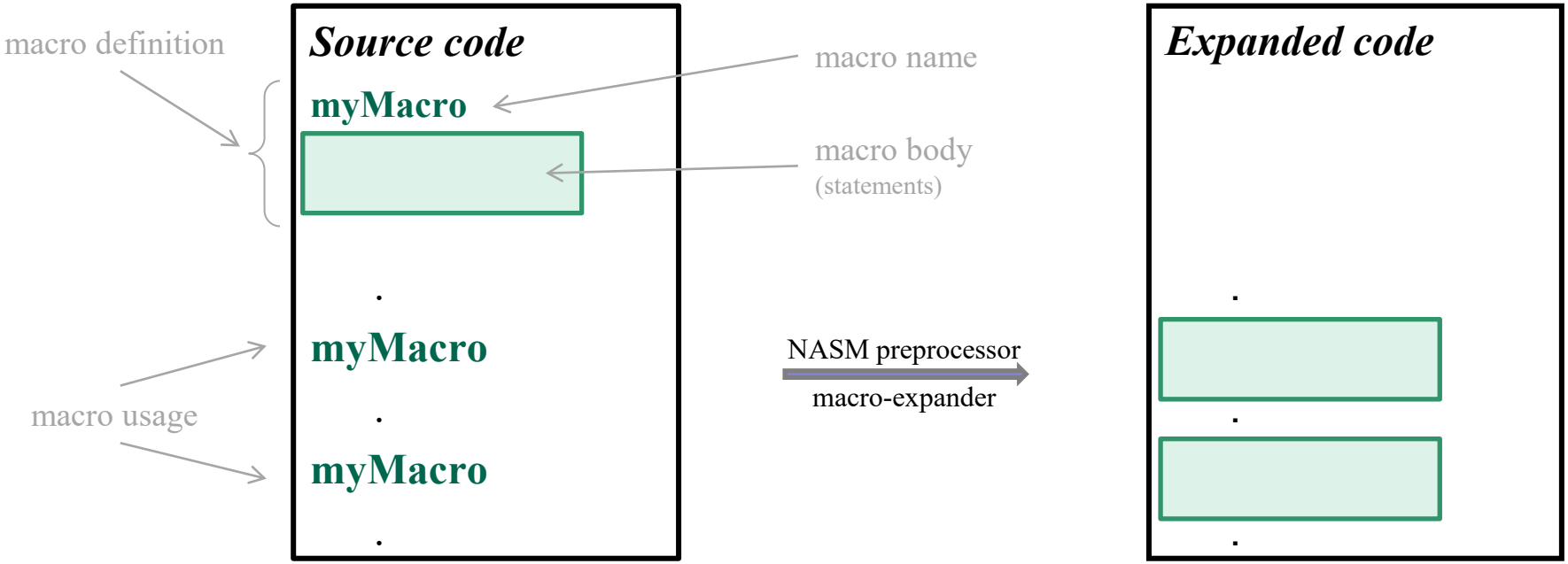
Computer Architecture and System Programming Laboratory

TA Session 6

Macro

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** (**%idefine** 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** (**%idefine** 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:

use the current value of 'isTrue'

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

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

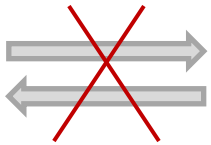
```
%xdefine isTrue 1
%xdefine isFalse isTrue
%xdefine isTrue 0
val1: db isFalse ; val1=1
%xdefine isTrue 2
val2: db isFalse; val2=1
```

- **%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.

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 multiple-line macro

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
```

NASM preprocessor →

```
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
```

this is overload of push instruction

```
push ebx        ; this is original push instruction
push eax, ecx   ; this is a macro invocation
```

NASM preprocessor →

```
push ebx
push eax
push ecx
```

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
```

Note:

DefineByte {"Hello",10,0}, msg → NASM preprocessor → 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 1
        push %1
%endmacro
```

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

push eax → NASM preprocessor → Error: infinite loop !

*There is a macro-expander 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 with internal labels

```
%macro addEAX10 0          ; if ZF == 0, add 10 to EAX
        jnz skip
        add eax, 10
    skip:
%endmacro
```

```
…
addEAX10

…
addEAX10
…
addEAX10
…
```

```
        …
        jnz skip
        add eax, 10
skip:

        …
        jnz skip
        add eax, 10
skip:
        …
        jnz skip
        add eax, 10
skip:
        …
```

same label cannot be defined several times in the code

Multiple-line macros with internal labels

```
%macro addEAX10 0          ; if ZF == 0, add 10 to EAX
            jnz %%skip
            add eax, 10
   %%skip:
%endmacro
```

Use –e option to get a source code with all your macros expanded.
*> **nasm -e sample.s***

```
…
addEAX10

…
addEAX10
…
addEAX10
…
```

```
            …
            jnz ..@1.skip
            add eax, 10
..@1.skip :

            …
            jnz ..@2.skip
            add eax, 10
..@2.skip :
            …
            jnz ..@3.skip
            add eax, 10
..@3.skip :
            …
```

*For every ‘addEAX10’ invocation, macro-expander creates a **unique label to substitute for %%skip**, where the number part of the label changes with every macro invocation.*

macro-label is not local, but also is not global label. Ignore it when detect local labels scope

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.

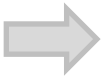
Example:

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

foo 42
```

```
%macro foo 1-3
    mov eax, %1
    mov ebx, %2
%endmacro

foo 42
```



```
mov eax, 42
mov ebx,
```

we will get compile-time error

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

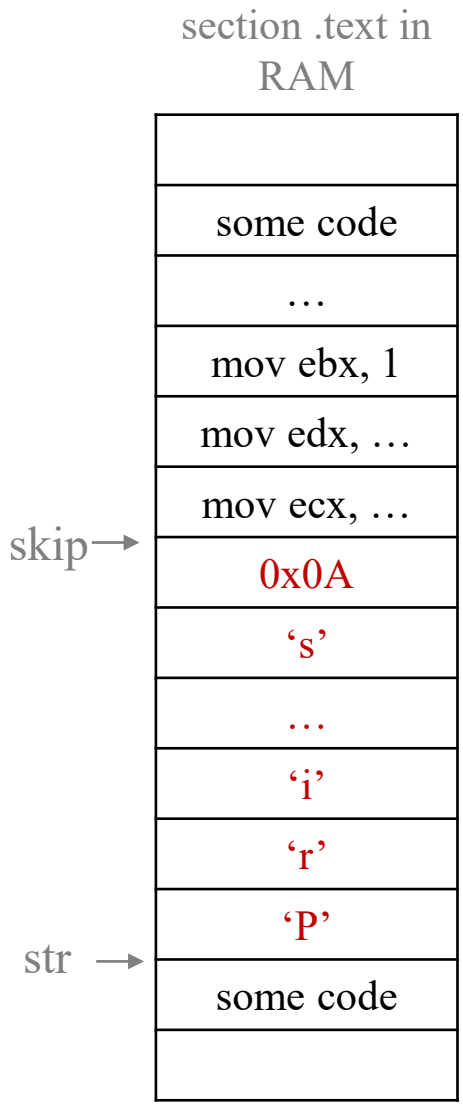
Example:
let's implement C standard library function int puts(const char * str)

myPuts "Print this", "and this", 10

```
%macro  myPuts 1+
        mov ecx, ?
        mov edx, ?
        mov ebx, 1 ; stdout
        mov eax, 4
        int 0x80
%endmacro
```

1+ means that if this macro gets more than one parameter, all the rest parameters would be attached to the first parameter

```
%macro myPuts 1+
        jmp    %% skip
    %%str: db %1
    %%skip:
        mov ecx, %%str
        mov edx, %%skip - %%str
        mov ebx, 1
        mov eax, 4
        int 0x80
%endmacro
```



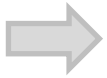
Macro with greedy parameters

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

Example:
let's implement C standard library function int puts(const char * str)

myPuts "Print this", "and this", 10

```
%macro  myPuts 1+
        mov ecx, ?
        mov edx, ?
        mov ebx, 1 ; stdout
        mov eax, 4
        int 0x80
%endmacro
```



```
%macro myPuts 1+
  section .data
    %%str: db %1
    %%len: equ $- %%str
  section .text
        mov ecx, %%str
        mov edx, %%len
        mov ebx, 1
        mov eax, 4
        int 0x80
%endmacro
```

*Note: same **section may be reopened in your code as many times as you wish**. Assembler would gather all the pieces of the same section into a single section.*

Self reading

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.

Read more here