Command: as – assembler

AS—ASSEMBLER [IBM]

This document describes the language accepted by the 80386 assembler that is part of the Amsterdam Compiler Kit. Note that only the syntax is described, only a few 386 instructions are shown as examples.

Tokens, Numbers, Character Constants, and Strings

The syntax of numbers is the same as in C. The constants 32, 040, and 0x20 all represent the same number, but are written in decimal, octal, and hex, respectively. The rules for character constants and strings are also the same as in C. For example, ′a′ is a character constant. A typical string is "string". Expressions may be formed with C operators, but must use [and] for parentheses. (Normal parentheses are claimed by the operand syntax.)

Symbols

Symbols contain letters and digits, as well as three special characters: dot, tilde, and underscore. The first character may not be a digit or tilde.

The names of the 80386 registers are reserved. These are:

al, bl, cl, dl
ah, bh, ch, dh
ax, bx, cx, dx, eax, ebx, ecx, edx
si, di, bp, sp, esi, edi, ebp, esp
cs, ds, ss, es, fs, gs

The xx and exx variants of the eight general registers are treated as synonyms by the assembler. Normally "ax" is the 16-bit low half of the 32-bit "eax" register. The assembler determines if a 16 or 32 bit operation is meant solely by looking at the instruction or the instruction prefixes. It is however best to use the proper registers when writing assembly to not confuse those who read the code.

The last group of 6 segment registers are used for selector + offset mode addressing, in which the effective address is at a given offset in one of the 6 segments.

Names of instructions and pseudo-ops are not reserved. Alphabetic characters in opcodes and pseudo-ops must be in lower case.

Separators

Commas, blanks, and tabs are separators and can be interspersed freely between tokens, but not within tokens. Commas are only legal between operands.

Comments

The comment character is ‘!’. The rest of the line is ignored.

Opcodes

The opcodes are listed below. Notes: (1) Different names for the same instruction are separated by ‘/’. (2) Square brackets ([]) indicate that 0 or 1 of the enclosed characters can be included. (3) Curly brackets ({}) work similarly, except that one of the enclosed characters *must* be included. Thus square brackets indicate an option, whereas curly brackets indicate that a choice must be made.

Data Transfer

```
mov[b]      dest, source      ! Move word/byte from source to dest
pop         dest              ! Pop stack
push        source            ! Push stack
xchg[b]     op1, op2          ! Exchange word/byte
xlat                          ! Translate
o16                           ! Operate on a 16 bit object instead of 32 bit
```

Input/Output

```
in[b]       source            ! Input from source I/O port
in[b]                         ! Input from DX I/O port
out[b]      dest              ! Output to dest I/O port
out[b]                        ! Output to DX I/O port
```

Address Object

```
lds         reg,source        ! Load reg and DS from source
les         reg,source        ! Load reg and ES from source
lea         reg,source        ! Load effect address of source to reg and DS
{cdsefg}seg                   ! Specify seg register for next instruction
a16                           ! Use 16 bit addressing mode instead of 32 bit
```

Flag Transfer

```
lahf                          ! Load AH from flag register
```

```
popf                          ! Pop flags
pushf                         ! Push flags
sahf                          ! Store AH in flag register
```

Addition

```
aaa                           ! Adjust result of BCD addition
add[b]      dest,source       ! Add
adc[b]      dest,source       ! Add with carry
daa                           ! Decimal Adjust after addition
inc[b]      dest              ! Increment by 1
```

Subtraction

```
aas                           ! Adjust result of BCD subtraction
sub[b]      dest,source       ! Subtract
sbb[b]      dest,source       ! Subtract with borrow from dest
das                           ! Decimal adjust after subtraction
dec[b]      dest              ! Decrement by one
neg[b]      dest              ! Negate
cmp[b]      dest,source       ! Compare
```

Multiplication

```
aam                           ! Adjust result of BCD multiply
imul[b]     source            ! Signed multiply
mul[b]      source            ! Unsigned multiply
```

Division

```
aad                           ! Adjust AX for BCD division
o16 cbw                       ! Sign extend AL into AH
o16 cwd                       ! Sign extend AX into DX
cwde                          ! Sign extend AX into EAX
cdq                           ! Sign extend EAX into EDX
idiv[b]     source            ! Signed divide
div[b]      source            ! Unsigned divide
```

Logical

```
and[b]      dest,source       ! Logical and
not[b]      dest              ! Logical not
or[b]       dest,source       ! Logical inclusive or
test[b]     dest,source       ! Logical test
xor[b]      dest,source       ! Logical exclusive or
```

Shift

```
sal[b]/shl[b]   dest,CL        ! Shift logical left
sar[b]          dest,CL        ! Shift arithmetic right
shr[b]          dest,CL        ! Shift logical right
```

Rotate

```
rcl[b]          dest,CL        ! Rotate left, with carry
rcr[b]          dest,CL        ! Rotate right, with carry
rol[b]          dest,CL        ! Rotate left
ror[b]          dest,CL        ! Rotate right
```

String Manipulation

```
cmps[b]                        ! Compare string element ds:esi with es:edi
lods[b]                        ! Load from ds:esi into AL, AX, or EAX
movs[b]                        ! Move from ds:esi to es:edi
rep                            ! Repeat next instruction until ECX=0
repe/repz                      ! Repeat next instruction until ECX=0 and ZF=1
repne/repnz                    ! Repeat next instruction until ECX!=0 and ZF=0
scas[b]                        ! Compare ds:esi with AL/AX/EAX
stos[b]                        ! Store AL/AX/EAX in es:edi
```

Control Transfer

As accepts a number of special jump opcodes that can assemble to instructions with either a byte displacement, which can only reach to targets within –126 to +129 bytes of the branch, or an instruction with a 32-bit displacement. The assembler automatically chooses a byte or word displacement instruction.

The English translation of the opcodes should be obvious, with ‘‘l(ess)’’ and ‘‘g(reater)’’ for signed comparisions, and ‘‘b(elow)’’ and ‘‘a(bove)*(CQ for unsigned comparisions. There are lots of synonyms to allow you to write "jump if not that" instead of "jump if this".

The ‘‘call’’, ‘‘jmp’’, and ‘‘ret’’ instructions can be either intrasegment or intersegment. The intersegment versions are indicated with the suffix ‘‘f’’.

Unconditional

```
jmp[f]          dest           ! jump to dest (8 or 32-bit displacement)
call[f]         dest           ! call procedure
ret[f]                         ! return from procedure
```

Conditional

```
ja/jnbe                 ! if above/not below or equal (unsigned)
jae/jnb/jnc             ! if above or equal/not below/not carry (uns.)
jb/jnae/jc              ! if not above nor equal/below/carry (unsigned)
jbe/jna                 ! if below or equal/not above (unsigned)
jg/jnle                 ! if greater/not less nor equal (signed)
jge/jnl                 ! if greater or equal/not less (signed)
jl/jnqe                 ! if less/not greater nor equal (signed)
jle/jgl                 ! if less or equal/not greater (signed)
je/jz                   ! if equal/zero
jne/jnz                 ! if not equal/not zero
jno                     ! if overflow not set
jo                      ! if overflow set
jnp/jpo                 ! if parity not set/parity odd
jp/jpe                  ! if parity set/parity even
jns                     ! if sign not set
js                      ! if sign set
```

Iteration Control

```
jcxz        dest        ! jump if ECX = 0
loop        dest        ! Decrement ECX and jump if CX != 0
loope/loopz dest        ! Decrement ECX and jump if ECX = 0 and ZF = 1
loopne/loopnz           dest! Decrement ECX and jump if ECX != 0 and ZF = 0
```

Interrupt

```
int         n           ! Software interrupt n
into                    ! Interrupt if overflow set
iretd                   ! Return from interrupt
```

Flag Operations

```
clc                     ! Clear carry flag
cld                     ! Clear direction flag
cli                     ! Clear interrupt enable flag
cmc                     ! Complement carry flag
stc                     ! Set carry flag
std                     ! Set direction flag
sti                     ! Set interrupt enable flag
```

Location Counter

The special symbol ‘.’ is the location counter and its value is the address of the first byte of the instruction in which the symbol appears and can be used in expressions.

Segments

There are four different assembly segments: text, rom, data and bss. Segments are declared and selected by the *.sect* pseudo-op. It is customary to declare all segments at the top of an assembly file like this:

```
.sect .text; .sect .rom; .sect .data; .sect .bss
```

The assembler accepts up to 16 different segments, but MINIX expects only four to be used. Anything can in principle be assembled into any segment, but the MINIX bss segment may only contain uninitialized data. Note that the ‘.’ symbol refers to the location in the current segment.

Labels

There are two types: name and numeric. Name labels consist of a name followed by a colon (:).

The numeric labels are single digits. The nearest 0: label may be referenced as 0f in the forward direction, or 0b backwards.

Statement Syntax

Each line consists of a single statement. Blank or comment lines are allowed.

Instruction Statements

The most general form of an instruction is

```
label: opcode operand1, operand2    ! comment
```

Expression Semantics

The following operators can be used: + – * / & | ^ ~ << (shift left) >> (shift right) – (unary minus). 32-bit integer arithmetic is used. Division produces a truncated quotient.

Addressing Modes

Below is a list of the addressing modes supported. Each one is followed by an example.

constant	mov eax, 123456
direct access	mov eax, (counter)
register	mov eax, esi
indirect	mov eax, (esi)
base + disp.	mov eax, 6(ebp)
scaled index	mov eax, (4*esi)
base + index	mov eax, (ebp)(2*esi)
base + index + disp.	mov eax, 10(edi)(1*esi)

Any of the constants or symbols may be replacement by expressions. Direct access, constants and displacements may be any type of expression. A scaled index with scale 1 may be written without the ‘‘1*’’.

Call and Jmp

The ‘‘call’’ and ‘‘jmp’’ instructions can be interpreted as a load into the instruction pointer.

call _routine	! Direct, intrasegment
call (subloc)	! Indirect, intrasegment
call 6(ebp)	! Indirect, intrasegment
call ebx	! Direct, intrasegment
call (ebx)	! Indirect, intrasegment
callf (subloc)	! Indirect, intersegment
callf seg:offs	! Direct, intersegment

Symbol Assigment

Symbols can acquire values in one of two ways. Using a symbol as a label sets it to ‘‘.’’ for the current segment with type relocatable. Alternative, a symbol may be given a name via an assignment of the form

symbol = expression

in which the symbol is assigned the value and type of its arguments.

Storage Allocation

Space can be reserved for bytes, words, and longs using pseudo-ops. They take one or more operands, and for each generate a value whose size is a byte, word (2 bytes) or long (4 bytes). For example:

```
.data1 2, 6          ! allocate 2 bytes initialized to 2 and 6
.data2 3, 0x10       ! allocate 2 words initialized to 3 and 16
.data4 010           ! allocate a longword initialized to 8
.space 40            ! allocates 40 bytes of zeros
```

allocates 50 (decimal) bytes of storage, initializing the first two bytes to 2 and 6, the next two words to 3 and 16, then one longword with value 8 (010 octal), last 40 bytes of zeros.

String Allocation

The pseudo-ops *.ascii* and *.asciz* take one string argument and generate the ASCII character codes for the letters in the string. The latter automatically terminates the string with a null (0) byte. For example,

```
.ascii "hello"
.asciz "world\n"
```

Alignment

Sometimes it is necessary to force the next item to begin at a word, longword or even a 16 byte address boundary. The *.align* pseudo-op zero or more null byte if the current location is a multiple of the argument of .align.

Segment Control

Every item assembled goes in one of the four segments: text, rom, data, or bss. By using the *.sect* pseudo-op with argument *next items to go in a particular segment.*

External Names

A symbol can be given global scope by including it in a *.define* pseudo-op. Multiple names may be listed, separate by commas. It must be used to export symbols defined in the current program. Names not defined in the current program are treated as "undefined external" automatically, although it is customary to make this explicit with the *.extern* pseudo-op.

Common

The *.comm* pseudo-op declares storage that can be common to more than one module. There are two arguments: a name and an absolute expression giving the size in bytes of the area named by the symbol. The type of the symbol becomes external. The statement can appear in any segment. If you think this has something to do with FORTRAN, you are right.

Examples

In the kernel directory, there are several assembly code files that are worth inspecting as examples. However, note that these files, are designed to first be run through the C preprocessor. (The very first character is a # to signal this.) Thus they contain numerous constructs that are not pure assembler. For true assembler examples, compile any C program provided with MINIX using the **–S** flag. This will result in an assembly language file with a suffix with the same name as the C source file, but ending with the .s suffix.