The NASM assembler is an open source project to develop a Net-wide Assembler. The assembler is included as standard in most Linux distributions and is available for download to run under Windows. It provides support for the full Intel and AMD SIMD instruction-sets and also recognises some extra MMX instructions that run on Cyrix CPUs. NASM provides support for multiple object module formats from the old MS-DOS com files to the obj and elf formats used under Windows and Linux. If you are programming in assembler, NASM provides a more complete range of instructions, in association with better portability between operating systems than competing assemblers. Microsoft's MASM assembler is restricted to Windows. The GNU assembler, `as`, runs under both Linux and Windows, but uses non-standard syntax which makes it awkward to use in conjunction with Intel documentation.

It is beyond the scope of this course to provide a complete guide to assembler programming for the Intel processor family. Readers wanting a general background in assembler programming should consult appropriate text books [] in conjunction with the processor reference manuals published by Intel[**?**][**?**] and AMD [**?**].

General instruction syntax

Assembler programs take the form of a sequence of lines with one machine instruction per line. The instructions themselves take the form of an optional label, an operation code name conditionally followed by up to three comma separated operands. For example:

```
l1: SFENCE                          ; 0 operand ins
    PREFETCH  [100]                 ; 1 operand ins
    MOVQ MM0,MM1                    ; 2 operand ins
    PSHUFD XMM1,XMM3,00101011b      ; 3 operand ins
```

As shown above, a comment can be placed on an assembler line, with the comment distinguished from the instruction by a leading semi-colon. The label, if present is separated from the operation code name by a colon.

Case is significant neither in operation code names nor in the names of registers. Thus `prefetch` is equivalent to `PREFETCH` and `mm4` equivalent to `MM4`.

In the NASM assembler, as in the original Intel assembler, the direction of assignment in an instruction follows high level language conventions. It is always from right to left*, so that

```
MOVQ MM0,MM4
```

is equivalent to

*If you chose to use the GNU assembler as, instead of NASM you should be aware that this follows the opposite convention of left to right assignment. This is a result of as having originated as a Motorola assembler that was converted to recognise Intel opcodes. Motorola follow a left to right assignment convention.

```
MM0:=MM4
```

and

```
ADDSS XMM0,XMM3
```

is equivalent to

```
XMM0:= XMM0 + XMM3
```

Operand forms

Operands to instructions can be constants, register names or memory locations.

Constants

Constants are values known at assembly time, and take the form of numbers, labels, characters or arithmetic expressions whose components are themselves constants.

The most important constant values are numbers. Integer numbers can be written in base 16, 10, 8 or 2.

```
mov al,0a2h         ; base 16 leading zero requi
mov bh,$0a2         ; base 16 alternate notation
mov cx,0xa2         ; base 16 C style
add ax,101          ; base 10
mov bl,76q          ; base 8
xor ax,11010011b    ; base 2
```

Floating point constants are also supported as operands to store allocation directives (see section **??**):

```
dd 3.14156
dq 9.2e3
```

It is important to realise that due to limitations of the AMD and Intel instruction-sets, floating point constants can not be directly used as operands to instructions. Any floating point constants used in an algorithm have to be assembled into a distinct area of memory and loaded into registers from there.

Labels

Constants can also take the form of labels. As the assembler program is processed, NASM allocates an integer value to each label. The value is either the address of the operation-code prefixed by the instruction or may have been explicitly set by an `EQU` directive:

```
Fseek equ 23
Fread equ 24
```

We can load a register with the address referred to by a label by including the label as a constant operand:

```
mov esi, sourcebuf
```

Using the same syntax we can load a register with an equated constant:

```
mov cl, fread
```

Constant expressions

Suppose there exists a data-structures for which one has a base address label, it is often convenient to be able to refer to fields within this structure in terms of their offset from the start of the structure. Consider the example of a vector of 4 single precision floating point values at a location with label `myvec`. The actual address at which `myvec` will be placed is determined by NASM, we do not know it. We may know that we want the address of the 3rd element of the vector:

```
mov esi, myvec + 3 *4
```

will place the address of this word into the esi register.

Constant expressions

NASM allows one to place arithmetic expressions whose sub-expressions are constants wherever a constant can occur. The arithmetic operators are written C style as shown below.

operator	means	operator	means
|	or	+	add
^	xor	-	subtract
&	and	*	multiply
<<	shift left	/	signed division
>>	shift right	//	unsigned division
%	modulus	%%	unsigned modulus

Registers

Operands can be register names. The available register names are shown in table **??**. In the binary operation codes interpreted by the CPU, registers are identified using 3-bit integers. Depending on the operation code, these 3 bit fields are interpreted as the different categories of register shown in table **??**.

You should be aware that in the Intel architecture a number of registers are aliased to the same state vectors, thus for example the `eax`, `ax`, `al`, `ah` registers all share bits. More insidiously the floating point registers ST0..ST7 not only share state with the MMX registers, but their mapping to these registers is dynamic and variable.

number	byte reg	word reg	dword reg	float reg	nnx reg	sse reg
	Aliased			Aliased		
0	al	ax	eax	st0	mm0	xmm0
1	cl	bx	ecx	st1	mm1	xmm1
2	dl	cx	edx	st2	mm2	xmm2
3	bl	bx	ebx	st3	mm3	xmm3
4	ah	sp	esp	st4	mm4	xmm4
5	ch	bp	ebp	st5	mm5	xmm5
6	dh	si	esi	st6	mm6	xmm6
7	bh	di	edi	st7	mm7	xmm7

Memory Locations

Memory locations are syntactically represented by the use of square brackets around an address expression thus: `[100]`, `[myvec]`, `[esi]` all represent memory locations.

The address expressions, unlike constant expressions, can contain components whose values are not known until program execution. The final example above refers to the memory location addressed by the value in the `esi` register, and as such, depends on the history of prior computations affecting that register.

Address expressions have to be encoded into machine instructions, and since machine instructions, although of variable length on a CISC are nonetheless finite, so too must the address expressions be. On Intel and AMD machines this constrains the complexity of address expressions to the following grammer:

```
memloc::= address | format address
format::= byte | word | dword | qword
address::= [ const ] | [ aexp ] | [ aexp + con
aexp::= reg | reg + iexp
iexp::= reg | reg * scale
scale::= 2|4|8
reg::= eax | ecx | ebx | edx | esp | ebp | esi
const::= integer | label
```

Examples

```
byte[edx]
```

byte pointed to by edx

```
dword[edx+10]
```

4 byte word pointed to by edx+10

```
dword[edx+esi+34]
```

4 byte word at the address given by the sum of the esi and edx registers +34

```
word[eax+edi*4+200]
```

2 byte word at the address given by the eax register + 4* edi register + 200

The format qualifiers are used to disambiguate the size of an operand in memory where the combination of the operation code name and the other non-memory operands are insufficient so to do.

Sectioning

Programs running under Linux have their memory divided into 4 sections:

`text` is the section of memory containing operation codes to be executed. It is typically mapped as read only by the paging system.

`data` is the section of memory containing initialised global variables, which can be altered following the start of the program.

`bss` is the section containing uninitialsed global variables.

`stack` is the section in which dynamically allocated local variables of subroutines are located.

The section directive is used by assembler programers to specify into which section of memory they want subsequent lines of code to be assembled. For example in the listing shown in algorithm **??** we divide the program into three sections: a `text` section containing `myfunc`, a bss section containing 64 undefined bytes and a data section containing a vector of 4 integers.

The label `myfuncbase` can be used with *negative* offsets to access locations within the `bss`, wilst the label `myfuncglobal` can be used with *positive* offsets to access elements of the vector in the data section.

Algorithm 1 Examples of the use of section and data reservation directives

```
section .text
        global myfunc
myfunc:enter 128,0
; body of function goes here
        leave
        ret 0
section .bss
         alignb 16
         resb 64  ; reserve 64 bytes
myfuncBase:
section .data
myfuncglobal: ; reserve 4  by 32-bit integers
          dd 1
          dd 2
          dd 3
          dd 5
```

Data reservation

Data must be reserved in distinct ways in the different sections. In the `data` section, the data definition directives `db`, `dw`, `dd`, and `dq` are used to define bytes, words, doublewords and quad words. The directive must be followed by a constant expression. When defining bytes or words the constant must be an integer. Doublewords and quadwords may be defined with floating point or integer constants as shown previously.

In the bss section the directive `resb` is used to reserve a specified number of bytes, but no value is associated with these bytes.

Stack data

Data can be allocated in the stack section by use of the `enter` operation code name. This takes the form:

```
enter space, level
```

It should be used as the first operation code name of a function. The level parameter is only of relevance in block structured languages and should be set to 0 for assembler programming. The space parameter specifies the number of bytes to be reserved for the private use of the function. Once the `enter` instruction has executed, the data can be accessed at *negative* offsets from the `ebp` register.

Releasing stack space dynamically

The last two instructions in a function should, as shown in algorithm be

```
leave
ret 0
```

The combined effect of these is to free the space reserved on the stack by enter, and pop the return address from the stack. The parameter to the operation code name `ret` is used to specify how many bytes of function parameters should be discarded from the stack. If one is interfacing to C this should always be set to 0.

Label qualification

The default scope of a label is the assembler source file containing the line it prefixes. But labels can be used to mark the start of functions that are to be called from C or other high level languges. To indicate that they have scope beyond the current asscembler file, the `global` directive should be used as shown in algorithm **??**.

The converse case, where an assembler file calls a function exported by a C program is handled by the `etern` directive:

```
extern printreal
```

```
call printreal
```

in the above example we assume that `printreal` is a C function called from assembler.

Linking and object file formats

There are 4 object file formats that are commonly used on Linux and Windows systems as shown in table **??**. This lists the name of the format, its file extension - which is often ambiguous and the combination of operating system and compiler that makes use of it. A flag provided to Nasm specifies which format it should use. We will only go into the use of the gcc compiler, since this is portable between Windows and Linux.

Let us assume we have a C program called `c2asm.c` and an assembler file `asmfromc.asm`. Suppose we wish to combine these into a single executable module `c2asm`. We issue the following commands at the console:

```
nasm -felf -o asmfromc.o asmfromc.asm
```

```
gcc -oc2asm c2asm.c asmfromc.o
```

This assumes that we are working either under Linux or under Cygwin. If we are using djgpp we type:

```
nasm -fcoff -o asmfromc.o asmfromc.asm
gcc -oc2asm c2asm.c asmfromc.o
```

Format	Extension	Operating System	C++ Co
win32	.obj	Windows	Microsoft
obj	.obj	Windows	Borland
coff	.o	Windows	Djgpp
.elf	.o	Windows	Cygwin
.elf	.o	Linux	gcc

Leading underbars

If working with djgpp all external labels in your program, whether imported with `extern` or imported with `global` must have a leading underbar character. Thus to call the C procedure `printreal` one would write:

```
extern _printreal

call _printreal
```

whilst to export myfunc one would write

```
    global _myfunc
    _myfunc:enter 128,0
```