Basic Elements of Assembly Language

An assembly language program is composed of:
• Constants
• Expressions
• Literals
• Reserved Words
• Mnemonics
• Identifiers
• Directives
• Instructions
• Comments
Integer Constants

• Integer constants can be written in decimal, hexadecimal, octal or binary, by adding a radix (or number base) suffix to the end.

• Radix Suffices:
  – d          decimal (the default)
  – h          hexadecimal
  – q or o     octal
  – b          binary

Examples of Integer Constants

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

- An integer expressions is a mathematical expressions involving integer values and integer operators.
- The expressions must be one that can be stored in 32 bits (or less).
- The precedence:
  - ( ) Expressions in Parentheses
  - +, - Unary Plus and minus
  - *, /, Mod Multiply, Divide, Modulus
  - +, - Add, Subtract

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) * (6 – 1)</td>
<td>-35</td>
</tr>
<tr>
<td>–3 + 4 * 6 – 1</td>
<td>20</td>
</tr>
<tr>
<td>4 + 5 * 2</td>
<td>1</td>
</tr>
<tr>
<td>–5 + 2</td>
<td></td>
</tr>
<tr>
<td>12 – 1 MOD 5</td>
<td></td>
</tr>
<tr>
<td>(4 + 2) * 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
    - -44.2E+05
    - 26.E5
  - **Encoded reals**, which are represented exactly as they are stored:
    - 3F800000r

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.
**Identifiers**

- **Identifiers** are *names* that the programmer chooses to represent variables, constants, procedures or labels.
- Identifiers:
  - can have 1 to 247 characters
  - are not case-sensitive
  - begin with a letter, underscore, @ or $ and can also contain digits after the first character.
  - cannot be reserved words

**Examples of Identifiers**

```
var1           open_file
_main          _12345
@@myfile       $first
Count          MAX
xVal
```
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:
  \begin{verbatim}
  mov ax, [9020]
  \end{verbatim}
  vs.
  \begin{verbatim}
  mov ax, MyVariable
  \end{verbatim}

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:
  \begin{verbatim}
  target:
  mov ax, bx
  ...
  jmp target
  \end{verbatim}
Data Labels

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

Instruction Mnemonics

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

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

<table>
<thead>
<tr>
<th>Operands</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>constants</td>
<td>96</td>
</tr>
<tr>
<td>constant expressions</td>
<td>2 + 4</td>
</tr>
<tr>
<td>registers</td>
<td>eax</td>
</tr>
<tr>
<td>memory locations</td>
<td>count</td>
</tr>
</tbody>
</table>

Operands and Instructions

- All instructions have a predetermined number of operands.
- Some instructions use no operands:
  \text{stc}; \text{set the Carry Flag}
- Some instructions use one operand:
  \text{inc} \ ax; \text{add 1 to AX}
- Some instructions use two operands:
  \text{mov} \ count, bx; \text{add BX to count}
Comments

- Comments serve as a way for programmers to document their programs,
- Comments can be specified:
  - on a single line, beginning with a semicolon until the end of the line:
    ```
    stc ; set the Carry Flag
    ```
  - in a block beginning with the directive COMMENT and a user-specified symbol wchih also ends the comment:
    ```
    COMMENT !
    ... ...
    !
    ```

Example: Adding Three Numbers

```assembly
TITLE Add And Subtract (AddSub.asm)
; This program adds and subtracts 32-bit integers.
INCLUDE Irvine32.inc
.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=00000026  CF=0  SF=0  ZF=0  OF=0

An Alternative AddSub

TITLE Add And Subtract  (AddSubAlt.asm)
; This program adds and subtracts 32-bit integers.

.386    ; Minimum CPU to run this is an Intel 386
.MODEL  flat, stdcall  ; Protected mode program
                     ; using call Windows calls
.STACK 4096       ; The stack is 4096 bytes in size
ExitProcess PROTO, dwExitCode:DWORD
DumpRegs PROTO     ; ExitProcess is an MS-Windows
                   ; procedure
                   ; DumpRegs is a procedure in
                   ; Irvine32.inc
                   ; dwExitCode is a 32-bit value
.code
main PROC

mov eax, 10000h
add eax, 40000h
sub eax, 20000h
call DumpRegs

INVOKEx ExitProcess, 0 ; INVOKEx is a directive
; that calls procedures.
; Call the ExitProcess
; procedure
; Pass back a return
; code of zero.

main ENDP
end main

A Program Template

TITLE Program Template  (Template.asm)
; Program Description:
; Author:
; Creation Date:
; Revisions:
; Date: Modified by:
INCLUDE Irvine32.inc
.data
; (insert variables here)
.code
main PROC
; (insert executable instructions here)
exit
main ENDP
; (insert additional procedures here)
   END   main
Assembling and Linking the Program

- A 32-bit assembly language program can be assembled and linked in one step by typing: 
  ```
  make32 filename
  ```
- A 16-bit assembly language program can be assembled and linked in one step by typing: 
  ```
  make16 filename
  ```
- Example: 
  ```
  make32 addsub
  ```
Other Files

- In addition to the .asm file (assembler source code), .obj file (object code) and .exe file (executable file), there are other files created by the assembler and linker:
  - **.LST (listing) file** – contains the source code and object code of the program
    - **.MAP file** – contains information about the segments being linked
    - **.PDB (Program database) file** – contains supplemental information about the program

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>
Intrinsic Data Types (continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>FWORD</td>
<td>48-bit integer; Far Pointer in Protected mode</td>
</tr>
<tr>
<td>QWORD</td>
<td>64-bit integer</td>
</tr>
<tr>
<td>TBYTE</td>
<td>80-bit (ten-byte) integer</td>
</tr>
<tr>
<td>REAL4</td>
<td>32-bit (4-byte) IEEE short real</td>
</tr>
<tr>
<td>REAL8</td>
<td>64-bit (8-byte) IEEE long real</td>
</tr>
<tr>
<td>REAL10</td>
<td>80-bit (10-byte) IEEE extended real</td>
</tr>
</tbody>
</table>

Defining Data

- A data definition statement allocates storage in memory for variables.
- We write:
  
  \[ \text{name} \ \text{directive} \ \text{initializer} \ [. \ \text{initializer}] \]
- There must be at least one initializer.
- If there is no specific initial value, we use the expression `?`, which indicates no special value.
- All initializers are converted to binary data by the assembler.
Defining 8-bit Data

- BYTE and SBYTE are used to allocate storage for an unsigned or signed 8-bit value:
  
<table>
<thead>
<tr>
<th>Value</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>value1</code></td>
<td>BYTE</td>
<td>character constant</td>
</tr>
<tr>
<td><code>value2</code></td>
<td>BYTE</td>
<td>0, smallest unsigned byte</td>
</tr>
<tr>
<td><code>value3</code></td>
<td>BYTE</td>
<td>255, largest unsigned byte</td>
</tr>
<tr>
<td><code>value4</code></td>
<td>SBYTE</td>
<td>-128, smallest signed byte</td>
</tr>
<tr>
<td><code>value5</code></td>
<td>SBYTE</td>
<td>+127, largest signed byte</td>
</tr>
<tr>
<td><code>value6</code></td>
<td>BYTE</td>
<td>?, no initial value</td>
</tr>
<tr>
<td><code>value7</code></td>
<td>BYTE</td>
<td>10h, offset is zero</td>
</tr>
<tr>
<td><code>value8</code></td>
<td>BYTE</td>
<td>20h, offset is 1</td>
</tr>
</tbody>
</table>
  
- **db Directive**

  - db is the older directive for allocating storage for 8-bit data.
  - It does not distinguish between signed and unsigned data:
    
    | Value   | Type | Description                  |
    |---------|------|-------------------------------|
    | `val1`  | db   | 255, unsigned byte           |
    | `val2`  | db   | -128, signed byte            |
Multiple Initializers

• If a definition has multiple initializers, the label is the offset for the first data item:

```
.data
list BYTE 10, 20, 30, 40
```

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>10</td>
</tr>
<tr>
<td>0001</td>
<td>20</td>
</tr>
<tr>
<td>0002</td>
<td>30</td>
</tr>
<tr>
<td>0003</td>
<td>40</td>
</tr>
</tbody>
</table>

Multiple Initializers (continued)

• Not all definitions need labels:

```
.data
list BYTE 10, 20, 30, 40
BYTE 50, 60, 70, 80
BYTE 81, 82, 83, 84
```

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>10</td>
</tr>
<tr>
<td>0001</td>
<td>20</td>
</tr>
<tr>
<td>0002</td>
<td>30</td>
</tr>
<tr>
<td>0003</td>
<td>40</td>
</tr>
<tr>
<td>0004</td>
<td>50</td>
</tr>
<tr>
<td>0005</td>
<td>60</td>
</tr>
</tbody>
</table>
Multiple Initializers (continued)

- The different initializers can use different radixes:

```
.data
list1 BYTE 10, 32, 41h, 00100010b
list2 BYTE 0aH, 20H, 'A', 22h
```

- list1 and list2 will have the identical contents, albeit at different offsets.

Defining Strings

- To create a string data definition, enclose a sequence of characters in quotation marks.
- The most common way to end a string is a null byte (0):

```
greeting1 BYTE "Good afternoon", 0
```

is the same as

```
greeting1 BYTE 'G', 'o', 'o', ... 0
```
Defining Strings (continued)

- Strings can be spread over several lines:

```plaintext
greeting2 BYTE "Welcome to the Encryption"
    BYTE " Demo program"
    BYTE "created by Kip Irvine",
    0dh, 0ah
    BYTE " If you wish to modify this"
    " program, please"
    BYTE "send me a copy", 0dh, 0ah
```

**Concatenates two lines**

Using **dup**

- **DUP** repeats a storage allocation however many times is specified:

```plaintext
BYTE 20 DUP(0) ; 20 bytes of zero
BYTE 20 DUP(?) ; 20 bytes uninitialized
BYTE 2 DUP(“STACK”)
    ; 20 bytes “STACKSTACK”
```
Defining 16-bit Data

- The **WORD** and **SWORD** directives allocate storage of one or more 16-bit integers:
  ```
  word1  WORD  65535 ; largest unsigned value  
  word2  SWORD -32768; smallest signed value  
  word3  WORD  ? ; uninitialized value
  ```

- The **dw** directive can be used to allocate storage for either signed or unsigned integers:
  ```
  val1   dw  65535 ; unsigned  
  val2   dw -32768 ; signed
  ```

Arrays of Words

- You can create an array of word values by listing them or using the **DUP** operator:
  ```
  myList  WORD  1, 2, 3, 4, 5
  ```

<table>
<thead>
<tr>
<th>Offset</th>
<th>0000</th>
<th>0002</th>
<th>0004</th>
<th>0006</th>
<th>0008</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value:</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

  ```
  array  WORD  5   DUP(?)  
  ; 5 values, uninitialized
  ```
Defining 32-bit Data

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

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

• The **dd** directive can be used to allocated storage for either signed or unsigned integers:

```cpp
val1   dd  12345678h ; unsigned
val2   dw  -21474836648 ; signed
```

Arrays of Doublewords

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

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>0004</td>
<td>2</td>
</tr>
<tr>
<td>0008</td>
<td>3</td>
</tr>
<tr>
<td>000C</td>
<td>4</td>
</tr>
<tr>
<td>0010</td>
<td>5</td>
</tr>
</tbody>
</table>
Defining 64-bit Data

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

- The **dq** directive can be used to allocated storage:
  
  ```assembly
  quad1 dq 1234567812345678h
  ```

Defining 80-bit Data

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

- The **dq** directive can be used to allocated storage:
  
  ```assembly
  val1 dt 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

<table>
<thead>
<tr>
<th>rVal1</th>
<th>REAL4</th>
<th>-2.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>rVal2</td>
<td>REAL8</td>
<td>3.2E-260</td>
</tr>
<tr>
<td>rVal3</td>
<td>REAL10</td>
<td>4.6E+4096</td>
</tr>
<tr>
<td>ShortArray</td>
<td>REAL4</td>
<td>20 DUP (?)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rVal1</th>
<th>DD</th>
<th>-1.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>rVal2</td>
<td>dq</td>
<td>3.2E-260</td>
</tr>
<tr>
<td>rVal3</td>
<td>dt</td>
<td>4.6E+4096</td>
</tr>
</tbody>
</table>
## 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>$1.18 \times 10^{-38}$ to $3.40 \times 10^{38}$</td>
</tr>
<tr>
<td>Long Real</td>
<td>15</td>
<td>$2.23 \times 10^{-308}$ to $1.79 \times 10^{308}$</td>
</tr>
<tr>
<td>Extended Real</td>
<td>19</td>
<td>$3.37 \times 10^{-4932}$ to $1.18 \times 10^{4932}$</td>
</tr>
</tbody>
</table>

## Little Endian Order

- Consider the number 12345678h:

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

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

- Equate directives allows constants and literals to be given symbolic names.
- The directives are:
  - Equal-Sign Directive
  - EQU Directive
  - TEXTEQU Directive

Equal-Sign Directive

- The equal-sign directive creates a symbol by assigning a numeric expression to a name.
- The syntax is:
  \[ name = expression \]
- The equal sign directive assigns no storage; it just ensures that occurrences of the name are replaced by the expression.
Equal-Sign Directive (continued)

• Expression must be expressable as 32-bit integers (this requires a .386 or higher directive).
• Examples:
  prod = 10 * 5 ; Evaluates an expression
  maxInt = 7FFFh ; Maximum 16-bit signed value
  minInt = 8000h ; Minimum 16-bit signed value
  maxUInt = 0FFFh ; Maximum 16-bit unsigned value
  String = 'XY' ; Up to two characters allowed
  Count = 500
  endvalue = count + 1 ;Can use a predefined symbol

  .386
  maxLong = 7FFFFFFFh
  ; Maximum 32-bit signed value
  minLong = 80000000h; Minimum 32-bit signed value
  maxULong = 0xffffffff; Maximum 32-bit unsigned value

Equal-Sign Directive (continued)

• A symbol defined with an equal-sign directive can be redefined with a different value later within the same program:
  - Statement: Assembled as:
    count = 5
    mov al, count mov al, 5
    mov dl, al mov al, dl
    count = 10
    mov cx, count mov cx, 10
    mov dx, count mov dx, 10
    count = 2000
    mov ax, count mov ax, 2000
EQU Directive

- The EQU Directive assigns a symbolic name to a string or numeric constant.
- Symbols defined using EQU cannot be redefined.
- Expressions are evaluated as integer values, but floating point values are evaluated as strings.
- Strings may be enclosed in the brackets `< >` to ensure their correct interpretation.
- Examples:

<table>
<thead>
<tr>
<th>Example</th>
<th>Type of value</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxint equ 32767</td>
<td>Numeric</td>
</tr>
<tr>
<td>maxuint equ 0FFFFh</td>
<td>Numeric</td>
</tr>
<tr>
<td>count equ 10 * 20</td>
<td>Numeric</td>
</tr>
<tr>
<td>float1 equ <code>&lt;2.345&gt;</code></td>
<td>String</td>
</tr>
</tbody>
</table>

TEXTEQU Directive

- The TEXTEQU directive assigns a name to a sequence of characters.
- Syntax:
  
  ```
  name TEXTEQU <text>
  name TEXTEQU textmacro
  name TEXTEQU %constExpr
  ```

  - Textmacro is a predefined text macro (*more about this later*)
  - constExpr is a numeric expression which is evaluated and used as a string.
- Example:

  ```
  continueMsg textequ <"Do you wish to continue?">
  .data
  prompt1    db     ContinueMsg
  ```
TEXTEQU Examples

;Symbol declarations:
move textequ <mov>
address textequ <offset>

; Original code:
move bx, address value
move al, 20

; Assembled as:
mov bx, offset value
mov al, 20

TEXTEQU Examples (continued)

.data
myString BYTE "A string", 0

.code
p1 textequ <offset MyString>
mov bx, p1
mov al, 20 ; bx = offset myString
p1 textequ <0>
mov si, p1 ; si = 0
Real-Address Mode
Programming

TITLE Add And Subtract  (AddSub3.asm)
; This program adds and subtracts 32-bit
; integers and stores the sum in a
; variable. Target : Real Mode
INCLUDE  Irvine16.inc
.data
val1    DWORD10000h
val2    DWORD40000h
val3    DWORD20000h
finalVal DWORD?

.code
main PROC
  mov ax, @data
  mov ds, ax    ; initialize the data
                ; segment register
  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