetched and decoded at once. SX version had its FPU disabled.
- The Pentium processor had an original clock speed of 90 MHz and cold decode and executed two instructions at the same time, using **dual pipelining**.

IA-32 Processor Modes of Operations

- There are three basic modes of operation on IA-32 processors:
  - **Protected Mode** – The native processor state, where all instructions and features are available. Each process is given its own memory segment and the processor catches any process attempting to go outside its own segment
  - **Real-address Mode** – The processor acts as if it were an Intel 8086 processor with its more limited environment
  - **System Management Mode** – provides a mechanism for implementation power management and system security
IA-32 Processor Address Space

- In protected mode IA-32 processors can access up to 4 Gigabytes of storage, with memory addresses from 0 to $2^{32} - 1$.
- In real mode, a maximum of 1 megabyte of memory can be accessed with memory addresses from 0 to $2^{20} - 1$.
- The IA-32 processors provide a Virtual 8086 where multiple MS-DOS programs can run safely within an Windows environment.

P6 Processor Family

- The P6 family of processors was introduced in 1995.
- It includes the Pentium Pro, Pentium II, Pentium III and Pentium 4.
- The Pentium II introduces MMX technology for multimedia applications.
- The Pentium III introduced SIMD with 128-bit registers to move larger amounts of data.
- The Pentium 4 uses NetBurst micro-architecture to allow the processors to operate at higher speeds.
CISC Architecture

- The Intel processors have been based on the CISC (Complex Instruction Set Computer) approach to processor design.
- CISC processors have large, powerful instruction sets that can include many high-level operations. But the size of the instruction set makes the control unit relatively slow.

RISC Architecture

- RISC computers use smaller, streamlined instruction sets that allow their control units to be quicker.
- Intel processors are backwards-compatible and are basically CISC but use RISC features such as pipelining and superscalar.
32-bit Register

General Purpose

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>AX (Accumulator) - favored for arithmetic operations</td>
</tr>
<tr>
<td>EBX</td>
<td>BX (Base) - Holds base address for procedures and variables</td>
</tr>
<tr>
<td>ECX</td>
<td>CX (Counter) - Used as a counter for looping operations</td>
</tr>
<tr>
<td>EDX</td>
<td>DX (Data) - Used in multiplication and division operations</td>
</tr>
</tbody>
</table>

Index

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBP</td>
<td></td>
</tr>
<tr>
<td>ESP</td>
<td></td>
</tr>
<tr>
<td>ESI</td>
<td></td>
</tr>
<tr>
<td>EDI</td>
<td></td>
</tr>
</tbody>
</table>

Status and Control

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFLAGS</td>
<td>Flags</td>
</tr>
<tr>
<td>EIP</td>
<td>IP</td>
</tr>
</tbody>
</table>

Segment

<table>
<thead>
<tr>
<th>Segment</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>SS</td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td></td>
</tr>
<tr>
<td>ES</td>
<td></td>
</tr>
<tr>
<td>FS</td>
<td></td>
</tr>
<tr>
<td>GS</td>
<td></td>
</tr>
</tbody>
</table>

16-bit Processor Architecture

General Purpose Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td>AH, AL (Accumulator) - favored for arithmetic operations</td>
</tr>
<tr>
<td>BX</td>
<td>BH, BL (Base) - Holds base address for procedures and variables</td>
</tr>
<tr>
<td>CX</td>
<td>CH, CL (Counter) - Used as a counter for looping operations</td>
</tr>
<tr>
<td>DX</td>
<td>DH, DL (Data) - Used in multiplication and division operations</td>
</tr>
</tbody>
</table>
Segment Registers

*Segment registers* are used to hold base addresses for program code, data and the stack.

- **CS (Code Segment)** - holds the base address for all executable instructions in the program.
- **SS (Stack Segment)** - holds the base address for the stack.
- **DS (Data Segment)** - holds the base address for variables.
- **ES (Extra Segment)** - an additional base address value for variable.

Index Registers

*Index Registers* contain the offsets for data and instructions.

- **Offset** - distance (in bytes) from the base address of the segment.
- **BP (Base Pointer)** - contains an assumed offset from the SS register; used to locate variables passed between procedures.
- **SP (Stack Pointer)** - contains the offset for the top of the stack.
- **SI (Source Index)** - Points to the source string in string move instructions.
- **DI (Destination Index)** - Points to the source destination in string move instructions.
Status and Control Registers

IP (Instruction Pointer) - contains the offset of the next instruction to be executed within the current code segment.

Flags register contain individual bits which indicate CPU status or arithmetic results. They are usually set by specific instructions.

- **O** = Overflow
- **D** = Direction
- **I** = Interrupt
- **T** = Trap
- **x** = undefined
- **S** = Sign
- **Z** = Zero
- **A** = Auxiliary Carry
- **P** = Parity
- **C** = Carry

Flags

There are two types of flags: control flags (which determine how instructions are carried out) and status flags (which report on the results of operations.

Control flags include:

- **Direction** Flag (DF) - affects the direction of block data transfers (like long character string). 1 = up; 0 = down.

- **Interrupt** Flag (IF) - determines whether interrupts can occur (whether hardware devices like the keyboard, disk drives, and system clock can get the CPU’s attention to get their needs attended to.

- **Trap** Flag (TF) - determines whether the CPU is halted after every instruction. Used for debugging purposes.
Status Flags

- **Status Flags include:**
  - **Carry** Flag (CF) - set when the result of *unsigned* arithmetic is too large to fit in the destination. 1 = carry; 0 = no carry.
  - **Overflow** Flag (OF) - set when the result of *signed* arithmetic is too large to fit in the destination. 1 = overflow; 0 = no overflow.
  - **Sign** Flag (SF) - set when an arithmetic or logical operation generates a negative result. 1 = negative; 0 = positive.
  - **Zero** Flag (ZF) - set when an arithmetic or logical operation generates a result of zero. Used primarily in jump and loop operations. 1 = zero; 0 = not zero.
  - **Auxiliary Carry** Flag - set when an operation causes a carry from bit 3 to 4 or borrow (from bit 4 to 3). 1 = carry, 0 = no carry.
  - **Parity** - used to verify memory integrity. Even # of 1s = Even parity; Odd # of 1s = Odd Parity

---

Floating-Point Unit

<table>
<thead>
<tr>
<th>80-bit Data Registers</th>
<th>48-bit Pointer Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(0)</td>
<td>FPU Instruction Pointer</td>
</tr>
<tr>
<td>ST(1)</td>
<td>FPU Data Pointer</td>
</tr>
<tr>
<td>ST(2)</td>
<td></td>
</tr>
<tr>
<td>ST(3)</td>
<td></td>
</tr>
<tr>
<td>ST(4)</td>
<td>16-bit Control Registers</td>
</tr>
<tr>
<td>ST(5)</td>
<td>Tag Register</td>
</tr>
<tr>
<td>ST(6)</td>
<td>Control Register</td>
</tr>
<tr>
<td>ST(7)</td>
<td>Status Register</td>
</tr>
<tr>
<td></td>
<td>Opcode Register</td>
</tr>
</tbody>
</table>
Segmented Memory Map, Real-Address Mode

Relocatable addressing

- This is an example of relocatable addressing, which allows programs on multitasking systems to be moved from one area of memory to another without rearranging every address referenced.
- Addresses can be rearranged simply by changing the value of the appropriate segment register.
Protected Mode Memory Management

- When the processor runs in protected mode, a program can access up to 4 gigabytes of memory.
- Although the programmer’s view of memory is a flat image of 4 GB, the operating system works in the background to create and maintain this image.
- The segment registers point to segment descriptor tables, which define locations of the program segments:
  - CS refers to the code segment’s descriptor table
  - DS refers to the data segment’s descriptor table
  - SS refers to the stack segment’s descriptor table

Flat Segmentation Memory Model

![Flat Segmentation Memory Model Diagram]
Paging

- IA-32 architecture also allows memory segments to be divided into 4K units called pages.
- Many of these pages of memory are saved on disk in a swap file and are loaded into memory (and rewritten in the swap file) when the CPU needs a page that is not present in physical memory. This situation is called a page fault.
- The use of paging and swap files allows the memory used to be several times larger than physical memory; it is known as virtual memory.

Examples of Integer Constants

- 26     Decimal
- 1Ah    Hexadecimal
- 1101b  Binary
- 36q    Octal
- 2Bh    Hexadecimal
- 42Q    Octal
- 36D    Decimal
- 47d    Decimal
Examples of Integer Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$16 / 5$</td>
<td>3</td>
</tr>
<tr>
<td>$-(3 + 4) \times (6 - 1)$</td>
<td>-35</td>
</tr>
<tr>
<td>$-3 + 4 \times 6 - 1$</td>
<td>20</td>
</tr>
<tr>
<td>$4 + 5 \times 2$</td>
<td>1</td>
</tr>
<tr>
<td>$-5 + 2$</td>
<td></td>
</tr>
<tr>
<td>$12 - 1 \text{ MOD } 5$</td>
<td></td>
</tr>
<tr>
<td>$(4 + 2) \times 6$</td>
<td></td>
</tr>
</tbody>
</table>

Real Number Constants

- There are two types of real number constants:
  - *Decimal reals*, which contain a sign followed by a number with decimal fraction and an exponent:
    
    $[\text{sign}] \text{ integer.}[\text{integer}][\text{exponent}]$

    Examples:
    
    2. $+3.0 \quad -44.2\text{E}+05 \quad 26.\text{E}5$

  - *Encoded reals*, which are represented exactly as they are stored:

    $3\text{F80000r}$
Characters Constants

• A character constant is a single character enclosed in single or double quotation marks.

• The assembler converts it to the equivalent value in the binary code ASCII:
  ‘A’
  “d”

String Constants

• A string constant is a string of characters enclosed in single or double quotation marks:
  ‘ABC’
  “x”
  “Goodnight, Gracie”
  ‘4096’
  “This isn’t a test”
  ‘Say “Goodnight, ” Gracie’
Reserved Words

- Reserved words have a special meaning to the assembler and cannot be used for anything other than their specified purpose.
- They include:
  - Instruction mnemonics
  - Directives
  - Operators in constant expressions
  - Predefined symbols such as `@data` which return constant values at assembly time.

Examples of Identifiers

<table>
<thead>
<tr>
<th>var1</th>
<th>open_file</th>
</tr>
</thead>
<tbody>
<tr>
<td>_main</td>
<td>_12345</td>
</tr>
<tr>
<td>@@myfile</td>
<td>$first</td>
</tr>
<tr>
<td>Count</td>
<td>MAX</td>
</tr>
<tr>
<td>xVal</td>
<td></td>
</tr>
</tbody>
</table>
Directives

• Directives are commands for the assembler, telling it how to assemble the program.
• Directives have a syntax similar to assembly language but do not correspond to Intel processor instructions.
• Directives are also case-insensitive:
• Examples
  .data
  .code
  name PROC

Instructions

• An instruction in Assembly language consists of a name (or label), an instruction mnemonic, operands and a comment
• The general form is:
  [name] [mnemonic] [operands] [; comment]
• Statements are free-form; i.e., they can be written in any column with any number of spaces between in each operand as long as they are on one line and do not pass column 128.
Labels

- Labels are identifiers that serve as place markers within the program for either code or data.
- These are replaces in the machine-language version of the program with numeric addresses.
- We use them because they are more readable:
  \[
  \text{mov ax, [9020]} \]
  vs.
  \[
  \text{mov ax, MyVariable} \]

Code Labels

- Code labels mark a particular point within the program’s code.
- Code labels appear at the beginning and are immediately followed by a colon:
  \[
  \text{target:} \\
  \text{mov ax, bx} \\
  \text{... ...} \\
  \text{jmp target} \]
Data Labels

- Labels that appear in the operand field of an instruction:
  \[ \text{mov} \quad \text{first, ax} \]
- Data labels must first be declared in the data section of the program:
  \[ \text{first} \ \text{BYTE} \ 10 \]

Instruction Mnemonics

- Instruction mnemonics are abbreviations that identify the operation carried out by the instruction:
  \[ \begin{align*}
  \text{mov} & \quad - \text{move a value to another location} \\
  \text{add} & \quad - \text{add two values} \\
  \text{sub} & \quad - \text{subtract a value from another} \\
  \text{jmp} & \quad - \text{jump to a new location in the program} \\
  \text{mul} & \quad - \text{multiply two values} \\
  \text{call} & \quad - \text{call a procedure}
  \end{align*} \]
Operands

• Operands in an assembly language instruction can be:
  – constants
  – constant expressions
  – registers
  – memory locations

Operands and Instructions

• All instructions have a predetermined number of operands.
• Some instructions use no operands:
  
  stc ; set the Carry Flag

• Some instructions use one operand:
  
  inc ax ; add 1 to AX

• Some instructions use two operands:
  
  mov count, bx ; add BX to count
Example: Adding Three Numbers

TITLE Add And Subtract  (AddSub.asm)  marks the program’s title
; This program adds and subtracts 32-bit ; integers.
INCLUDE Irvine32.inc Copies the file’s contents into the program
.code
main  PROC
      mov eax, 10000h ;Copies 10000h into EAX
      add eax, 40000h ;Adds 40000h to EAX
      sub eax, 20000h ;Subtracts 20000h from EAX
      call DumpRegs ; Call the procedure DumpRegs
      exit ; Call Windows procedure Exit ; to halt the program
main  ENDP ; marks the end of main
end main ; last line to be assembled

Program output

EAX=00030000  EBX=00530000  ECX=0063FF68  EDX=BFFC94C0
ESI=817715DC  EDI=00000000  EBP=0063FF78  ESP=0063FE3C
EIP=00401024  EFL=00000206  CF=0  SF=0  ZF=0  OF=0
Assembling, Linking and Running Programs

Intrinsic Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>8-bit unsigned integer</td>
</tr>
<tr>
<td>SBYTE</td>
<td>8-bit signed integer</td>
</tr>
<tr>
<td>WORD</td>
<td>16-bit unsigned integer; also Near Pointer in Real Mode</td>
</tr>
<tr>
<td>SWORD</td>
<td>16-bit signed integer</td>
</tr>
<tr>
<td>DWORD</td>
<td>32-bit unsigned integer; also Near pointer in Protected Mode</td>
</tr>
<tr>
<td>SDWORD</td>
<td>32-bit signed integer</td>
</tr>
</tbody>
</table>
Defining 32-bit Data

• The **DWORD** and **SDWORD** directives allocate storage of one or more 32-bit integers:

  ```
  val1    DWORD    12345678h ; unsigned
  val2    SDWORD   -21474836648; signed
  val3    DWORD    20 DUP(?)     ; unsigned array
  ```

Arrays of Doublewords

• You can create an array of word values by listing them or using the **DUP** operator:

  ```
  myList   DWORD 1, 2, 3, 4, 5
  ```

  Offset 0000 0004 0008 000C 0010
  Value:  1  2  3  4  5
Defining 64-bit and 80-bit Data

• The **QWORD** directive allocate storage of one or more 64-bit (8-byte) values:
  ```
  quad1  QWORD  1234567812345678h
  ```

• The **TBYTE** directive allocate storage of one or more 80-bit integers, used mainly for binary-coded decimal numbers:
  ```
  val1  TBYTE  1000000000123456789h
  ```

Defining Real Number Data

• There are three different ways to define real values:
  – REAL4 defines a 4-byte single-precision real value.
  – REAL8 defines a 8-byte double-precision real value.
  – REAL10 defines a 10-byte extended double-precision real value.

• Each requires one or more real constant initializers.
Examples of Real Data Definitions

rVal1 REAL4 -2.1
rVal2 REAL8 3.2E-260
rVal3 REAL10 4.6E+4096
ShortArray REAL4 20 DUP (?)

rVal1 DD -1.2
rVal2 dq 3.2E-260
rVal3 dt 4.6E+4096

Ranges For Real Numbers

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Significant Digits</th>
<th>Approximate Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Real</td>
<td>6</td>
<td>1.18×10^{-38} to 3.40×10^{38}</td>
</tr>
<tr>
<td>Long Real</td>
<td>15</td>
<td>2.23×10^{-308} to 1.79×10^{308}</td>
</tr>
<tr>
<td>Extended Real</td>
<td>19</td>
<td>3.37×10^{-4932} to 1.18×10^{4932}</td>
</tr>
</tbody>
</table>
Little Endian Order

- Consider the number 12345678h:

<table>
<thead>
<tr>
<th>Little-endian</th>
<th>0000: 78</th>
<th>0001: 56</th>
<th>0002: 34</th>
<th>0003: 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big-endian</td>
<td>0000: 12</td>
<td>0001: 34</td>
<td>0002: 56</td>
<td>0003: 78</td>
</tr>
</tbody>
</table>

Adding Variables to AddSub

TITLE Add And Subtract (AddSub2.asm)
; This program adds and subtracts 32-bit integers.
; and stores the sum in a variable
INCLUDE Irvine32.inc
.data
val1   DWORD10000h
val2   DWORD40000h
val3   DWORD20000h
finalVal DWORD?
.code
main PROC

    mov    eax, val1 ; Start with 10000h
    add    eax, val2 ; Add 40000h
    sub    eax, val3 ; Subtract 2000h
    mov    finalVal, eax ; Save it
    call   DumpRegs ; Display the
             ; registers
    exit
main ENDP
end main

mov Instruction

• The mov instruction copies data from one location to another.
• The following formats are legal for moving data to or from general purpose registers:
  - mov reg, reg
  - mov mem, reg
  - mov reg, mem
• The following formats are legal for immediate operands
  - mov mem, immed
  - mov reg, immed
• The following format are legal for segment registers:
  - mov segreg, r/m16 ; not CS
  - mov r/m16, segreg
Moving Data From Memory to Memory

- Memory to memory moves cannot be done in a single instruction; it requires two instructions:

  .data
  var1 WORD ?
  var2 WORD ?
  ...
  ...
  .code
  mov ax, var1
  mov var1, ax

mov Instruction Examples

Examples of mov instructions

  .data
  Count BYTE 10
  Total WORD 4126h
  Bigval DWORD 12345678h
  ...
  ...
  .code
  mov al, bl ; 8-bit register to register
  mov bl, count ; 8-bit memory to register
  mov count, 26 ; 8-bit immediate to memory
  mov bl, 1 ; 8-bit immediate to register
  mov dx, cx ; 16-bit register to register
  mov bx, 8FE2h ; 16-bit immediate to register
  mov eax, ebx ; 32-bit register to register
  mov edx, bigVal ; 32-bit memory to register
Arithmetic Instructions

Assembly language include many instructions to perform basic arithmetic. They include:

• inc
• dec
• add
• sub

inc and dec Instructions

• The *inc* and *dec* instructions have the format:

  ```
  inc reg/mem ; add 1 to destination’s contents
  dec reg/mem ; subtract 1 to destination’s contents
  ```

• The operand can be either a register or memory operand.
• All status flags (except Carry) are affected.
inc and dec - Examples

- Simple examples
  \begin{verbatim}
  inc  al ; increment 8-bit register
  dec  bx ; decrement 16-bit register
  inc  eax ; increment 32-bit register
  inc  val1 ; increment memory operand
  \end{verbatim}

- Another example
  \begin{verbatim}
  .data
  myWord  WORD  1000h
  .code
  inc  myWord  ; 1001h
  mov  bx, myWord
  dec  bx      ; 1000h
  \end{verbatim}

add Instruction

- add adds a source operand to the destination operand of the same size.
- Format:
  \begin{verbatim}
  add  destination, source
  \end{verbatim}
- Source is unchanged; destination stores the sum. All the status flags are affected.
- The sizes must match and only one can be a memory location.
**add** Instruction - Examples

- Simple examples
  - `add cl, al` ; add 8-bit register to register
  - `add eax, edx` ; add 32-bit register-to-register
  - `add bx, 1000h` ; add immediate value to 16-bit reg
  - `add var1, ax` ; add 16-bit register to memory
  - `add var1, 10` ; add immediate value to memory

- Numeric example
  ```
  .data
  var1 DWORD 10000h
  var2 DWORD 20000h
  .code
  mov eax, var1
  add eax, var2 ; 30000h
  ```

**sub** Instruction

- **sub** subtracts a *source* operand from the *destination* operand of the **same size**.
- Format:
  ```
  sub destination, source
  ```
- *Source* is unchanged; *destination* stores the difference. All the status flags are affected.
- The sizes must match and only one can be a memory location.
Flags Affected by \textbf{add} and \textbf{sub}

- If \textbf{add} or \textbf{sub} generates a result of zero, ZF is set
- If \textbf{add} or \textbf{sub} generates a negative result, SF is set.

Examples:

\begin{verbatim}
mov    ax, 10
sub    ax, 10 ; AX = 0, ZF = 1
mov    bx, 1
sub    bx, 2 ; BX = FFFF, SF = 1
\end{verbatim}

- \texttt{inc} and \texttt{dec} affect ZF but not CF.

\begin{verbatim}
mov    bl, 4Fh
add    bl, 0B1h ; BF = 00, ZF = 1, CF = 1
mov    ax, 0FFFFh
inc    ax ; ZF = 1 (CF unchanged)
\end{verbatim}

Flags Affected by \textbf{add} and \textbf{sub} (continued)

- The Overflow flag is useful when performing signed arithmetic:

\begin{verbatim}
mov    al, +126
add    al, 2 ; AL = 80h, OF = 1
126    01111110
+2     + 00000010
-128   0 10000000

mov    al, -128
sub    al, 2 ; AL = 7Eh, OF = 1
-128   10000000
-2     -11111110
-126   1 10000010
\end{verbatim}
Implementing Arithmetic Expressions

- Imagine we are implementing the statement

\[ Rval = -Xval + (Yval - Zval) \]

.data
Rval SDWORD  ?
Xval SDWORD  26
Yval SDWORD  30
Zval SDWORD  40

.code
; first term: -Xval
    mov  eax, Xval
    neg  eax ; EAX = -26

Implementing Arithmetic Expressions (continued)

; second term: (Yval - Zval)
    mov  ebx, Yval
    sub  ebx, Zval ; EBX = -10

; add the terms and store
    add  eax, ebx
    mov  Rval, eax ; Rval = -36
Indirect Operands

• An indirect operand is a register containing the offset for data in a memory location.
  – The register points to a label by placing its offset in that register
  – This is very convenient when working with arrays; it is just a matter of incrementing the address so that it points to the next array element.
  – The ESI, EDI, EBX, EBP, SI, DI, BX and BP registers can be used for indirect operands as well as the 32-bit general purpose registers (with a restriction on the ESP).

Indirect Operands: A Real Mode Example

• We create a string in memory at offset 0200 and set the BX to the string’s offset; we can process any element in the string by adding to the offset:

```
.data
        aString BYTE "ABCDEFG"

.code
    mov  bx, offset aString  ; BX = 0200
    add  bx, 5              ; BX = 0205
    mov  dl, [bx]           ; DL = 'F'
```

<table>
<thead>
<tr>
<th>0200</th>
<th>0205</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>G</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

aString
Indirect Operands: A Protected Mode Example

.data
val1 BYTE 10h
.code
    mov esi OFFSET val1
    mov al, [esi] ; AL = 10h
    mov [esi], bl ; The variable to
                   ; which ESI points is
                   ; changed
    mov esi, 0
    mov ax, [esi] ; General Protection
                   ; Error
    inc [esi] ; Error - needs size
    inc byte ptr [esi] ; Works!

Arrays

- Indirect arrays are useful when manipulating arrays:
.data
arrayB BYTE 10h, 20h, 30h
.code
    mov esi, OFFSET arrayB
    mov al, [esi] ; AL = 10h
    inc esi
    mov al, [esi] ; AL = 20h
    inc esi
    mov al, [esi] ; AL = 30h
Arrays of Doublewords

- If we use an array of 32-bit integers, we add 4 to ESI to address each subsequent array element:

```
.data
arrayD DWORD 10000h, 20000h, 30000h
.code
mov esi, OFFSET arrayD
mov eax, [esi] ; first #
add esi, 4
mov eax, [esi] ; second #
add esi, 4
mov eax, [esi] ; third #
```

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10200</td>
<td>10000h</td>
</tr>
<tr>
<td>10204</td>
<td>20000h</td>
</tr>
<tr>
<td>10208</td>
<td>30000h</td>
</tr>
</tbody>
</table>

Indexed Operands

- An indexed operand adds a constant to a register to generate an effective address.
- Any of the 32-bit general purpose register may be used as an index registers.
- There are two forms that are legal:
  - `constant[reg]`
  - `[constant+reg]`
- In both cases, we are combining the constant offset of a variable label with the contents of a register.
Indexed Operands – An Example

.data
arrayB BYTE 10h, 20h, 30h
arrayW WORD 1000h, 2000h, 3000h
.code
  mov esi, 0
  mov al, [arrayB+esi] ; AL = 10h

  mov esi, OFFSET arrayW
  mov ax, [esi] ; AX = 1000h
  mov ax, [esi+2] ; AX = 2000h
  mov ax, [esi+4] ; AX = 3000h

Transfer of Control

• A transfer of control is way of altering the order in which instructions are executed.
• The two basic ways are:
  – Unconditional transfer – the program branches to a statement elsewhere in the program
  – Conditional transfer – the program branches to a statement elsewhere in the program IF some condition is true.
**JMP Instruction**

- The JMP statement causes an unconditional transfer to the target address within the same code segment.
- The syntax is:
  
  ```
  JMP targetLabel
  ```
  
  where the targetLabel is the offset of an instruction elsewhere in the program.
- Example:
  ```
  top:
  ... ...
  jmp top; infinite loop
  ```

**LOOP Instruction**

- The LOOP instruction is used to end a block of statements that will be performed a predetermined number of times, with the number of times stored in the ECX (or CX) register.
- The syntax is:
  ```
  LOOP destination
  ```
  
  where destination is the label of the statement to which it jumps if the (E)CX register is nonzero.
- Because the (E)CX register controls the loop, it is extremely unwise to change it during the loop.
Nested Loops

- In writing nested loops, it is important to save the outer loop’s counter:

```assembly
.data
count DWORD ?
.code
    mov ecx, 100 ; set outer loop’s count
L1:    mov count, ecx ; save outer loop count
    mov ecx, 20 ; set inner loop count
L2:    ... ...
        loop L2 ; repeat inner loop
    mov ecx, count ; restore outer loop count
    loop L1
```

Summing An Integer Array

```assembly
TITLE Summing An Array (SumArray.asm)

INCLUDE Irvine32.inc
.data
intarray WORD 100h, 200h, 300h, 400h
.code
main PROC
    mov edi, OFFSET intarray ; address of intarray
    mov ecx, LENGTHOF intarray; ; loop counter
    mov ax, 0
```
Procedures

• As programs get larger and larger, it becomes necessary to divide them into a series of *procedures*.

• A procedure is a block of logically-related instruction that can be called by the main program or another procedure.

• Each procedure should have a single purpose and be able to do its job independent of the rest of the program.
Runtime Stack

• The runtime stack is a memory array that is managed directly by the CPU using the SS and ESP registers.
• In Protected mode, the SS register holds a segment descriptor and is not modified by user programs; the ESP register holds a 32-bit offset into some memory location on the stack.

The Intel Processor’s Stack

• The stack in an Intel processor is a special memory area.
  – The stack is a temporary holding area for addresses and data.
  – Most of the data held here allows a program to return (successfully) to the calling program and procedures or to pass parameters.
  – The stack resides in the stack segment.
Stack Operations - Push

Before

mov eax, 00000A5h
push eax

After

mov ebx, 00000001h
mov ecx, 00000002h
push ebx
push ecx

Stack Operations - Push (continued)
Stack Operations - Pop

Uses of the Stack

• There are several important uses of stacks in programs:
  – A stack makes an excellent temporary save area for registers, allowing a program to use them as a scratch area and then to restore them.
  – When a subroutine is called, the CPU saves a return address on the stack, allowing the program to return to the location after the procedure call.
  – When calling a procedure, you can push arguments on the stack, allowing the procedure to retrieve them.
  – High-level languages create an area on the stack inside subroutines where procedure store local variables and then discard them when it leaves the procedure.
Stack Operations - PUSH

- **PUSH Instruction**
  - Decrements ESP and copies a 16-bit or 32-bit register or memory operand onto the stack at the location indicated by SP.
  - With 80286+ processors, you can push an immediate operand onto the stack.
  - Examples:
    ```
push ax       ; push a 16-bit register operand
push ecx      ; push a 32-bit register operand
push memval   ; push a 16-bit memory operand
push 1000h    ; push an immediate operand
    ```

Stack Operations - POP

- **POP Instruction**
  - copies the contents of the stack pointed to by ESP into a register or variable and increments ESP.
  - CS and IP cannot be used as operands.
  - Examples:
    ```
pop cx        ; pop stack into 16-bit register
pop memval    ; pop stack into 16-bit memory operand
pop eds       ; pop stack into 32-bit register
    ```
In general, there are two types of subprograms: functions and procedures (or subroutines).

- **Functions** return a value (or **result**).
- **Procedures** (or **subroutines**) do not.
- The terms procedures and subroutines are used interchangeably although some languages use one term and others use the other.
- Calling a procedure implies that there is a return. Also implies is that the state of the program, (register values, etc.) are left unaffected when the program returns to the calling procedure or program.

### PROC and ENDP Directives

- PROC and ENDP mark the beginning and end of a procedure respectively.

```plaintext
.code
main proc
... ...
call MySub
... ...
main endp

; **Nb**: procedures cannot overlap
MySub proc ; one must have endp before
... ret
MySub endp ; the next can have proc
```
The **exit** Instruction

- While all other procedures end with the **ret** instruction, **exit** is used by the main procedure.
- **exit** is actually an not an instruction but an alias for
  
  `INVOKE ExitProcess, 0`
  
  the Windows system function for terminating programs
- In Irvine16.inc, it is defined as
  
  `mov    ah, 4ch`
  `int    21h`

---

Passing Parameters

- Passing arguments in registers
  - The most common method for passing parameter between the calling program (or procedure) and the procedures that it calls is through the registers
  - It is efficient because the called procedure has immediate and direct use of the parameters and registers are faster than memory.
  - Example: WriteInt
    
    ```
    .data
    aNumber    DWORD  234
    .code
    mov        eax, aNumber
    call        WriteInt
    ```
Preserving Registers

- Ordinarily procedures have the responsibility to preserve register contents.
  - This ensures that the main procedure has no surprises.
  - What would happen here if WriteInt modified ECX?

```assembly
.data
DECIMAL_RADIX = 10
LIST_COUNT=20
aList dw LIST_COUNT dup(?)
.code
mov ecx, LIST_COUNT
mov ebx, DECIMAL_RADIX
mov esi, offset aList
L1: mov eax, [si]
call WriteInt
add esi, size aList
loop L1
```

Using Registers to Return a Value

- Some functions will use a register as a method of returning a value to the calling procedure:

```assembly
SumOf proc
push eax
mov eax, ebx
add eax, ecx
pop eax ; Error – EAX reset to orig.
        ; value
ret
SumOf endp
```
Procedure **ArraySum**

ArraySum PROC

; Calculates the sum of an array of 32-bit integers.
; Receives: ESI - the array offset
; ECX = # of elements in array
; Returns EAX - the sum of the array

push esi ; save ESI, ECX
push ecx
mov eax, 0 ; Sum = 0
L1: add eax, [esi] ; Sum = Sum + x[i]
add esi, 4 ; Point to next integer
loop L1 ; Repeat for array size

pop ecx
pop esi
ret

ArraySum ENDP

Calling **ArraySum**

TITLE Driver for Array Sum (ArrayDr.asm)

INCLUDE Irvine32.inc

.data
array DWORD 10000h, 20000h, 30000h, 40000h
theSum DWORD ?

.code
main PROC
mov esi, OFFSET array ; ESI points to array
mov ecx, LENGTHOF array ; ECX = array count
call ArraySum ; calculate the sum
mov theSum, eax ; returned in EAX
call WriteHex ; Is it correct?
exit

main ENDP

ArraySum PROC... ...

Procedure goes here

END main
**CMP Instruction**

- The CMP instruction sets the flags *as if* it had performed subtraction on the operand.
- Neither operand is changed.
- The CMP instruction takes the forms:
  - CMP reg, reg
  - CMP mem, reg
  - CMP reg, mem
  - CMP mem, immed
  - CMP reg, immed

**CMP Results**

<table>
<thead>
<tr>
<th>CMP Results</th>
<th>ZF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>destination &lt; source</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>destination &gt; source</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>destination = source</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
**CMP Results**

<table>
<thead>
<tr>
<th>CMP Results</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>destination &lt; source</td>
<td>SF ≠ OF</td>
</tr>
<tr>
<td>destination &gt; source</td>
<td>SF = OF</td>
</tr>
<tr>
<td>destination = source</td>
<td>ZF = 1</td>
</tr>
</tbody>
</table>

**CMP Instruction : Examples**

- Subtracting 5-10 requires a borrow:
  ```
  mov ax, 5
  cmp ax, 10 ; CF = 1
  ```
- Subtracting 1000 from 1000 results in zero.
  ```
  mov ax, 1000
  mov cx, 1000
  cmp cx, ax ; ZF = 1
  ```
- Subtracting 0 from 105 produces a positive difference:
  ```
  mov si, 105
  cmp si, 0; ZF = 0 and CF = 0
  ```
Setting & Clearing Individual Flags

• Setting and Clearing the Zero Flag
  and al, 0 ; Set Zero Flag
  or al, 1 ; Clear Zero Flag

• Setting and Clearing the Sign Flag
  or al, 80h ; Set Sign Flag
  and al, 7fh ; Clear Sign Flag

• Setting and Clearing the Carry Flag
  stc ; Set Carry Flag
  clc ; Clear Carry Flag

• Setting and Clearing the Overflow Flag
  mov al, 7fH ; AL = +127
  inc al ; AL = 80H; OF = 1
  or eax, 0 ; Clear Overflow
  ; Flag
Conditional Structures – An Example

• Compare AL to Zero. Jump to L1 if the zero flag was set by the comparison:
  
  ```
  cmp al, 0
  jz L1
  ...
  ```

L1:

Conditional Structures – Another Example

• Perform a bitwise AND on the DL register. Jump to L2 if the Zero flag is clear:
  
  ```
  and dl, 10110000b
  jnz L2
  ...
  ```

L2:
**J_{cond} Instruction**

- A conditional jump instruction branches to a destination label when a flag condition is true.
- If the flag is false, the instruction immediately following the conditional jump is performed instead.
- The syntax is:
  \[ J_{cond} \text{ destination} \]

**Limitations of Conditional Jumps**

- Microsoft Macro assembler limits jumps to a label within the current procedure and within –128 to +127 of the current address.
- To jump to another procedure, you must use a global label:
  ```
  jc MyLabel ; Jump if Carry
              ; (flag is set)
  ...
  MyLabel::
  ```
Examples of Conditional Jumps

• In all three cases, the jump is made:

  mov ax, 5
  cmp ax, 5
  je L1 ; jump if equal

  mov ax, 5
  cmp ax, 6
  jl L1 ; jump if less

  mov ax, 5
  cmp ax, 4 ; jump if greater

Jumps based on General Comparisons

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Flags/Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>JZ</td>
<td>Jump if zero</td>
<td>ZF = 1</td>
</tr>
<tr>
<td>JE</td>
<td>Jump if equal</td>
<td>ZF = 1</td>
</tr>
<tr>
<td>JNZ</td>
<td>Jump if not zero</td>
<td>ZF = 0</td>
</tr>
<tr>
<td>JNE</td>
<td>Jump if not equal</td>
<td>ZF = 0</td>
</tr>
</tbody>
</table>
### Jumps based on General Comparisons

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Flags/Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>JC</td>
<td>Jump if carry</td>
<td>CF = 1</td>
</tr>
<tr>
<td>JNC</td>
<td>Jump if not carry</td>
<td>CF = 0</td>
</tr>
<tr>
<td>JCXZ</td>
<td>Jump if CX = 0</td>
<td>CX = 0</td>
</tr>
<tr>
<td>JECXZ</td>
<td>Jump if ECX = 0</td>
<td>ECX = 0</td>
</tr>
</tbody>
</table>

### Jumps based on General Comparisons

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Flags/Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP</td>
<td>Jump if Parity even</td>
<td>PF = 1</td>
</tr>
<tr>
<td>JNP</td>
<td>Jump if Parity odd</td>
<td>PF = 0</td>
</tr>
</tbody>
</table>
### Jumps based on Unsigned Comparisons

<table>
<thead>
<tr>
<th>Mnenomic</th>
<th>Description</th>
<th>Flag(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JA</td>
<td>Jump if above (op1 &gt; op2)</td>
<td>CF = 0 &amp; ZF = 0</td>
</tr>
<tr>
<td>JNBE</td>
<td>Jump if not below or equal</td>
<td>CF = 0 &amp; ZF = 0</td>
</tr>
<tr>
<td>JAE</td>
<td>Jump if above or equal</td>
<td>CF = 0</td>
</tr>
<tr>
<td>JNB</td>
<td>Jump if not below</td>
<td>CF = 0</td>
</tr>
</tbody>
</table>

### Jumps based on Unsigned Comparisons

<table>
<thead>
<tr>
<th>Mnenomic</th>
<th>Description</th>
<th>Flag(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JB</td>
<td>Jump if below (op1 &lt; op2)</td>
<td>CF = 1</td>
</tr>
<tr>
<td>JNAE</td>
<td>Jump if not above</td>
<td>CF = 1</td>
</tr>
<tr>
<td>JBE</td>
<td>Jump if below or equal</td>
<td>CF = 1 or ZF = 1</td>
</tr>
<tr>
<td>JNA</td>
<td>Jump if not above</td>
<td>CF = 1 or ZF = 1</td>
</tr>
</tbody>
</table>
### Jumps based on Signed Comparisons

<table>
<thead>
<tr>
<th>Mnenomic</th>
<th>Description</th>
<th>Flag(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JG</td>
<td>Jump if greater</td>
<td>SF = 0 &amp; ZF = 0</td>
</tr>
<tr>
<td>JNLE</td>
<td>Jump if not less than or equal</td>
<td>SF = 0 &amp; ZF = 0</td>
</tr>
<tr>
<td>JGE</td>
<td>Jump if greater than or equal</td>
<td>SF = OF</td>
</tr>
<tr>
<td>JNL</td>
<td>Jump if not less than</td>
<td>SF = OF</td>
</tr>
</tbody>
</table>

### Jumps based on Signed Comparisons

<table>
<thead>
<tr>
<th>Mnenomic</th>
<th>Description</th>
<th>Flag(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JL</td>
<td>Jump if less</td>
<td>SF &lt;&gt; OF</td>
</tr>
<tr>
<td>JNGE</td>
<td>Jump if not greater than or equal</td>
<td>SF &lt;&gt; OF</td>
</tr>
<tr>
<td>JLE</td>
<td>Jump if less than or equal</td>
<td>ZF = 1 or SF = &lt;&gt; OF</td>
</tr>
<tr>
<td>JNG</td>
<td>Jump if not greater than</td>
<td>ZF = 1 or SF = &lt;&gt; OF</td>
</tr>
</tbody>
</table>
Jumps based on Signed Comparisons

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Flag(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JS</td>
<td>Jump if signed (op1 is negative)</td>
<td>SF = 1</td>
</tr>
<tr>
<td>JNS</td>
<td>Jump if not signed</td>
<td>SF = 0</td>
</tr>
<tr>
<td>JO</td>
<td>Jump if overflow</td>
<td>OF = 1</td>
</tr>
<tr>
<td>JNO</td>
<td>Jump if not overflow</td>
<td>OF = 0</td>
</tr>
</tbody>
</table>

Example – Smallest of Three Integers

```assembly
.data
V1      WORD  ?
V2      WORD  ?
V3      WORD  ?
.code
    mov  ax, V1 ; assume that V1 is smallest
    cmp  ax, V2 ; IF AX <= V2 then
    jbe  L1  ; jump to L1
    mov  ax, V2 ; else move V2 to AX
    L1:   cmp  ax, V3 ; if AX <= V3 then
             jbe  L2  ; jump to L3
           mov  ax, V3 ; else move to V3 to AX
    L2:    ; smallest is in AX
```
Writing IF-THEN

In C++:
```
if (x > y)
  z = 0;
```

In Assembler:
```
mov ax, x
cmp ax, y
jng L1
mov ax, 0
mov z, ax
L1:
```

Writing IF-THEN-ELSE

In C++:
```
if (x > y)
  z = x;
else
  z = y;
```

In Assembler:
```
mov ax, x
cmp ax, y
jng L1
mov ax, x
jmp L2
L1:
  mov ax, y
L2:
  mov z, ax
```
Writing WHILE loops

In C++:
```cpp
while (x <= y)
    x = x + 3;
```

In Assembler
```
L1:
mov ax, x
cmp ax, y
jg L2
mov ax, x
add ax, 3
jmp L1
```
```
L2:
```

**IMUL Instruction**

- The **IMUL** instruction multiplies an 8-, 16, or 32-bit *signed* operand by either the AL, AX or EAX register (depending on the operand’s size).

- The instruction formats are:
  - **IMUL r/m8**
  - **IMUL r/m16**
  - **IMUL r/m32**

  $\begin{array}{c}
  \text{IMUL} \ \ r/m8 \\
  \text{IMUL} \ \ r/m16 \\
  \text{IMUL} \ \ r/m32 \\
  \end{array}$

  $\begin{array}{c}
  \text{EAX} \\
  \text{EDX} \\
  \text{EAX} \\
  \end{array}$
**IMUL Instruction (continued)**

- The **IMUL** instruction sets the Carry and Overflow flags if the upper half of the product is not a sign extension of the low-order product equal to zero.
- E.g., if AX is multiplied by a 16-bit multiplier, the product is stored in DX:AX. IF the AX contains a negative value and the DX is not all 1s, the Carry and Overflow flags are set.

**IMUL Instruction - Examples**

- 8-bit signed multiplication (48 * 4)
  ```
  mov al, 48
  mov bl, 4
  imul bl ; AX = 00C0h, OF = 1
  ```
- 16-bit signed multiplication (-4 * 4)
  ```
  mov al, -4
  mov bl, 4
  imul bl ; AX = FFF0h, OF = 0
  ```
- 32-bit signed multiplication (12345h*1000h)
  ```
  mov eax, +4823424
  mov ebx, -423
  imul ebx ; EDX:EAX =
             ; FFFFFFFF86636D80h, OF = 0
  ```
CBW, CWD and CDQ Instructions

- **CBW** intends the sign bit of AL into the AH register.
- **CWD** intends the sign bit of AX into the DX register.
- **CDQ** intends the sign bit of EAX into the EDX register.

```
data
byteVal SBYTE -65 ; 9Bh
wordVal SWORD -65 ; FF9Bh
dwordVal SDWORD -65 ; FFFFFFF9Bh
code
  mov al, byteVal ; AL = 9Bh
  cbw ; AX = FF9Bh
  mov ax, wordVal ; AX = FF9Bh
  cwd ; DX:AX = FFFFFFF9Bh
  mov eax, dwordVal; EAX = FFFFFFF9Bh
  cdq ; EDX:EAX = FFFFFFFFFFFFFFF9Bh
```

IDIV Instruction

- The **IDIV** instruction divides an 8-, 16, or 32-bit *signed* divisor into either the AL, AX or EAX register (depending on the operand’s size).
- Signed division requires that the sign bit be extend into the AH, DX or EDX (depending on the operand’s size) using **CBW, CWD or CDQ**.
IDIV Instruction – 8-bit Example

.data
byteVal SBYTE -48
.code
  mov   al, byteVal ; dividend
  cbw   ; extend Al into AH
  mov   bl, 5      ; divisor
  idiv  bl         ; AL = -9, AH = -3

IDIV Instruction – 16-bit Example

.data
wordVal SWORD -5000
.code
  mov   ax, wordVal ; dividend, low
  cwd   ; extend AX into DX
  mov   bx, 256    ; divisor
  idiv  bx         ; quotient AX = -19
                   ; rem. DX = -136
IDIV Instruction – 32-bit Example

.data
wordVal SWORD -50000

.code
    mov   eax, dwordVal ; dividend, low
    cdq    ; extend EAX into EDX
    mov   ebx, 256 ; divisor
    idiv  bx    ; quotient EAX = -195
                 ; remainder EDX = -80