Introduction to Assembly Language

The assembly language templates in these notes will enable you to convert C-like pseudo-code into assembly language. After this section you should be able to:

- *explain why you should (almost) never program in assembler*
- *be able to convert the following pseudo-code into 80x86 assembly language:*
  - *variable declarations*
  - *array subscripts*
  - *assignment statements*
  - *if/else statements*
  - *while loops*

Why *not* to Write Assembly Code

You should almost never write in assembly language because:

- assembler is harder and slower to read and write
- assembler is more difficult to document
- The greatest optimization gains result from changes to algorithms and data structures, not from optimizing the implementation of a given algorithm. Therefore large assembly language programs are usually less efficient and harder to optimize because they are difficult write and to modify.
- it takes most programmers several months of learning about a processor and an instruction set before they can write assembly code that is as efficient as that generated by a good optimizing compiler.
- assembler is not portable.

The only things that should be written in assembler are:

- code that is not possible to write in a high-level language such as the code that does a context save/restore (or context switch) before/after an interrupt
- short functions (typically a few lines) that make use of processor-specific instructions which are not yet supported by your compiler. Some examples are a filter routine written using DSP-processor instructions or an image transform using MMX instructions

How to Write Assembly Language

1. Start by writing simple C or C-like pseudo-code that solves the given problem. Use only the simple statements described below.
2. Use the "templates" described below to convert each C statement into the equivalent assembly-language instructions.

Guidelines

Declare storage (using `DW` or `DB`) for each variable required by your code.

Use the `AX` register for computing 16-bit results and AL for computing 8-bit results. Do not use other registers for computations.

Use `BX` only for computing addresses (indexed addressing), and `DX` only for `IN` and `OUT` instructions.

Save the result of each expression at the end of each statement – do not try to save a value across two statements by leaving it in AX or AL.

All labels should be unique. Functions should have meaningful labels but targets of short branches

within a function may assigned non-meaningful labels. For example, the labels in the templates below are of the form `l`*n*.

Notation

The notation `op` below refers to an arithmetic or logical operator. For example, use `ADD` for `+`, `SUB` for `-`, `AND` for `&`, `OR` for `|`, etc.)

The notation `cop` below is used for a comparison operator while `cop*` is its complement. The following table shows the C comparison operators and their complements along with the conditional branch instruction to be used. The signed version is used when the variables being compared are in two's-complement (C `signed` variables).

C op	C op*	assembly signed op	assembly unsigned op
>	<=	JG	JA
<	>=	JL	JB
<=	>	JLE	JBE
>=	<	JGE	JAE
==	!=	JE	JE
!=	==	JNE	JNE

The notation `s1...` stands for other statements (of any type).

Variable Declarations

After the last instruction in your program, declare each variable using `DB` (for `char` or 1-byte variables) or `DW` (for `int` or 16-bit variables). For example, the C declarations:

```
int x ;
char y, z[100] ;
```

can written as follows in assembler:

```
x       dw      1 dup (?)
y       db      1 dup (?)
z       db      100 dup (?)
```

Assignment Statement Template

A simple assignment statement of the form:

```
z = x op y ;
```

can written as follows in assembler:

```
mov     ax,x
op      ax,y
mov     z,ax
```

For example example, the C expression:

```
c = a - b ;
```

involving byte variables is written as follows in assembler:

```
mov     al,a
sub     al,b
mov     c,al
```

and the C expression:

```
e = d & 0x1000 ;
```

is written as follows:

```
mov     ax,d
and     ax,01000H
mov     e,ax
```

Array Subscripting Template

To obtain a value involving an array subscript the BX register is first loaded with the address of the desired location. For example, the C code:

```
y = x[i] ;
```

is written as follows in assembler:

```
mov     bx,offset x
add     bx,i
mov     ax,[bx]
mov     y,ax
```

Note that if `x` is an array of word, the value of `i` has to be multiplied by two before it is added to `bx` (`ADD i` twice).

if/else Statement Template

The C code:

```
if ( a cop b ) {
        s1...
} else {
        s2...
}
```

is written as follows in assembler:

```
        mov     ax,a
        cmp     ax,b
        cop*    l1

        s1...

        jmp     l2
l1:
        s2...

l2:
```

For example, the C code:

```
if ( a > b ) {
        c = 0 ;
} else {
        c = 1 ;
}
```

is written as follows in assembler:

```
        mov     al,a
        cmp     al,b
        jle     l3

        mov     ax,0
        mov     c,ax

        jmp     l5

l3:

        mov     ax,1
        mov     c,ax

l5:
```

while Statement Template

A C `while` loop:

```
while ( a cop b ) {
        s1...
}
```

is written as follows:

```
l1:     mov     ax,a
        cmp     ax,b
        cop*    l2

        s1...

        jmp     l1

l2:
```

For example, the C `for` loop, `for( i=0 ; i<n ; i++ )` can be re-written as follows in C:

```
i = 0 ;
while ( i < n ) {

        s1...

        i = i + 1 ;
}
```

which is written as follows in assembler:

```
        mov     ax,0
        mov     i,ax

l1:     mov     ax,i
        cmp     ax,n
        jge     l2

        s1...

        mov     ax,i
        add     ax,1
        mov     i,ax

        jmp     l1

l2:
```