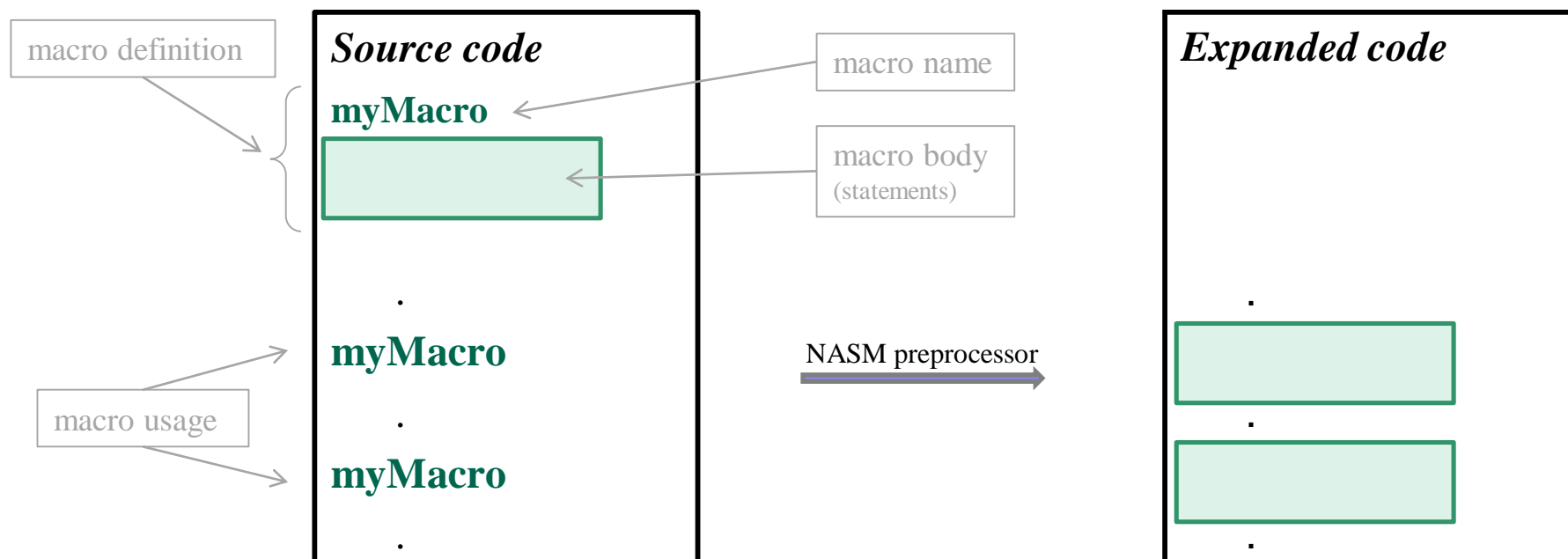


Computer Architecture and Assembly Language

Practical Session 6

NASM Preprocessor - Macro - definition

- Macro is a set of statements given a **symbolic name**
- Macro is **invoked, not called**. A copy of macro is **inserted directly** into the program
- After being defined, NASM preprocessor will **substitute (expand)** those statements whenever it finds the symbolic name



Note: we cover only part of NASM macro processor features. Read more [here](#)

Single-line macros

- **%define** (**%ifndef** for case insensitive) - a macro resolved at the time that it is invoked

Example:

```
%define ctrl 0x1F &  
%define param(a, b) ((a)+(a)*(b))  
mov byte [param(2,ebx)], ctrl 'D'  
expanded ↓ by NASM preprocessor  
mov byte [((2)+(2)*(ebx))], 0x1F & 'D'
```

- **%xdefine** - a macro resolved at the time that it is defined

Example:

```
%define isTrue 1  
%define isFalse isTrue  
%define isTrue 0  
val1: db isFalse ; val1 = ?  
%define isTrue 2  
val2: db isFalse ; val2 = ?
```

```
%xdefine isTrue 1  
%xdefine isFalse isTrue  
%xdefine isTrue 0  
val1: db isFalse ; val1=?  
%xdefine isTrue 2  
val2: db isFalse; val2=?
```

Single-line macros

- **%define** (**%ifndef** for case insensitive) - a macro resolved at the time that it is invoked

Example:

```
%define ctrl 0x1F &  
%define param(a, b) ((a)+(a)*(b))  
mov byte [param(2,ebx)], ctrl 'D'
```

expanded ↓ by NASM preprocessor

```
mov byte [((2)+(2)*(ebx))], 0x1F & 'D'
```

- **%xdefine** - a macro resolved at the time that it is defined

Example:

```
%define isTrue 1  
%define isFalse isTrue  
%define isTrue 0  
val1: db isFalse ; val1 = 0  
%define isTrue 2  
val2: db isFalse ; val2 = 2
```

use the current
value of 'isTrue'

```
%xdefine isTrue 1  
%xdefine isFalse isTrue  
%xdefine isTrue 0  
val1: db isFalse ; val1=1  
%xdefine isTrue 2  
val2: db isFalse; val2=1
```

use 'isTrue' value to
at the time that
'isFalse' was defined

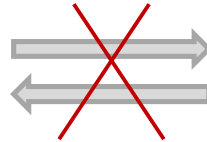
- **%undef** – undefines defined single-line macro

Single-line macros

- We can overload single-line macros. The preprocessor will be able to handle both types of macro call, by counting the parameters you pass.

```
%define foo (x) 1+x
```

```
%define foo (x, y) 1+x*y
```



```
%define foo 1+ebx
```

A macro *with no parameters* prohibits the definition of the same name as a macro *with parameters*, and vice versa.

Note: there is a mechanism which detects when a macro call has occurred as a result of a previous expansion of the same macro, to guard against circular references and infinite loops.

Multiple-line macros

- **%macro** (**%imacro** for case insensitive) <name, numOfParams> ... **%endmacro**
- macro parameters is referred to as %1, %2, %3, ...

Example:

```
%macro startFunc 1 ← gets single parameter
    push ebp
    mov ebp, esp
    sub esp, %1 ← first macro parameter
%endmacro
```

```
my_func:
    startFunc 12
    ...
```

NASM preprocessor



```
my_func:
    push ebp
    mov ebp, esp
    sub esp,12
```

%unmacro – undefines defined single-line macro

For example: **%unmacro startFunc 1**

Multiple-line macros

- If we need to pass **a comma as part of a parameter** to a multi-line macro, we can do that by enclosing the entire parameter in braces.

```
%macro DefineByte 2
```

```
    %2: db %1
```

```
%endmacro
```

```
DefineByte {"Hello",10,0}, msg  msg: db "Hello",10,0
```

- Multi-line macros can be **overloaded** by defining the same macro name several times with different amounts of parameters. (Also macros with no parameters.)

```
%macro push 2
```

```
    push %1
```

```
    push %2
```

```
%endmacro
```

```
push ebx ; this line is original push instruction
```

```
push eax, ecx ; but this one is a macro invocation
```

Note: if define macro 'push' with one parameter, the original 'push' instruction would be overloaded.

Multiple-line macros with internal labels

Example:

```
%macro retz 0          ; return (from function) if ZF == 0, else continue
    jnz %%skip
    ret
    %%skip:
%endmacro
```

In every ‘retz’ invocation, the preprocessor creates some unique label of the form: **..@2345.skip** to substitute for the label %%skip, where **the number 2345 changes with every macro invocation.**

If a label begins with the special prefix **..@**, then it doesn’t interfere with the local label mechanism.

Example:

```
label1:                ; a non-local label
    ..@ 2345.skip :    ; this is a macro label
    .local:           ; this is label1.local
```


Macro with default parameters

We supply a minimum and maximum number of parameters for a macro of this type; the minimum number of parameters are required in the macro call, and we provide defaults for the optional ones.

```
%macro name min - max <default parameters list> ... %endmacro
```

Example:

```
%macro foo 1-3 eax, [ebx+2]
```

```
%macro foo 1-2  
  mov eax, %1  
  mov ebx, %2  
%endmacro  
  
foo 2
```

```
mov eax, 2  
mov ebx,
```

- could be called with between one (min) and three (max) parameters
- %1 would always be taken from the macro call (minimal number of parameters)
- %2, if not specified by the macro call, would default to `eax`
- %3, if not specified by the macro call, would default to `[ebx+2]`

Note: we may omit parameter defaults from the macro definition, in which case the parameter default is taken to be **blank**. This can be useful for macros which can take a variable number of parameters, since the %0 token allows us to determine how many parameters were really passed to the macro call.

Macro with greedy parameters

If invoke the macro with more parameters than it expects, all the spare parameters get lumped into the last defined one.

```
%macro macro Name numOfParams + ... %endmacro
```

The mark %numOfParams will be replaced with numOfParams's parameter and whatever follows it.

Example:

```
%macro writefile 2+
```

```
    mov    ecx, ?
```

```
    mov    edx, ?
```

```
    mov    ebx, %1
```

```
    mov    eax, 4
```

```
    int    0x80
```

```
%endmacro
```

```
writefile [fileHandle], "String to print",10,0
```

Macro with greedy parameters

If invoke the macro with more parameters than it expects, all the spare parameters get lumped into the last defined one.

```
%macro macro Name numOfParams + ... %endmacro
```

The mark `%numOfParams` will be replaced with `numOfParams`'s parameter and whatever follows it.

Example:

```
%macro writefile 2+
```

```
%%str:    db    %2  
    mov    ecx, %%str  
    mov    edx, ?  
    mov    ebx, %1  
    mov    eax, 4  
    int    0x80  
%endmacro
```

```
writefile [fileHandle], "String to print", 10, 0
```



Macro with greedy parameters

If invoke the macro with more parameters than it expects, all the spare parameters get lumped into the last defined one.

```
%macro macro Name numOfParams + ... %endmacro
```

The mark %numOfParams will be replaced with numOfParams's parameter and whatever follows it.

Example:

```
%macro writefile 2+
```

```
%%str: db %2
```

```
%%endstr:
```

```
mov ecx, %%str
```

```
mov edx, %%endstr - %%str
```

```
mov ebx, %1
```

```
mov eax, 4
```

```
int 0x80
```

```
%endmacro
```

```
writefile [fileHandle], "String to print", 10, 0
```



Macro with greedy parameters

If invoke the macro with more parameters than it expects, all the spare parameters get lumped into the last defined one.

```
%macro macro Name numOfParams + ... %endmacro
```

The mark %numOfParams will be replaced with numOfParams's parameter and whatever follows it.

Example:

```
%macro writefile 2+  
    jmp    %%endstr          ; otherwise get runtime error  
%%str:    db    %2  
%%endstr:  
    mov    ecx, %%str  
    mov    edx, %%endstr - %%str  
    mov    ebx, %1  
    mov    eax, 4  
    int    0x80  
%endmacro
```

```
writefile [fileHandle], "String to print", 10, 0
```

Macro with greedy parameters

Example:

```
%macro writefile 2+
```

```
    jmp    %%endstr
```

```
    %%str: db    %2
```

```
%%endstr:
```

```
    mov    ecx, %%str
```

```
    mov    edx, %%endstr - %%str
```

```
    mov    ebx, %1
```

```
    mov    eax, 4
```

```
    int    0x80
```

```
%endmacro
```

writefile 1, "12345"

```
> nasm -f elf asm.s -l asm.l
```

```
1
2          section .text
3
4          %macro writefile 2+
5              jmp    %%endstr
6              %%str: db    %2
7              %%endstr:
8              mov    ecx, %%str
9              mov    edx, %%endstr - %%str
10             mov    ebx, %1
11             mov    eax, 4
12             int    0x80
13         %endmacro
14
15         writefile 1, "12345"
16         <1> jmp %%endstr
17         <1> %%str: db %2
18         <1> %%endstr:
19         <1> mov ecx, %%str
20         <1> mov edx, %%endstr - %%str
21         <1> mov ebx, %1
22         <1> mov eax, 4
23         <1> int 0x80
24
```

After compilation, the string "12345" becomes a part of section .text. In order to tell CPU not execute it (since this is not code !) we should skip it by jumping.

Advanced example of macro usage

```
%macro multipush 1-*  
    %rep %0  
        push %1  
        %rotate 1  
    %endrep  
%endmacro
```

This macro invokes the PUSH instruction on each of its arguments in turn, from left to right. It begins by pushing its first argument, %1, then invokes %rotate to move all the arguments one place to the left, so that the original second argument is now available as %1. Repeating this procedure as many times as there were arguments (achieved by supplying %0 as the argument to %rep) causes each argument in turn to be pushed.

Note also the use of * as the maximum parameter count, indicating that there is no upper limit on the number of parameters you may supply to the multipush macro.

```
%macro multipop 1-*  
    %rep %0  
        %rotate -1  
        pop    %1  
    %endrep  
%endmacro
```

It would be convenient, when using this macro, to have a POP equivalent, which didn't require the arguments to be given in reverse order. Ideally, you would write the multipush macro call, then cut-and-paste the line to where the pop needed to be done, and change the name of the called macro to multipop, and the macro would take care of popping the registers in the opposite order from the one in which they were pushed.

This macro begins by rotating its arguments one place to the right, so that the original last argument appears as %1. This is then popped, and the arguments are rotated right again, so the second-to-last argument becomes %1. Thus the arguments are iterated through in reverse order.

Macro Expansion

Use `-e` option to get a source code with all your macros expanded.

> `nasm -e sample.s`

`%line 12+0` means: expansion line of line # 12 in the original file, incremented by 0 additional lines

```
1  %macro retz 0
2      jnz %%skip
3      ret
4  %%skip:
5      %endmacro
6  %define m(x) x * 1
7  section .text
8      global _start
9
10 _start:
11
12  retz
13  retz
14  retz
15  mov ebx,m(5)
16  mov eax,1
17  int 0x80
```

```
%line 7+1 sample.s
[section .text]
[global _start]

_start:

jnz ..@2.skip
%line 12+0 sample.s
ret
..@2.skip:
%line 13+1 sample.s
jnz ..@3.skip
%line 13+0 sample.s
ret
..@3.skip:
%line 14+1 sample.s
jnz ..@4.skip
%line 14+0 sample.s
ret
..@4.skip:
%line 15+1 sample.s
mov ebx,5 * 1
mov eax,1
int 0x80
```

replaces text by text !