

# Assembly Language Sample Code

Ole Herman Schumacher Elgesem

June 11, 2015

## Contents

|          |  |          |
|----------|--|----------|
| <b>1</b> | <b>Template</b>                                  | <b>2</b> |
| <b>2</b> | <b>Basics</b>                                    | <b>3</b> |
| <b>3</b> | <b>Conversion to ASCII</b>                       | <b>6</b> |
| <b>4</b> | <b>Calling a C Function</b>                      | <b>7</b> |
| 4.1      | The function defined in C . . . . .              | 7        |
| 4.2      | Assembly code calling the function . . . . .     | 7        |
| <b>5</b> | <b>Copying a section of memory</b>               | <b>8</b> |
| <b>6</b> | <b>Using "Variables"</b>                         | <b>9</b> |
| <b>7</b> | <b>Index of Character and Contains Character</b> | <b>9</b> |
| 7.1      | Test program in C . . . . .                      | 9        |
| 7.2      | Function in Assembly . . . . .                   | 10       |

# 1 Template

```
# Simple template for assembly code ( AT&T syntax )
# Author: github.com/olehermanse

# Simple beginner function in asm, equivalent to:
# int asmtemplate(int input)
#     return input + 9000;

# asmtemplate() should be globally available:
.globl asmtemplate
.globl _asmtemplate          # Mac OSX Compatibility

_asmtemplate:               # Mac OSX Compatibility
asmtemplate:
    push %ebp               # save base pointer for safe return
    movl %esp, %ebp         # store constant stack pointer ref

    # Get argument from stack:
    # Stack pointer is at (%ebp), nothing is there (yet)
    # Old %ebp is on stack, at 4(%ebp)
    # Argument is at 8(%ebp)
    movl 8(%ebp), %eax      # ( movl src, dest )

    # Add 9000 to the value
    addl $9000, %eax        # $ sign for constants like: $10 , $('c'), $0xFF, etc.

    # Restore and return:
    pop %ebp
    ret
```

## 2 Basics

```
# The very basics of assembly programming (AT&T syntax)
# Author: github.com/olehermanse

# ===INTRODUCTION:===
# This is meant as a quick reference for beginners (with a background in C).

# COMMENTS
# Hash symbol # is used for comments

# INSTRUCTIONS
# In assembly language we use instructions to program the cpu.
# Instructions take 0 to 2 operands (arguments)
# and often have a byte, word and long version (1 byte, 2 bytes, 4 bytes)
# Example: mov(bwl) src, dest -> movl %eax, %ebx
# The long version of mov is movl
# movl takes two long (4 byte) values as arguments
# mov copies the value from src to dest
# In this example we copied the value of the %eax register to the %ebx register

# CONSTANTS
# Constants are written with a dollar sign $:
# $0, '$c', $0xFF, $0b10110100...

# REGISTER VALUES
# Registers are prefixed with a percentage sign %:
# %eax, %ebx, %ecx, %edx...

# MEMORY ADDRESSES
# Sometimes registers contain addresses(pointers)
# and we want to edit the data at the address, not the address itself:
# Parentheses () are used to get value at an address:
# (%eax), (%ebx), (%ecx), (%edx) ...
# We can also add an offset to the addresses:
# 1(%eax), 4(%ebp), 8(%esp),

# REGISTERS
# Every register has a purpose,
# you should try to follow these standards when possible:
# EAX: Accumulator ( and return value )
# EBX: Base Index ( when using arrays )
# ECX: Counter
# EDX: Misc / Extend Precision of EAX
# ESI: Source index ( Copying data )
# EDI: Destination index ( Copying data )
# ESP: Stack pointer ( Used by push and pop )
# EBP: Stack Base Frame pointer

# Caller / Callee convention:
# Callee saved, these are registers you should save before using:
# EBX EDI ESI EBP (ESP)      # Leave the stack pointer alone!
```

```

# Caller saved, you dont need to save these:
# EAX ECX EDX

# Splitting registers into words and bytes:
# You can use byte or word operations on only part of a 32 bit register:
# %ax is the 16 LSB of %eax, %al is the 8 LSB of %ax, %ah is the 8 MSB of %ax
# %bx is the 16 LSB of %ebx, %bl is the 8 LSB of %bx, %bh is the 8 MSB of %bx
# %cx is the 16 LSB of %ecx, %cl is the 8 LSB of %cx, %ch is the 8 MSB of %cx
# %dx is the 16 LSB of %edx, %dl is the 8 LSB of %dx, %dh is the 8 MSB of %dx

# Something like this:
# EAX: | 03 || 02 || 01 || 00 | We can use long operations on eax
# EAX: |   ||   ||   ax   | We can use word operations on ax
# EAX: |   ||   || ah || al | We can use byte operations on ah and al

# ===END OF INTRODUCTION===

# EXAMPLE CODE

# This will declare asmBasics() as a global function:
# Which means we can call it from our main function in asmBasics.c
.globl asmBasics
.globl _asmBasics          # This is for Mac OS Compatibility

# Our "function" starts here:
_asmBasics:                # This is for Mac OS Compatibility
asmBasics:
    pushl %ebp              # Save Base Frame pointer for later
                           # Its old value is now at 4(%esp)
    movl %esp, %ebp         # Set %ebp to point to where the stack is now

    # Save the registers we are responsible for saving before using: (See REGISTERS)
    pushl %ebx
    pushl %edi
    pushl %esi

    # We can now use almost all the registers freely.

    # MOVING / COPYING DATA:
    # the mov instruction can copy data from one place to another:
    # mov(bwl) src, dest
    movl $0, %eax           # eax = 0
    movl %eax, %ebx         # ebx = eax
    movl %ebx, %ecx         # ecx = ebx
    movl %ecx, %edx         # edx = ecx

    # xor is also commonly used to clear a register:
    xorl %eax, %eax         # eax = 0

    # INCREMENT: (+1)
    incl %eax               # ++eax

```

```

# DECREMENT: (-1)
decl %eax                # --eax

# ADDITION: (+)
addb $7,    %al          # al += 7    // al is 8LSB of ax and eax
addb $18,   %ah          # ah += 18   // ah is 8MSB of ax and bits 8-15 of eax

# SUBTRACTION: (-)
subb $9,    %ah          # ah -= 9
# 18 - 9 = 9

# MULTIPLICATION: (*)
imulb %ah                # ax = al * ah
# 7 * 9 = 63

# DIVISION: (/)
movb $3,    %cl          # cl = 3
idivb %cl              # al = al/cl
andl $0x000000FF, %eax   # eax = eax    & 255 // Bitmask using AND, remove MSB
# 63 / 3 = 21

# COMPARE AND COND. JMP: (IF)
cmpl $22, %eax           # if(eax == 21)
je if_body              #    goto if_body

error:                  # else          // Unexpected value - return 1
    movl $1, %eax        #    returnvalue = 1
    jmp return

if_body:
    xorl %eax, %eax      # eax = 0          // We got the expected answer so we will ret 0

return:
    # Restore callee-saved registers:
    popl %esi
    popl %edi
    popl %ebx

    # We can set the stack pointer to what it was,
    # this isnt neccessary as long as we have pushed and popped correctly:
    movl %ebp, %esp

    # Always Restore base frame pointer:
    popl %ebp

    # Return, the return value (int) is in %eax:
    ret

```

### 3 Conversion to ASCII

# Simple demonstration of how to find ascii value of a number from 0 to 9 in assembly  
# Author: [github.com/olehermanse](https://github.com/olehermanse)

```
.globl asciiConvert
.globl _asciiConvert      # Mac OSX Compatibility

# Finds the ascii value of a number form 0 to 9
# All other inputs return -1

_asciiConvert:            # Mac OSX Compatibility
asciiConvert:
    push %ebp              # save base pointer for return
    movl %esp, %ebp        # store constant stack pointer ref

    # movl src, dest
    movl $'0', %eax        # Start with ascii value of '0' in eax register (return)

    cmpl $0, 8(%ebp)        # Compare the input argument to 0
    jl returnEarly         # Return -1 if input is less than 0

    cmpl $9, 8(%ebp)        # Compare the input argument to 9
    jg returnEarly         # Return -1 if input is over 9

    # Add the input value to the ascii value of '0' (in %eax).
    # ( 0 + '0' = 48, ... , 9 + '0' = '9' = 57 )
    addl 8(%ebp), %eax

    # restore ebp and return
    pop %ebp
    ret

# Return -1 on invalid inputs
returnEarly:
    movl $-1, %eax          # Set return value -1
    popl %ebp               # Restore %ebp register
    ret                     # Return
```

## 4 Calling a C Function

### 4.1 The function defined in C

```
// This function is called from assembly code in callCFunc.s:
int cFunc(int input){
    return input * 2;          // Must do something, to ensure function call worked
}

// For Cross platform compatibility Linux <-> Mac OSX :
int _cFunc(int input){
    return cFunc(input);
}
```

### 4.2 Assembly code calling the function

# Simple demonstration of how to call functions from assembly code  
# Author: [github.com/olehermanse](https://github.com/olehermanse)

```
.globl callCFunc
.globl _callCFunc      # For OSX Compatibility

_callCFunc:            # For OSX Compatibility
callCFunc:
    push %ebp          # save base pointer for safe return
    movl %esp, %ebp    # store constant stack pointer ref

    # Get first argument from stack:
    movl 8(%ebp), %eax

    # Push argument on stack for c function call:
    pushl %eax

    # Call the c function from cFunc.c with argument on stack:
    call _cFunc

    # Remove argument from stack after function call:
    # We want to remove the argument from the stack (pop without saving the value)
    addl $4, %esp      # We just move the stack pointer instead of using popl

    # Restore base pointer and return:
    popl %ebp

    # Return with return value in eax:
    ret
```

## 5 Copying a section of memory

```
# Simple example of a memcpy like function in asm (AT&T)
# Author: github.com/olehermanse

# declaration:
# extern int memCopy(char* dest, char* src, int n);

# memCopy should be globally available:
.globl memCopy
.globl _memCopy          # Mac OSX Compatibility

_memCopy:                # Mac OSX Compatibility
memCopy:
    # Standard:
    push %ebp             # save base pointer for safe return
    movl %esp, %ebp       # store constant stack pointer ref

    # Since the function that called memCopy might be using %edi and %esi
    # we should save their values before overwriting them ( and restore them before ret )
    # (Callee-save convention)
    pushl %esi            # save source index register (callee-saved)
    pushl %edi            # save destination index register (callee-saved)

    # Get arguments:
    movl 16(%ebp), %ecx    # n, a count of how many bytes
    movl 12(%ebp), %esi    # src, a pointer for where to get data from
    movl 8(%ebp), %edi     # dest, a pointer for where to put data

    # Since there is no jump here, execution will continue down, to iteration:

# Copy 1 byte, update pointers and counter, repeat until done:
iteration:
    # Check if done:
    cmpl $0, %ecx
    jz return             # if counter == 0 Return
    decl %ecx             # else --counter

    # Copy data:
    # Need to go via %eax, mov doesnt take 2 memory addresses
    movb (%esi), %al      # Get data from source (%al is the 8 LSB of %eax)
    movb %al, (%edi)      # Write data to dest

    # Update pointers:
    incl %esi             # Increment src pointer (address)
    incl %edi             # Increment dest pointer (address)
    jmp iteration         # repeat this routine

# restore and return
return:
    popl %edi             # Restore callee-saved value
    popl %esi             # Restore callee-saved value
```



```

pop %ebp          # Standard, restore base pointer
xorl %eax, %eax   # Return value, 0
ret

```

## 6 Using "Variables"

```

# Demonstration of "variables" (.data) in assembly (AT&T)
# Author: github.com/olehermanse

# variables() should be globally available:
.globl variables
.globl _variables          # Mac OSX Compatibility

_variables:                # Mac OSX Compatibility
variables:
    push %ebp              # save base pointer for safe return
    movl %esp, %ebp        # store constant stack pointer ref

    movl num, %eax
    movl $0, num

    addl 8(%ebp), %eax

    # Restore and return:
    pop %ebp
    ret

.data
num:    .long    100

```

## 7 Index of Character and Contains Character

### 7.1 Test program in C

```

// Examples of assembly functions indexOf and contains
// Author: github.com/olehermanse
#include <stdio.h>

extern int indexOf(char c, char* str);
extern int contains(char c, char* str);

int main(void){
    char str[256];
    sprintf(str, "Hello World!");
    printf("\nIndex of 's' in '%s': %d\n", str, indexOf('s', str));
    printf("Does 's' contain 's': %d\n", str, contains('s', str));
    printf("\nIndex of 'l' in '%s': %d\n", str, indexOf('l', str));
    printf("Does 's' contain '!': %d\n\n", str, contains('!', str));
    return 0;
}

```

## 7.2 Function in Assembly

# Example of indexOf Function in asm

# Author: github.com/olehermanse

```
.globl indexOf
.globl _indexOf          # Mac OSX Compatibility

# Name: indexOf.
# Synopsis: What is the index of c in str.
# C-signature: int indexOf(char c, char* str).
# Registers:
# BL:   char c
# ESI:  char* str
# EAX:  int i (Counter / Index)
_indexOf:
indexOf:
    push %ebp             # save base pointer for safe return
    movl %esp, %ebp       # store constant stack pointer ref

    pushl %ebx            # callee saved register
    pushl %esi            # callee saved register

    xorl %eax, %eax       # eax = 0
    xorl %ebx, %ebx       # ebx = 0
    xorl %esi, %esi       # esi = 0

    movb 8(%ebp), %bl      # bl = c
    movl 12(%ebp), %esi    # esi = str

    cmpl $0, %esi
    je notfound

stringloop:              # while(1){
    cmpb %bl, (%esi)      #     if(c == *str)
    je return             #         return i / goto return
    cmpb $0, (%esi)       #     if(*str == 0)
    je notfound           #         return -1 / goto notfound
    incl %esi              #     ++str
    incl %eax              #     ++i
    jmp stringloop        # }

notfound:
    movl $-1, %eax        # return -1

return:
    # Restore and return:
    popl %esi             # Restore callee saved register
    popl %ebx             # Restore callee saved register
    movl %ebp, %esp       # Restore stack pointer if we fucked up push/pops
    popl %ebp             # Standard
    ret
```

```

.globl contains
.globl _contains          # Mac OSX Compatibility
# Name: contains.
# Synopsis: Returns 1 if string contains c and 0 otherwise
# C-signature: int contains(char c, char* str).
# Registers:
# BL:   char c
# ESI:  char* str
_contains:
contains:
    push %ebp              # save base pointer for safe return
    movl %esp, %ebp        # store constant stack pointer ref

    pushl %ebx             # callee saved register
    pushl %esi             # callee saved register

    xorl %eax, %eax        # eax = 0
    xorl %ebx, %ebx        # ebx = 0
    xorl %esi, %esi        # esi = 0

    movb 8(%ebp), %bl      # bl = c
    movl 12(%ebp), %esi    # esi = str
    pushl %esi
    pushl %ebx
    call index0f
    subl $8, %esp

    cmpl $-1, %eax
    je false

true:
    movl $1, %eax          # return 1
    jmp creturn

false:
    movl $0, %eax          # return 0

creturn:
    # Restore and return:
    popl %esi              # Restore callee saved register
    popl %ebx              # Restore callee saved register
    movl %ebp, %esp        # Restore stack pointer if we fucked up push/pops
    popl %ebp              # Standard
    ret

```