Overview of Assembly Language

Chapter 3

S. Dandamudi

Outline

- Assembly language statements
- Data allocation
- Where are the operands?
  - Addressing modes
    - Register
    - Immediate
    - Direct
    - Indirect
- Data transfer instructions
  - **`mov`**, **`xchg`**, and **`xlat`**
  - PTR directive
- Overview of assembly language instructions
  - Arithmetic
  - Conditional
  - Logical
  - Shift
  - Rotate
- Defining constants
  - EQU and = directives
- Illustrative examples
- Performance: When to use the **`xlat`** instruction

To be used with S. Dandamudi, "Introduction to Assembly Language Programming," Springer-Verlag, 1998.

Assembly Language Statements

- Three different classes
  - Instructions
    - » Tell CPU what to do
    - » Executable instructions with an op-code
  - Directives (or pseudo-ops)
    - » Provide information to assembler on various aspects of the assembly process
    - » Non-executable
      - – Do not generate machine language instructions
  - Macros
    - » A shorthand notation for a group of statements
    - » A sophisticated text substitution mechanism with parameters

Assembly Language Statements (cont'd)

- Assembly language statement format:

  `[label] mnemonic [operands] [;comment]`

  - Typically one statement per line
  - Fields in [ ] are optional
  - **`label`** serves two distinct purposes:
    - » To label an instruction
      - – Can transfer program execution to the labeled instruction
    - » To label an identifier or constant
  - **`mnemonic`** identifies the operation (e.g. **`add`**, **`or`**)
  - **`operands`** specify the data required by the operation
    - » Executable instructions can have zero to three operands

Assembly Language Statements (cont'd)

* **comments**
  - » Begin with a semicolon (;) and extend to the end of the line

Examples

```
repeat:  inc     result   ; increment result
CR       EQU     0DH      ; carriage return character
```

- White space can be used to improve readability

```
repeat:
   inc     result   ; increment result
```


Data Allocation

- Variable declaration in a high-level language such as C

```
char    response
int     value
float   total
double  average_value
```

  specifies
  - » Amount storage required (1 byte, 2 bytes, …)
  - » Label to identify the storage allocated (response, value, …)
  - » Interpretation of the bits stored (signed, floating point, …)
    - – Bit pattern **1000 1101 1011 1001** is interpreted as
      - ✈ –29,255 as a signed number
      - ✈ 36,281 as an unsigned number

Data Allocation (cont'd)

- In assembly language, we use the *define* directive
  - Define directive can be used
    - » To reserve storage space
    - » To label the storage space
    - » To initialize
    - » But *no interpretation* is attached to the bits stored
      - – Interpretation is up to the program code
  - Define directive goes into the .DATA part of the assembly language program
- Define directive format

```
[var-name]  D?  init-value [,init-value],...
```


Data Allocation (cont'd)

- Five define directives

```
DB  Define Byte        ;allocates 1 byte
DW  Define Word        ;allocates 2 bytes
DD  Define Doubleword  ;allocates 4 bytes
DQ  Define Quadword    ;allocates 8 bytes
DT  Define Ten bytes   ;allocates 10 bytes
```

Examples

```
sorted    DB   'y'
response  DB   ?   ;no initialization
value     DW   25159
float1    DQ   1.234
```

Data Allocation (cont'd)

- Multiple definitions can be abbreviated

Example

```
message  DB   'B'
         DB   'y'
         DB   'e'
         DB   0DH
         DB   0AH
```

can be written as

```
message  DB   'B','y','e',0DH,0AH
```

- More compactly as

```
message  DB   'Bye',0DH,0AH
```


Data Allocation (cont'd)

- Multiple definitions can be cumbersome to initialize data structures such as arrays

Example

To declare and initialize an integer array of 8 elements

```
marks  DW   0,0,0,0,0,0,0,0
```

- What if we want to declare and initialize to zero an array of 200 elements?
  - There is a better way of doing this than repeating zero 200 times in the above statement
    - Assembler provides a directive to do this (DUP directive)

Data Allocation (cont'd)

- Multiple initializations
  - The DUP assembler directive allows multiple initializations to the same value
  - Previous marks array can be compactly declared as

```
marks  DW   8 DUP (0)
```

Examples

```
table1  DW  10 DUP (?)   ;10 words, uninitialized
message DB  3  DUP ('Bye!');12 bytes, initialized
                          ;    as Bye!Bye!Bye!
Name1   DB  30 DUP ('?')  ;30 bytes, each
                          ;     initialized to ?
```


Data Allocation (cont'd)

- The DUP directive may also be nested

Example

```
stars DB 4 DUP(3 DUP ('*'),2 DUP ('?'),5 DUP ('!'))
```

Reserves 40-bytes space and initializes it as

```
***??!!!!!***??!!!!!***??!!!!!***??!!!!!
```

Example

```
matrix  DW  10 DUP (5 DUP (0))
```

defines a 10X5 matrix and initializes its elements to zero

This declaration can also be done by

```
matrix  DW  50 DUP (0)
```

Data Allocation (cont'd)

Symbol Table

* Assembler builds a symbol table so we can refer to the allocated storage space by the associated label

Example

```
.DATA
value    DW  0
sum      DD  0
marks    DW  10 DUP (?)
message  DB  'The grade is:',0
char1    DB  ?
```

name	offset
value	0
sum	2
marks	6
message	26
char1	40

Data Allocation (cont'd)

Correspondence to C Data Types

Directive	C data type
DB	char
DW	int, unsigned
DD	float, long
DQ	double
DT	internal intermediate float value

Data Allocation (cont'd)

LABEL Directive

* LABEL directive provides another way to name a memory location
* Format:

```
name   LABEL   type
```

`type` can be

```
BYTE     1 byte
WORD     2 bytes
DWORD    4 bytes
QWORD    8 bytes
TWORD    10 bytes
```


Data Allocation (cont'd)

LABEL Directive

Example

```
.DATA
 count  LABEL  WORD
 Lo-count  DB  0
 Hi_count  DB  0
.CODE
 ...
 mov  Lo_count,AL
 mov  Hi_count,CL
```

* `count` refers to the 16-bit value
* `Lo_count` refers to the low byte
* `Hi_count` refers to the high byte

Where Are the Operands?

- Operands required by an operation can be specified in a variety of ways
- A few basic ways are:
  - operand in a register
    - register addressing mode
  - operand in the instruction itself
    - immediate addressing mode
  - operand in memory
    - variety of addressing modes
      - ✈direct and indirect addressing modes
  - operand at an I/O port

Where Are the Operands? (cont'd)

Register addressing mode

- Most efficient way of specifying an operand
  - » operand is in an internal register

Examples

```
mov     EAX,EBX
mov     BX,CX
```

- The **mov** instruction

  mov destination,source

  copies data from **source** to **destination**

Where Are the Operands? (cont'd)

Immediate addressing mode

* Data is part of the instruction
  - » operand is located in the code segment along with the instruction
  - » Efficient as no separate operand fetch is needed
  - » Typically used to specify a constant

Example

```
mov     AL,75
```

* This instruction uses register addressing mode for specifying the *destination* and immediate addressing mode to specify the *source*

Where Are the Operands? (cont'd)

Direct addressing mode

* Data is in the data segment
  - » Need a logical address to access data
    - – Two components: segment:offset
  - » Various addressing modes to specify the offset component
    - – offset part is referred to as the ***effective address***
* The offset is specified directly as part of the instruction
* We write assembly language programs using memory labels (e.g., declared using DB, DW, LABEL,...)
  - » Assembler computes the offset value for the label
    - – Uses symbol table to compute the offset of a label

Where Are the Operands? (cont'd)

Direct addressing mode

Examples

`mov AL,response`

» Assembler replaces **`response`** by its effective address (i.e., its offset value from the symbol table)

`mov table1,56`

» **`table1`** is declared as

```
table1   DW   20 DUP (0)
```

» Since the assembler replaces **`table1`** by its effective address, this instruction refers to the first element of **`table1`**

– In C, it is equivalent to

```
table1[0] = 56
```


Where Are the Operands? (cont'd)

Direct addressing mode

- Problem with direct addressing
  - Useful only to specify simple variables
  - Causes serious problems in addressing data types such as arrays
    - » As an example, consider adding elements of an array
      - – Direct addressing does not facilitate using a loop structure to iterate through the array
      - – We have to write an instruction to add each element of the array
- Indirect addressing mode remedies this problem

Where Are the Operands? (cont'd)

Indirect addressing mode

- The offset is specified indirectly via a register
  - Sometimes called *register indirect* addressing mode
  - For 16-bit addressing, the offset value can be in one of the three registers: BX, SI, or DI
  - For 32-bit addressing, all 32-bit registers can be used

Example

```
mov     AX,[BX]
```

  - Square brackets [ ] are used to indicate that BX is holding an offset value
    - » BX contains a pointer to the operand, not the operand itself

Where Are the Operands? (cont'd)

- Using indirect addressing mode, we can process arrays using loops

Example: Summing array elements

  - Load the starting address (i.e., offset) of the array into BX
  - Loop for each element in the array
    - » Get the value using the offset in BX
      - – Use indirect addressing
    - » Add the value to the running total
    - » Update the offset in BX to point to the next element of the array

Where Are the Operands? (cont'd)

Loading offset value into a register

- Suppose we want to load BX with the offset value of **`table1`**
- We cannot write

```
mov     BX,table1
```

- Two ways of loading offset value
  - » Using OFFSET assembler directive
    - – Executed only at the assembly time
  - » Using **`lea`** instruction
    - – This is a processor instruction
    - – Executed at run time

Where Are the Operands? (cont'd)

Loading offset value into a register

- Using OFFSET assembler directive
  - ∗ The previous example can be written as

```
mov     BX,OFFSET table1
```

- Using **`lea`** (load effective address) instruction
  - ∗ The format of **`lea`** instruction is

```
lea     register,source
```

  - ∗ The previous example can be written as

```
lea     BX,table1
```

Where Are the Operands? (cont'd)

Loading offset value into a register

Which one to use -- OFFSET or **`lea`**?

- Use OFFSET if possible
  - » OFFSET incurs only one-time overhead (at assembly time)
  - » **`lea`** incurs run time overhead (every time you run the program)
- May have to use **`lea`** in some instances
  - » When the needed data is available at run time only
    - – An index passed as a parameter to a procedure
  - » We can write

```
lea   BX,table1[SI]
```

    to load BX with the address of an element of **`table1`** whose index is in SI register
  - » We cannot use the OFFSET directive in this case

Data Transfer Instructions

- We will look at three instructions
  - **`mov`** (move)
    - » Actually copy
  - **`xchg`** (exchange)
    - » Exchanges two operands
  - **`xlat`** (translate)
    - » Translates byte values using a translation table
- Other data transfer instructions such as
  - **`movsx`** (move sign extended)
  - **`movzx`** (move zero extended)

  are discussed in Chapter 6

Data Transfer Instructions (cont'd)

The `MOV` instruction

* The format is

  `mov destination,source`

  » Copies the value from **`source`** to **`destination`**

  » **`source`** is not altered as a result of copying

  » Both operands should be of same size

  » **`source`** and **`destination`** cannot both be in memory

  – Most Pentium instructions do not allow both operands to be located in memory

  – Pentium provides special instructions to facilitate memory-to-memory block copying of data

Data Transfer Instructions (cont'd)

The `MOV` instruction

* Five types of operand combinations are allowed:

Instruction type	Example
`mov register,register`	`mov DX,CX`
`mov register,immediate`	`mov BL,100`
`mov register,memory`	`mov BX,count`
`mov memory,register`	`mov count,SI`
`mov memory,immediate`	`mov count,23`

* The operand combinations are valid for all instructions that require two operands

Data Transfer Instructions (cont'd)

Ambiguous moves: PTR directive

- For the following data definitions

```
.DATA
table1    DW    20 DUP (0)
status    DB    7 DUP (1)
```

  the last two **MOV** instructions are ambiguous

```
mov    BX,OFFSET table1
mov    SI,OFFSET status
mov    [BX],100
mov    [SI],100
```

  - Not clear whether the assembler should use byte or word equivalent of 100

Data Transfer Instructions (cont'd)

Ambiguous moves: PTR directive

- The PTR assembler directive can be used to clarify
- The last two **MOV** instructions can be written as

```
mov    WORD PTR [BX],100
mov    BYTE PTR [SI],100
```

  - WORD and BYTE are called *type specifiers*
- We can also use the following type specifiers:
  - **DWORD** for doubleword values
  - **QWORD** for quadword values
  - **TWORD** for ten byte values

Data Transfer Instructions (cont'd)

The `xchg` instruction

- The syntax is

```
xchg    operand1,operand2
```

  Exchanges the values of **`operand1`** and **`operand2`**

Examples

```
xchg    EAX,EDX
xchg    response,CL
xchg    total,DX
```

- Without the **`xchg`** instruction, we need a temporary register to exchange values using only the **`mov`** instruction

Data Transfer Instructions (cont'd)

The `xchg` instruction

- The **`xchg`** instruction is useful for conversion of 16-bit data between little endian and big endian forms
  - Example:

```
mov     AL,AH
```

    converts the data in AX into the other endian form
- Pentium provides **`bswap`** instruction to do similar conversion on 32-bit data

```
bswap    32-bit register
```

  - **`bswap`** works only on data located in a 32-bit register

Data Transfer Instructions (cont'd)

The `xlat` instruction

- The **`xlat`** instruction translates bytes
- The format is

  `xlatb`
- To use **`xlat`** instruction
  - » BX should be loaded with the starting address of the translation table
  - » AL must contain an index in to the table
    - – Index value starts at zero
  - » The instruction reads the byte at this index in the translation table and stores this value in AL
    - – The index value in AL is lost
  - » Translation table can have at most 256 entries (due to AL)

Data Transfer Instructions (cont'd)

The `xlat` instruction

Example: Encrypting digits

Input digits: **`0 1 2 3 4 5 6 7 8 9`**

Encrypted digits: **`4 6 9 5 0 3 1 8 7 2`**

```
.DATA
xlat_table  DB  '4695031872'
...
.CODE
mov    BX,OFFSET xlat_table
GetCh  AL
sub    AL,'0' ; converts input character to index
xlatb         ; AL = encrypted digit character
PutCh  AL
...
```

Overview of Assembly Instructions

- Pentium provides several types of instructions
- Brief overview of some basic instructions:
  - Arithmetic instructions
  - Jump instructions
  - Loop instruction
  - Logical instructions
  - Shift instructions
  - Rotate instructions
- These sample instructions allows you to write reasonable assembly language programs

Overview of Assembly Instructions (cont'd)

Arithmetic Instructions

INC and DEC instructions

- Format:

```
inc  destination          dec  destination
```

- Semantics:

```
destination := destination +/- 1
```

  - » **`destination`** can be 8-, 16-, or 32-bit operand, in memory or register
    - ✈No immediate operand

- **Examples**

```
inc    BX     ; BX := BX+1
dec    value  ; value := value-1
```

Overview of Assembly Instructions (cont'd)

Arithmetic Instructions

ADD instruction

* Format:

```
add  destination,source
```

* Semantics:

```
destination := (destination)+(source)
```

- **Examples**

```
add    EBX,EAX
add    value,35
```

* **`inc EAX`** is better than **`add EAX,1`**
  - **`inc`** takes less space
  - Both execute at about the same speed

Overview of Assembly Instructions (cont'd)

Arithmetic Instructions

SUB instruction

* Format:

```
sub  destination,source
```

* Semantics:

```
destination := (destination)-(source)
```

- **Examples**

```
sub    EBX,EAX
sub    value,35
```

* **`dec EAX`** is better than **`sub EAX,1`**
  - **`dec`** takes less space
  - Both execute at about the same speed

Overview of Assembly Instructions (cont'd)

Arithmetic Instructions

CMP instruction

- Format:
  `cmp destination,source`
- Semantics:
  `(destination)-(source)`
- **`destination`** and **`source`** are not altered
- Useful to test relationship (>, =) between two operands
- Used in conjunction with conditional jump instructions for decision making purposes

- **Examples**

```
cmp   EBX,EAX                cmp   count,100
```


Overview of Assembly Instructions (cont'd)

Jump Instructions

Unconditional Jump

- Format:
  `jmp label`
- Semantics:
  - » Execution is transferred to the instruction identified by **`label`**

- **Examples:** Infinite loop

```
        mov    EAX,1
inc_again:
        inc    EAX
        jmp    inc_again
        mov    EBX,EAX    ; never executes this
```

Overview of Assembly Instructions (cont'd)

Jump Instructions

Conditional Jump

- Format:
  `j<cond>  label`
- Semantics:
  - Execution is transferred to the instruction identified by **`label`** only if **`<cond>`** is met

- **Examples:** Testing for carriage return

```
        GetCh    AL
        cmp    AL,0DH  ; 0DH = ASCII carriage return
        je     CR_received
        inc    CL
           ...
    CR_received:
```


Overview of Assembly Instructions (cont'd)

Jump Instructions

Conditional Jump

- Some conditional jump instructions
  - Treats operands of the CMP instruction as signed numbers

```
je      jump if equal
jg      jump if greater
jl      jump if less
jge     jump if greater or equal
jle     jump if less or equal
jne     jump if not equal
```

Overview of Assembly Instructions (cont'd)

Jump Instructions

Conditional Jump

* Conditional jump instructions can also test values of the individual flags

```
jz      jump if zero (i.e., if ZF = 1)
jnz     jump if not zero (i.e., if ZF = 0)
jc      jump if carry (i.e., if CF = 1)
jnc     jump if not carry (i.e., if CF = 0)
```

* **jz** is synonymous for **je**
* **jnz** is synonymous for **jne**

Overview of Assembly Instructions (cont'd)

Loop Instruction

LOOP Instruction

* Format:

```
loop  target
```

* Semantics:
  - » Decrements CX and jumps to target if CX ≠ 0
    - – CX should be loaded with a loop count value

- **Example:** Executes loop body 50 times

```
        mov    CX,50
    repeat:
        <loop body>
        loop   repeat
           ...
```

Overview of Assembly Instructions (cont'd)

Loop Instruction

- The previous example is equivalent to

```
        mov    CX,50
    repeat:
        <loop body>
        dec    CX
        jnz    repeat
           ...
```

  - Surprisingly,

```
        dec    CX
        jnz    repeat
```

    executes faster than

```
        loop   repeat
```


Overview of Assembly Instructions (cont'd)

Logical Instructions

- Format:

```
and  destination,source
or   destination,source
not  destination
```

- Semantics:
  - » Performs the standard bitwise logical operations
    - – result goes to **`destination`**
- TEST is a non-destructive AND instruction

```
test  destination,source
```

- Performs logical AND but the result is not stored in **`destination`** (like the CMP instruction)

Overview of Assembly Instructions (cont'd)

Logical Instructions

Example: Testing the value in AL for odd/even number

```
    test  AL,01H  ; test the least significant bit
    jz    even_number
odd_number:
    <process odd number>
    jmp   skip
even_number:
    <process even number>
skip:
     . . .
```


Overview of Assembly Instructions (cont'd)

Shift Instructions

* Format:

  Shift left

```
shl  destination,count
shl  destination,CL
```

  Shift right

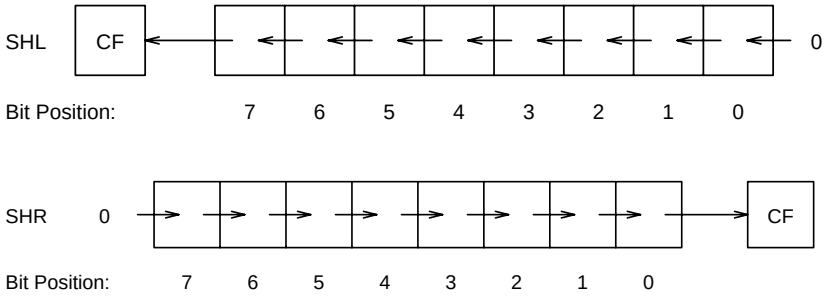
```
shr  destination,count
shr  destination,CL
```

* Semantics:
  - » Performs left/right shift of **`destination`** by the value in **`count`** or **`CL`** register
    - – **`CL`** register contents are not altered

Overview of Assembly Instructions (cont'd)

Shift Instructions

- Bit shifted out goes into the carry flag
  - » Zero bit is shifted in at the other end



Overview of Assembly Instructions (cont'd)

Shift Instructions

- **`count`** is an immediate value

```
shl     AX,5
```

- Specification of **`count`** greater than 31 is not allowed
  - » If a greater value is specified, only the least significant 5 bits are used
- CL version is useful if shift count is known at run time
  - » E.g. when the shift count value is passed as a parameter in a procedure call
  - » Only the CL register can be used
    - ✈Shift count value should be loaded into CL

```
mov     CL,5
shl     AX,CL
```

Overview of Assembly Instructions (cont'd)

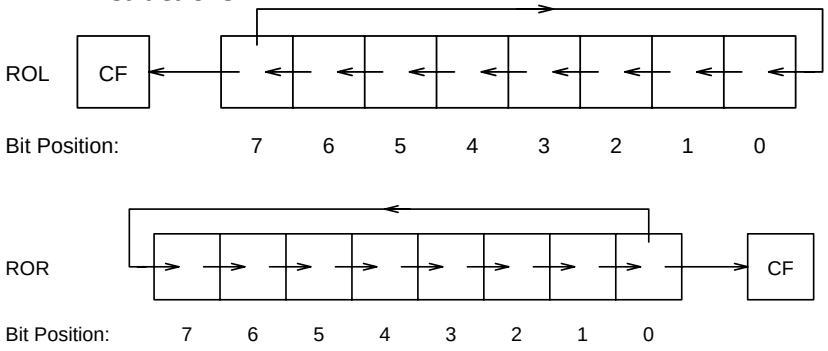
Rotate Instructions

* Two types of ROTATE instructions
* Rotate without carry
  - » **`rol`** (ROtate Left)
  - » **`ror`** (ROtate Right)
* Rotate with carry
  - » **`rcl`** (Rotate through Carry Left)
  - » **`rcr`** (Rotate through Carry Right)
* Format of ROTATE instructions is similar to the SHIFT instructions
  - » Supports two versions
    - – Immediate count value
    - – Count value in CL register

Overview of Assembly Instructions (cont'd)

Rotate Instructions

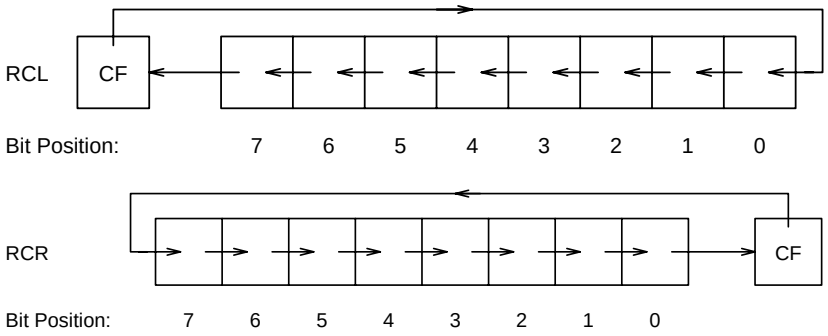
* Bit shifted out goes into the carry flag as in SHIFT instructions



Overview of Assembly Instructions (cont'd)

Rotate Instructions

* Bit shifted out goes into the carry flag as in SHIFT instructions



Defining Constants

- Assembler provides two directives:
  - » EQU directive
    - – No reassignment
    - – String constants can be defined
  - » = directive
    - – Can be reassigned
    - – No string constants
- Defining constants has two advantages:
  - * Improves program readability
  - * Helps in software maintenance
    - » Multiple occurrences can be changed from a single place
- Convention
  - » We use all upper-case letters for names of constants

Defining Constants

The EQU directive

- Syntax:

```
name   EQU  expression
```

  - Assigns the result of **`expression`** to **`name`**
  - The **`expression`** is evaluated *at assembly time*
    - → Similar to **`#define`** in C

Examples

```
NUM_OF_ROWS   EQU   50
NUM_OF_COLS   EQU   10
ARRAY_SIZE    EQU   NUM_OF_ROWS * NUM_OF_COLS
```

  - Can also be used to define string constants

```
JUMP   EQU   jmp
```


Defining Constants

The = directive

- Syntax:

```
name = expression
```

  - Similar to EQU directive
  - Two key differences:
    - » Redefinition is allowed

```
count = 0
. . .
count = 99
```

      is valid

    - » Cannot be used to define *string constants* or to redefine *keywords* or *instruction mnemonics*

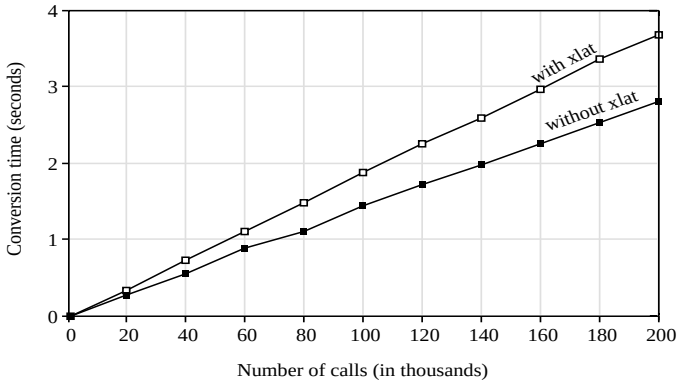
      Example: **`JUMP = jmp`** is *not* allowed

Illustrative Examples

- Five examples are presented:
  - Conversion of ASCII to binary representation (BINCHAR.ASM)
  - Conversion of ASCII to hexadecimal by character manipulation (HEX1CHAR.ASM)
  - Conversion of ASCII to hexadecimal using the XLAT instruction (HEX2CHAR.ASM)
  - Conversion of lowercase letters to uppercase by character manipulation (TOUPPER.ASM)
  - Sum of individual digits of a number (ADDIGITS.ASM)

Performance: When to Use XLAT

- Lowercase to uppercase conversion
  - XLAT is bad for this application



Performance: When to Use XLAT (cont'd)

- Hex conversion
  - XLAT is better for this application