# Using the GNU assembler for Intel processors

## 1 Introduction

This tutorial is a minimal description of what you need to know to generate code for an 80386 (or later) Intel processor using the *as* [3] assembler on a Linux system. The appendix contains instructions and modifications for running the example programs using the C library instead of the Linux kernel for input and output.

There are two kinds of assemblers:

1. Intel style assemblers. The main assembler of this kind is *nasm*.
2. AT&T style assemblers, mainly the Gnu assembler, *as* or *gas*.

The main differences between them are the order of the operands in the instructions, the special characters indicating constants, registers, contents of memory locations, operand types ([3], sec. 8.8). They both generate the same machine instructions, so the choice is mostly a matter of taste.

There is a lot of documents on the web describing how to program this processor. A good one, using the Intel style, is by Carter [6].

## 2 The processor

There are eight 32 bit general purpose registers: `eax`, `ebx`, `ecx`, `edx`, `ebp`, `edi`, `esi`, and `esp`. The last one, `esp`, should be reserved for use as a stack pointer. The other registers may be used freely, but some instructions require the use of specific registers. The letters `e` and `x` in the first four registers just indicates that they are 32 bits long. The base pointer register, `ebp`, is conventionally used as a pointer to the current activation record. `edi` and `esi` are used as destination and source pointers by string instructions.

The first four registers may also be used as byte and 16 bits word registers. The name of the last 8 bits of `eax` is `al`, and the last 16 bits may be referred by `ax`. The first 8 bits of `ax` are referred by `ah`. There are analogous names for parts of the `ebx`, `ecx` and `edx` registers. There are some one bit flags that are set by the are set as side effects of some instructions and used by the conditional instructions.

The processor can execute more than 300 different instructions. This document describes about 30 of them. Each instruction has at most two operands. There are four kinds of operands: registers, memory addresses, and constant data held by the instruction. With few exceptions, an instruction can refer to at most one memory location.

# 3 An example

Our first example shows how to compute the number of decimal digits required to print a nonnegative integer. We will repeatedly divide the number by 10 until it becomes 0.

```
        .data                   # allocating memory
n:      .long   234             # the number
length: .long   0               # the result
ten:    .long   10              # the divisor

        .text                   # instructions
        .global _start          # make _start globally known
_start: movl    $0, %ebx        # use ebx as counter
        movl    n, %eax         # copy number to eax
nextdigit:
        movl    $0, %edx        # prepare for long division
        idivl   ten             # divide combined edx:eax registers by 10
                                # quotient to eax, remainder to edx
        addl    $1, %ebx        # add 1 to counter
        cmpl    $0, %eax        # compare eax to 0
        jg      nextdigit       # jump if eax>0
        movl    %ebx, length    # copy counter to memory
        # exit to OS kernel to terminate execution
        movl    $0, %ebx        # first argument: exit code
        movl    $1, %eax        # sys_exit index
        int     $0x80           # kernel interrupt
```

The program has two *sections*, a `.data` section that describes how to allocate memory for global variables, and a `.text` section with the instructions. The sections may appear in either order. Each instruction should start a new line. No indentation is required, but improves readability. Comments start with a `#` character.

Each variable has a label, a type, and an initial value. The label is a name that denotes the memory address of the variable. All variables in this example have type `.long` using 32 bits. `n` is the label of the number to analyze, 234, `length` will contain the number of digits, i.e. 3, on completion of the execution, and `ten` is the divisor.

The first line of the `.text` section is a *directive* making a label accessible outside this section. `_start` is the default label used by the loader for the first instruction to be executed.

All register references start with a percent character. The first instruction,

```
        movl $0, %ebx
```

will set the `ebx` register to 0. `$0` is a *constant* operand. The dollar sign is important. Without it the value at location 0 in memory will be used. We will use the `ebx` register to count the digits.

The next two `movl` instructions will prepare for dividing 234 by 10. The first one

```
        movl n, \%eax
```

will copy the value at the memory location with label `n` to the `eax` register.

Division will be performed on the combined 64 bits of the `edx` and `eax` registers. We set the first register to 0. The `idivl` instruction performs signed integer division, i.e. it assumes that negative numbers are represented as two complements. The `idivl` instruction has one operand, the divisor. This instruction cannot take constant operands, so the value of the operand is fetched from memory location `ten`.

After the execution of the `idivl` instruction the quotient will be in `eax` while `edx` will contain the remainder.

We add 1 to the `ebx` in order to count this digit. The actual value of the digit is in `edx`.

Next 0 is compared to `eax` and if the contents of `eax` is greater than 0 the next instruction, `jg nextdigit`, will make a jump to `nextdigit`. Notice the order of the operands!

After another two divisions the comparison will direct the jump not to occur. The value in `ebx` will be saved in the memory location `length`.

The proper way to terminate the execution is to call the `exit` procedure in the operating system kernel using an *interrupt*. The last three lines do that with an exit code equal to 0 signaling normal return.

The program may now be translated to machine code by the assembler, *as*, loaded by *ld*, and executed with no visible result.

```
> as -o digit1.o digit1.s
> ld -o digit1 digit1.o
> ./digit1
>
```

# 4 Debugging with ddd

We may use the debugger *ddd* to inspect registers and memory during the execution. *ddd* is a graphical user interface to the Gnu debugger, *gdb*. You should invest some time in learning to use it.

The assembler requires an extra option to keep the symbol table in the executable program.

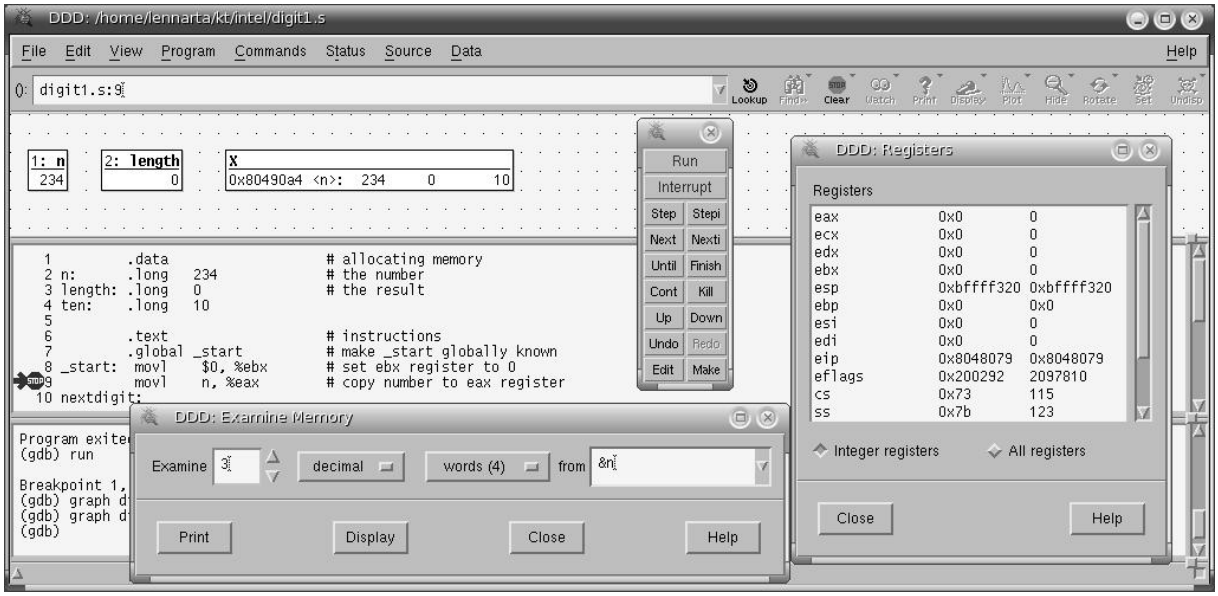
```
> as --gstabs -o digit1.o digit1.s
> ld -o digit1 digit1.o
> ddd digit1&
```

Breakpoints are inserted and deleted by right clicking on the line number and selecting the appropriate item. A breakpoint at the first instruction will have no effect.

The buttons `Run`, **Step**, and **Cont** are used to execute the program, single step the execution, and continue the execution after a break point.

Variables are displayed by right clicking in the same way on the label.

The Register window is opened via **Status→Registers**. Sections of the memory are displayed through the **Data→Memory** dialog.




# 5 Memory allocation

Programming languages supporting recursion will have all variables in activation records on a stack. When using Linux on an Intel based machine there is no need to allocate a stack since the loader, *ld*, will reserve 2 Mb of memory for this purpose. The stack pointer register, `esp`, will point to the top of this stack.

Constant an global data may be allocated in the `.data` section. Some examples:

```
        .data
v32:    .long 0          # 32 bits, initial value 0
v16:    .word 0xffff     # 16 bits, all ones
v8:     .byte '-         # 8 bits, acsii code for -
vs:     .ascii "input"   # string with 5 bytes
vs0:    .asciz "input"   # string with 6 bytes, last byte is 0
        .align 4         # align at 32 bit word boundary
stack:  .skip 1024       # allocate 1024 bytes
tos = .                  # tos will be the address of the next byte
```

Most C functions expect strings to be terminated by a 0 byte generated by `.asciz`.

The directive `.align 4` will allocate the next item at a 4*8 bit address boundary. Memory accesses to 32 bit words should have this alignment.

At the label `stack`, 1024 bytes are allocated.

# 6 Operands

An instruction can have four kinds of operands: *constant*, *register*, *address*, and *implicit*.

A constant is preceded by a dollar sign. Examples: `$10`, `$n`. The last operand denotes the address corresponding to the label `n`.

A register operand starts with a `%` character.

An address operand refers to a value at a memory location. The most common example is a label of a variable or an instruction. The assembler can evaluate simple arithmetic expressions.

| operand | refers to |
|---|---|
| `length` | value at label `length` |
| `length+4` | value at 4 bytes after `length` |
| `length-4` | value at 4 bytes before `length` |
| `nextdigit` | instruction at `nextdigit` |

Some of the following operands are well suited for accessing values in activation records and arrays.

| operand | refers to |
|---|---|
| `(%ebp)` | value at address contained in `ebp` |
| `4(%ebp)` | value at 4 bytes after address contained in `ebp` |
| `stack(%eax)` | value at `stack+eax` |
| `(%ebp,%eax,4)` | value at `ebp+4*eax` |
| `stack(%ebp,%eax,4)` | value at `stack+ebp+4*eax` |

Some instructions have implicit operands, e.g. the `idivl` instruction uses the `eax` and `edx` registers.

# 7 Instructions

The following table lists the most common instructions for arithmetic operations and copying of data. For each instruction the permitted kinds of operands are indicated by the initial letters of register, memory, constant, and the number of bits taking part in the operation. There are similar instructions for byte (8 bits) and word (16 bit) operands. The trailing `l` in instruction names is then replaced by `b` and `w`.

An instruction can have at most one operand of the address type.

| instruction | operands | effect |
|---|---|---|
| `movl` | rmc32, rm32 | rm32 = rmc32 |
| `addl` | rmc32, rm32 | rm32 = rm32+rmc32 |
| `subl` | rmc32, rm32 | rm32 = rm32-rmc32 |
| `negl` | rm32 | rm32 = -rm32 |
| `incl` | rm32 | rm32 = rm32+1 |
| `decl` | rm32 | rm32 = rm32-1 |
| `imull` | rmc32, r32 | r32 = r32*rmc32 |
| `imull` | rm32 | edx:eax = r32*eax, 64 bit result |
| `idivl` | rm32 | eax = edx:eax/rm32, edx = remainder |
| `notl` | rm32 | rm32 = ! rm32, bitwise, false = 0 |
| `andl` | rmc32, rm32 | rm32 = rm32 & rmc32, bitwise |
| `orl` | rmc32, rm32 | rm32 = rm32 \| rmc32, bitwise |
| `cmpl` | $\text{rmc32}_1$, $\text{rmc32}_2$ | compare by computing $\text{rmc32}_2$-$\text{rmc32}_1$ |
| `leal` | m32, r32 | r32 = location denoted by m32 |

The `leal` instruction will compute the address of an operand and save the result in a register.

The `cmpl` instruction (and some other arithmetic instructions) sets some flags that can be used by conditional instructions. These are some of the condition codes, *cc*:

| `l` | `le` | `e` | `ne` | `g` | `ge` |
|---|---|---|---|---|---|
| $<$ | $\leq$ | $=$ | $\neq$ | $>$ | $\geq$ |

A byte can be set to 1 or 0 if the condition holds or not and a word can be copied if the condition holds.

| `set`*cc* rm8 | rm8 = *cc* ? 1 : 0 |
|---|---|
| `cmov`*cc* rm32, r32 | r32 = rm32 if *cc* |

A jump may be unconditional or conditional. The `dest` operand of these instructions should be a label.

| `jmp` dest | jump unconditionally |
|---|---|
| `j`*cc* dest | jump if *cc* |

# 8 Stack instructions

As mentioned above the Linux loader, *ld*, will allocate a stack that is intended for activation records. The loader will set the stack pointer register, `esp`, to the top of the stack. The stack grows towards the bottom of the memory, i.e. to the left in the figure below. The address in the stack pointer is indicated by a thick line. `(%esp)` will refer to the topmost value on the stack.

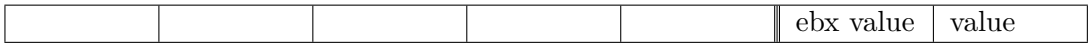

← towards address 0

Pushing and popping 32 bit operands are done with the following instructions.

| instruction | operand | effect |
|---|---|---|
| `pushl` | rmc32 | push value in rmc32 |
| `popl` | rm32 | pop to rm32 |

It is a convention used by the C compiler that the contents of the `ebx` register should be restored to its original value after a procedure call. Hence this value could be pushed on the stack on entry to a procedure

```
pushl %ebx
```




At the end of the procedure the value shoud be restored. Assuming that the value of the stack pointer is the same, `ebx` is restored by

```
popl %ebx
```


When executing a program under an operating system it may be *interrupted* by certain events. During such an interrupt other machine instructions may be executed using the registers and the stack, but all register contents will be restored before control is returned to your procedure.

# 9 Procedure calls

There are some instructions that support procedure calls. Most compilers respect the conventions used by the C compiler. This will make it easier to call procedures compiled by different compilers.

| instruction | operands | effect |
|---|---|---|
| `call` | dest | push return address and jump |
| `ret` | | pop return address and jump |
| `ret` | c32 | pop return address and c32 bytes |
| `int` | c32 | interrupt to kernel |

The instruction pointer, `eip`, holds the address of the next instruction to be executed. The jump instructions modify this register but it cannot be modified directly by a `movl` or an arithmetic instruction.

When a `call dest` instruction is executed the address of the next instruction will be pushed onto the stack and jump to the instruction at the `dest` label will be performed.

To return from a procedure the `ret` instructions should be used. It assumes that the return address is at the top of the stack, pops it and jumps to that location.

The `ret c32` instruction will pop the return address and the specified number of bytes from the stack and proceed execution at the address. This may be used to deallocate the memory used by the arguments of the calling procedure.

The C compiler expects the arguments to be pushed onto the stack in the activation record of the calling procedure before pushing the return address. The arguments should be pushed in reverse order.

The following diagrams show what happens to the stack during a procedure call with two arguments. Before the call the stack pointer, `esp`, points to the topmost word currently in use indicated by a thick line.

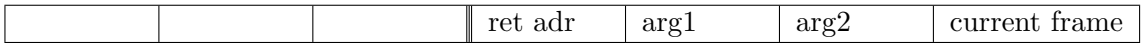

After pushing the two arguments the situation is as follows.

```
pushl arg2
pushl arg1
```


Next we call the procedure, `p`

```
call p
```

The return address is pushed onto the stack:




Assuming that the base pointer, `ebp`, is used to indicate the start of the current activation record, it should be saved on the stack in order to be restored upon return from the procedure. The base pointer is reassigned to make it easy to find the arguments and local variables.

```
pushl %ebp
movl  %esp, %ebp
```

| | | | | | | |
|---|---|---|---|---|---|---|
| | | dyn link | ret adr | arg1 | arg2 | current frame |

The first argument may now be accessed as `8(%ebp)`. At the end the procedure restores the stack pointer and returns:

```
        popl   %ebp
        ret
```

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | arg2 | arg1 | current frame |

After the return the arguments must be popped.

```
        addl $8, %esp
```

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | | current frame |

In the following example the code from page 2 has been extended to print the value of the number and use two procedures. The first procedure computes a string representation of a given nonnegative number and prints it using the OS kernel and the second one just returns control to the kernel.

```
        .text
        .global  _start
_start:
        pushl   $234            # push argument
        call    writeint
        addl    $4, %esp        # pop stack
        pushl   $0              # push argument
        call    exit

writeint:
        pushl   %ebp            # save old base pointer
        movl    %esp, %ebp      # set base pointer
        movl    8(%ebp), %eax   # copy argument to eax
        movl    $10, %ebx       # set divisor to 10
        subl    $12, %esp       # allocate for result string
        movl    %ebp, %edi      # edi points to previous digit
writedigit:
        movl    $0, %edx        # divide edx:eax ...
        idivl   %ebx            # by 10
        addl    $'0, %edx       # convert remainder to ascii
        decl    %edi            # push ...
        movb    %dl, (%edi)     # digit
        cmp     $0, %eax
        jg      writedigit       # jump if eax>0
        # let the OS kernel print the string
        movl    %ebp, %edx      # compute ...
        subl    %edi, %edx      # third argument: string length
        movl    %edi, %ecx      # second argument: string address
        movl    $1, %ebx        # first argument: file descriptor
        movl    $4, %eax        # sys_write call index
        int     $0x80           # kernel interupt
```

```
        addl    $12, %esp       # deallocate result string
        popl    %ebp            # restore base pointer
        ret                     # return
exit:
        movl    4(%esp), %ebx   # first argument: error code
        movl    $1, %eax        # sys_exit call index
        int     $0x80           # kernel interrupt
```

Since these procedures do not use global variables their activation records have no static links.

If you execute this example it may happen that the printed result will not be visible since the shell prompter overprints the result. Use `less` or redirection to avoid this.

The file `eda180.s` shown in an appendix includes versions of `printint` and `readint` that can handle negative numbers and some more procedures that may be useful in the course project. They respect the C conventions restoring the values of the `ebp`, `ebx`, and`edi` registers upon exit.

# 10 Representation of memory words

The Intel architecture uses a peculiar representation for a word in the memory called *little endian*. It means that the bytes within a word are stored in reverse order.

You may observe this when inspecting the same part of the memory using hexadecimal bytes and hexadecimal words. Looking at string data using words the character will appear in reverse order within each word. On the other hand if you inspect 32 bit integer data using bytes the least significant byte will appear first.

This might be confusing but can usually be ignored.

# 11 Block structured languages

There are two instructions that support block structured languages by setting up a *display* of static links that makes access to global variables efficient.

| instruction | operands | effect |
|---|---|---|
| `enter` | c32, c5 | set up dynamic and c5 static links, allocate c32 bytes |
| `leave` | | deallocate ditto and restore `ebp` |

The `enter` instruction will push the contents of the `ebp` register and follow the statics links `c5` times and push each link on the stack. Then it will allocate `c32` bytes for local variables by decrementing `esp`. The second argument should be equal to the static nesting level of the procedure. The main procedure should have level 1.

The `leave` instruction will deallocate the memory used for the links and local variables and restore `ebp`.

The following pseudo code describes in detail the execution of

`enter` c32, c5

when c5$\geq$1 using a temporary variable `frameptr`.

```
pushl %ebp
movl %esp, frameptr
repeat (c5-1) times {
    subl 4,  %ebp
    pushl (%ebp);
}
pushl frameptr
movl frameptr, %ebp
subl c32, %esp
```

The `leave` instruction performs

```
movl %ebp, %esp
popl %ebp
```

There is no illustrating example using these instructions since we believe that it is more instructive to handle the static links by yourself, but you are free to explore this alternative. Notice that the offset of the first variable in a frame depends on the frame level.

# 12 Interfacing C functions

It is easy to call C functions from an assembler program. The following program uses three functions from the standard C library. It reads a number from the keyboard using `scanf`, adds 1, prints the result using `printf`, and terminates by calling `exit`.

```
        .text
        .global main
main:
        pushl   $n              # push second arg, address of n
        pushl   $sfmt           # push first arg, address of sfnt
        call    scanf           # call scanf("%d", &n)
        addl    $8, %esp        # pop 2 arguments
        addl    $1, n
        pushl   n               # push second argument, n
        pushl   $fmt            # push first argument, address of fmt
        call    printf          # call printf("%d\n", eax)
        addl    $8, %esp        # pop 2 arguments
        pushl   $0              # push first argument, exit code = 0
        call    exit            # call exit(0)

        .data
n:      .long   0               # number
fmt:    .asciz  "%d\n"          # format for printf
sfmt:   .asciz  "%d"            # format for scanf
```

The `scanf` function has two arguments. The first argument, `"%d"`, is the format string describing how the input string should be interpreted as a decimal number. The second argument should be the address where the resulting number should be saved.

The `printf` function requires one or more arguments. The first argument is the memory address of a string describing how to print the other arguments. In this case we use the string "`\%d\n`"

to indicate that we shall print a decimal number followed by a newline character. The second argument is the number to be printed.

We terminate the execution by calling the `exit` function with 0 as an argument. Since this call will not return it is useless to restore the stack pointer in a subsequent instruction.

The standard C library is a *shared library*. This means that several processes using the same library function can share one copy in memory. There is some overhead for this that you should leave to the compiler. The compiler can take an assembler program as input. It will generate a small program with the `_start` label and a call to a procedure called `main`. This should be the first label in your assembler program.

By convention, a C function will use the `eax` register when returning a value. You should assume that all registers except `esp`, `ebp`, `ebx`, and `edi` may have changed. You may "compile" the file `writen.s` using the *gcc* (or *cc*) command and execute the program with an option to use *ddd*:

```
> gcc -gstabs -o writen writen.s
> ./writen
234
235
```

A program compiled by the C compiler may call assembler functions. It will then expect the values in `ebx`, `esi`, `edi`, `esp`, and `ebp` to be unchanged upon return.

You may compile a C program and inspect the generated assembler instructions using the `-S` option: `gcc -S -o main.s main.c`

# 13 64 bit mode

When using 64 bit mode the names of the general registers have an initial `r` instead of `e`: `rax`, `rbx`, `rcx`, `rdx`, `rbp`, `rdi`, `rsi`, and `rsp`. The 32 rightmost bits of the first for registers are still available using their old names: `eax`, `ebx`, `ecx`, and `edx`. Most instructions have a 64 bit with *l* replaced by `q`. Exceptions are `push` and `pop`. An example program:

```
        .text
        .global main
        .global _start
_start:
        call main
        movl $0, %ebx
        movl $1, %eax
        int $0x80
main:
        pushq %rbp
        movq %rsp, %rbp
        subq $8, %rsp
        movl $1, 16(%rbp)
        movq %rbp, %rsp
        popq %rbp
        ret
```

For all the details, see [2]

# 14 Mac OS

Appendix 14 contains an example that may be assembled using *gcc* under Mac OS.

When calling external functions compiled by `gcc` like `printf` and `exit` the stack must be aligned on a 16 byte boundary. This means that the last hexadecimal digit of `%esp` must be 0 when the call instruction is executed. Otherwise you will get a `Segmentation error`. See e.g. [4] for further details.

I believe that it is best to let your compiler generate code to assure proper alignment.

# References


[1] Intel, Manuals, `developer.intel.com/design/Pentium4/documentation.htm#manuals`

[2] Intel, Manuals, `www.intel.com/products/processor/manuals/`

[3] Gnu, *Using as*, `gnu-mirror.dkuug.dk/software/binutils/manual/gas-2.9.1/as.html`

[4] Sanglard, F., *IA-32 assembly on Mac OS X*, `http://www.fabiensanglard.net/macosxassembly/index.php`

[5] Zeller, A., *Debugging with ddd*, `www.gnu.org/manual/ddd/pdf/ddd.pdf`

[6] Carter, P.A., *PC Assembly Language*, `www.drpaulcarter.com/pcasm/`

[7] The Netwide Assembler, `sourceforge.net/projects/nasm`

# Appendix. eda180.s



```
        .text
        .global writeint
        ## void writeint(int n)
        ## writes n on sysout

        .global readint
        ## int readint()
        ## reads an integer from sysin
        ## return value is in eax

        .global writestr
        ## void writestr(char[] str, int n)
        ## writes n characters from str on sysout

        .global writeln
        ## void writeln()
        ## writes a newline character on sysout

        .global exit
        ## void exit(int n)
        ## terminates execution with error code n

writeint:
         pushl   %ebp            # save old base pointer
         pushl   %ebx            # save ebx
         pushl   %edi            # save edi
         movl    %esp, %ebp      # set base pointer
         movl    16(%ebp), %eax  # copy argument to eax
         movl    $10, %ecx       # set divisor to 10
         subl    $12, %esp       # allocate for local string
         movl    %ebp, %edi      # set edi
         cmpl    $0, %eax        # argument negative?
         jge     writedigit
         negl    %eax            # negate
writedigit:
         movl    $0, %edx        # set up for 64 bit division
         idivl   %ecx            # divide edx:aex by 10
         addl    $'0, %edx       # ascii digit in edx
         decl    %edi            # push ...
         movb    %dl, (%edi)     # digit
         cmpl    $0, %eax        # quotient=0 ?
         jg      writedigit
         cmpl    $0, 16(%ebp)    # argument negative?
         jge     nosign
         decl    %edi            # push ...
         movb    $'-, (%edi)     # '-'
nosign:  movl    %ebp, %edx
         subl    %edi, %edx      # third argument: string length
         movl    %edi, %ecx      # second argument: string address
         movl    $1, %ebx        # first argument: file descriptor
         movl    $4, %eax        # sys_write interrupt index
         int     $0x80           # kernel interrupt sys_write
         movl    %ebp, %esp      # restore stack pointer
         popl    %edi            # restore edi
         popl    %ebx            # restore ebx
         popl    %ebp            # restore base pointer
         ret                     # return

readint:
         pushl   %ebp            # save old base pointer
         pushl   %ebx            # save ebx
         pushl   %edi            # save edi
```



```
         movl    %esp, %ebp      # set base pointer
         subl    $12, %esp       # allocate memory
         movl    %esp, %edi
         movl    $12, %edx       # maximal string length
         movl    %edi, %ecx      # string address
         movl    $0, %ebx        # file descriptor
         movl    $3, %eax        # sys_read interrupt index
         int     $0x80           # kernel interrupt sys_read
         addl    %eax, %ecx      # address of ...
         decl    %ecx            # last character
         # parse input
         movl    $0, %eax        # result accumulator
         cmpb    $'-, -12(%ebp)  # negative number?
         jne     readdigit
         incl    %edi            # skip '-'
readdigit:
         imull   $10, %eax       # multiply result by 10
         movl    $0, %edx        # copy digit ...
         movb    (%edi), %dl     # to dl
         subl    $'0, %edx       # convert ascii to value
         addl    %edx, %eax      # add value to result
         incl    %edi            # address of next digit
         cmpl    %edi, %ecx      # end of string?
         jg      readdigit
         cmpb    $'-, -12(%ebp)  # negative number?
         jne     rest
         negl    %eax            # negate result
rest:    movl    %ebp, %esp      # restore stack pointer
         popl    %edi            # restore edi
         popl    %ebx            # restore ebx
         popl    %ebp            # restore base pointer
         ret

writestr:
         pushl   %ebx            # save ebx
         movl    12(%esp), %edx  # third argument: string length
         movl    8(%esp), %ecx   # second argument: string address
         movl    $1, %ebx        # first argument: file descriptor
         movl    $4, %eax        # sys_write interrupt index
         int     $0x80           # kernel interrupt sys_write
         popl    %ebx            # restore ebx
         ret                     # return

writeln:
         pushl   %ebx            # save ebx
         pushl   $10             # push newline character
         movl    $1, %edx        # third argument: string length
         movl    %esp, %ecx    # second argument: string address
         movl    $1, %ebx        # first argument: file descriptor
         movl    $4, %eax        # sys_write interrupt index
         int     $0x80           # kernel interrupt sys_write
         addl    $4, %esp        # pop stack
         popl    %ebx            # restore ebx
         ret                     # return

exit:
         movl    4(%esp), %ebx   # first argument: error code
         movl    $1, %eax        # sys_exit interrupt index
         int     $0x80           # kernel interrupt sys_exit
```

# Appendix. Using the C library

Apr 20, 06 9:56 **digit1c.s** Page 1/1

```
        .data                   # allocating memory
n:      .long   234             # the number
length: .long   0               # the result
ten:    .long   10              # the divisor

        .text
        .global main
        .extern exit

main:   movl    $0, %ebx        # use ebx as counter
        movl    n, %eax         # copy number to eax
nextdigit:
        movl    $0, %edx        # prepare for long division
        idivl   ten             # divide combined edx:eax registers by 10
                                # quotient to eax, remainder to edx
        addl    $1, %ebx        # add 1 to counter
        cmpl    $0, %eax        # compare eax to 0
        jg      nextdigit       # jump if eax>0
        movl    %ebx, length    # copy counter to memory
        # call C function exit to terminate execution
        pushl   $0              # first argument: exit code = 0
        call    exit            # call exit(0)

        # compile and debug using:
        #   gcc -g -o digit1c digit1c.s
        #   ddd digit1c&
```

Apr 20, 06 9:58 **digit2c.s** Page

```
        .text
        .global  main
        .extern printf
        .extern exit
main:
        pushl   $234            # push argument
        call    writeint
        addl    $4, %esp        # pop
        push    $0              # push argument
        call    exit

writeint:
        movl    %esp, %ebp      # save stack pointer
        movl    4(%ebp), %eax   # copy argument to eax
        movl    $10, %ebx       # divisor
        decl    %esp            # push ...
        movb    $0, (%esp)      # null character
nextdigit:
        movl    $0, %edx        # divide edx:eax ...
        idivl   %ebx            # by 10
        addl    $'0, %edx       # convert remainder to ascii
        decl    %esp            # push ...
        movb    %dl, (%esp)     # digit
        cmp     $0, %eax
        jg      nextdigit       # jump if eax>0

        # call printf(esp) to print the string
        pushl   %esp            # push address of string
        call    printf
        addl    $4, %esp        # pop argument (unnecessarily)
        movl    %ebp, %esp      # restore stack pointer
        ret

        # compile and run using:
        #   gcc -o digit2c digit2c.s
        #   ./digit2c&
```

# Appendix. Using the C library under Mac OS



```
.data
buffer:
.asciz          "       "
.text

.globl _main

_main:
main:   nop                     # required by ddd?
        pushl   $234            # push argument
        call    writeint
        addl    $4, %esp        # pop argument
        subl    $0x8, %esp      # padding stack
        pushl   $0              # push argument
        call    _exit

writeint:
        movl    %esp, %ebp      # save stack pointer
        movl    4(%ebp), %eax   # copy argument to eax
        movl    $10, %ebx       # divisor
        movl    $buffer, %edi   # set up ...
        addl    $16, %edi       # buffer using %edi
        decl    %edi            # push ...
        movb    $0, (%edi)      # null character

nextdigit:
        movl    $0, %edx        # divide edx:eax ...
        idivl   %ebx            # by 10
        addl    $'0, %edx       # convert remainder to ascii
        decl    %edi            # push ...
        movb    %dl, (%edi)     # digit
        cmp     $0, %eax
        jg      nextdigit       # jump if eax>0

        subl    $0x0, %esp      # padding stack
        pushl   %edi            # push argument
        call    _printf
        popl    %edi            # pop argument
        movl    %ebp, %esp      # restore stack pointer
        ret

# compile and run under Mac OS
#       gcc -gstabs -o digitmac digitmac.s
#       ./digitmac
```