flat assembler 1.40

Programmer's Manual

Tomasz Grysztar

document version 0.5.6

Contents

Chapter 1

Introduction

This chapter contains all the most important information you need to begin using the flat assembler. If you are experienced assembly language programmer, you should read at least this chapter before using this compiler.

1.1 Compiler overview

Flat assembler is a fast assembly language compiler for the Intel Architecture processors, which does multiple passes to optimize the size of generated machine code. It is self–compilable and versions for different operating systems are provided. The `fasm.exe` file is a dual executable, which contains both the DOS and Windows version, the Linux version is available in a separate file. All the versions are designed to be used from the system command line and they should not differ in behavior.

1.1.1 System requirements

All versions require the Intel Architecture 32–bit processor (at least 80386), although they can produce programs for Intel Architecture 16–bit processors, too. DOS version requires an OS compatible with MS DOS 2.0, Windows version requires a Win32 console compatible with 3.1 version.

1.1.2 Executing compiler from command line

To execute flat assembler from the command line you need to provide two parameters – first should be name of source file, second should be name of destination file. After displaying short information about the program name and version, compiler will read the data from source file and compile it. When

the compilation is successful, compiler will write the generated code to the destination file and display the summary of compilation process; otherwise it will display the information about error that occurred.

The source file should be a text file, and can be created in any text editor. Line breaks are accepted in both DOS and Unix standards, tabulators are treated as spaces.

There are no additional command line options, flat assembler requires only the source code to include the information it really needs. For example, to specify output format you specify it by using the `format` directive at the beginning of source.

1.1.3 Compiler messages

As it is stated above, after the successful compilation compiler displays the compilation summary. It includes the information of how many passes was done, how much time it took, and how many bytes were written into destination file. Here is an example of the compilation summary:

```
flat assembler  version 1.40
38 passes, 5.3 seconds, 77824 bytes.
```

In case of error during the compilation process, program will display an error message. For example, when compiler can't find the input file, it will display the following message:

```
flat assembler  version 1.40
error: source file not found.
```

If the error is connected with a specific part of source code, the source line that caused the error will be also displayed. Also placement of this line in the source is given to help you finding this error, for example:

```
flat assembler  version 1.40
example.asm [3]:
        mob     ax,1
error: illegal instruction.
```

It means that in the third line of the `example.asm` file compiler has encountered an unrecognized instruction. When the line that caused error contains a macroinstruction, also number of erroneous line inside the macroinstruction is gived:

```
flat assembler  version 1.40
example.asm [6] stoschar [2]:
        stoschar 7
error: illegal instruction.
```

It means that the macroinstruction in the sixth line of the `example.asm` file contained an unrecognized instruction in the second line of its definition.

1.1.4 Output formats

By default, when there is no `format` directive in source file, flat assembler simply put generated instruction codes into output, creating this way flat binary file. By default it generates 16–bit code, but you can always turn it into the 16–bit or 32–bit mode by using `use16` or `use32` directive. Some of the output formats switch into 32–bit mode, when selected – more information about formats which you can choose can be found in 2.4.

All output code is always in the order in which it was entered into the source file.

1.2 Assembly syntax

The information provided below is intended mainly for the assembler programmers that have been using some other assembly compilers before. If you are beginner, please look for the assembly programming tutorials in chapter 3.

Flat assembler by default uses the Intel syntax for the assembly instructions, although you can customize it using the preprocessor capabilities (macroinstructions and symbolic constants). It also has its own set of the directives – the instructions for compiler.

All symbols defined inside the sources are case–sensitive.

1.2.1 Instruction syntax

Instructions in assembly language are separated by line breaks, and one instruction is expected to fill the one line of text. If line contains a semicolon (except for the semicolons in quoted strings), the rest of this line is the comment and compiler ignores it. If line contains \ characters, the next line is attached at this point. Line should not contain anything but comments (started with semicolon) after the \ character.

Every instruction consists of the mnemonic and the various number of operands, separated with commas. The operand can be register, immediate

Operator	Bits	Bytes
`byte`	8	1
`word`	16	2
`dword`	32	4
`fword`	48	6
`pword`	48	6
`qword`	64	8
`tword`	80	10
`dqword`	128	16

Table 1.1: Size operators.

value or a data addressed in memory, it can also be preceded by size operator to define or override its size (table 1.1). Names of available registers you can find in table 1.2, their sizes cannot be overridden. Immediate value can be specified by any numerical expression.

When operand is a data in memory, the address of that data (also any numerical expression, but it may contain registers) should be enclosed in square brackets or preceded by `ptr` operator. For example instruction `mov eax,3` will put the immediate value 3 into the `eax` register, instruction `mov eax,[7]` will put the 32–bit value from the address 7 into `eax` and the instruction `mov byte [7],3` will put the immediate value 3 into the byte at address 7, it can also be written as `mov byte ptr 7,3`. To specify which segment register should be used for addressing, segment register name followed with a colon should be put just before the address value (inside the square brackets or after the `ptr` operator).

Type	Bits								
General	8	`al`	`cl`	`dl`	`bl`	`ah`	`ch`	`dh`	`bh`
	16	`ax`	`cx`	`dx`	`bx`	`sp`	`bp`	`si`	`di`
	32	`eax`	`ecx`	`edx`	`ebx`	`esp`	`ebp`	`esi`	`edi`
Segment	16	`es`	`cs`	`ss`	`ds`	`fs`	`gs`		
Control	32	`cr0`		`cr2`	`cr3`	`cr4`			
Debug	32	`dr0`	`dr1`	`dr2`	`dr3`	`dr4`	`dr5`	`dr6`	`dr7`
FPU	80	`st0`	`st1`	`st2`	`st3`	`st4`	`st5`	`st6`	`st7`
MMX	64	`mm0`	`mm1`	`mm2`	`mm3`	`mm4`	`mm5`	`mm6`	`mm7`
SSE	128	`xmm0`	`xmm1`	`xmm2`	`xmm3`	`xmm4`	`xmm5`	`xmm6`	`xmm7`

Table 1.2: Registers.

1.2.2 Data definitions

To define data or reserve a space for it, use one of the directives listed in table 1.3. The data definition directive should be followed by one or more of numerical expressions, separated with commas. These expression define the values for data cells of size depending on which directive is used. For example `db 1,2,3` will define the three bytes of values 1, 2 and 3 respectively.

The `db` and `du` directives also accept the quoted string values of any length, which will be converted into chain of bytes when `db` is used and into chain of words with zeroed high byte when `du` is used. For example `db 'abc'` will define the three bytes of values 61, 62 and 63.

The `dp` directive and its synonym `df` accept the values consisting of two numerical expressions separated with colon, the first value will become the high word and the second value will become the low double word of the far pointer value. The `dt` directive accepts only floating point values and creates data in FPU temporary format.

The `file` is a special directive and its syntax is different. This directive includes a chain of bytes from file and it should be followed by the quoted file name, then optionally numerical expression specifying offset in file preceded by the colon, then – also optionally – comma and numerical expression specifying count of bytes to include (if no count is specified, all data up to the end of file is included).

Size (bytes)	Define data	Reserve data
1	`db` `file`	`rb`
2	`dw` `du`	`rw`
4	`dd`	`rd`
6	`dp` `df`	`rp` `rf`
8	`dq`	`rq`
10	`dt`	`rt`

Table 1.3: Data directives.

The data reservation directive should be followed by only one numerical expression, and this value defines how many cells of the specified size should be reserved. All data definition directives also accept the `?` value, which means that this cell should not be initialized to any value and the effect

is the same as by using the data reservation directive. The uninitialized data may not be included in the output file, so its values should be always considered unknown.

1.2.3 Constants and labels

In the numerical expressions you can also use constants or labels instead of numbers. To define the constant or label you should use the specific directives. Each label can be defined only once and it is accessible from the any place of source (even before it was defined). Constant can be redefined many times, but in this case it is accessible only after it was defined, and is always equal to the value from last definition before the place where it's used. When constant is defined only once in source, it's – like the label – accessible from anywhere.

The definition of constant consists of name of the constant followed by the `=` character and numerical expression, which after calculation will become the value of constant. This value is always calculated at the time the constant is defined. For example you can define `count` constant by using the directive `count = 17`, and then use it in the assembly instructions, like `mov cx,count` – which will become `mov cx,17` during the compilation process.

There are different ways to define labels. The simplest is to follow the name of label by the colon, this directive can even be followed by the other instruction in the same line. It defines the label whose value is equal to offset of the point where it's defined. This method is usually used to label the places in code. The other way is to follow the name of label (without a colon) by some data directive. It defines the label with value equal to offset of the beginning of defined data, and remembered as a label for data with cell size as specified for that data directive in table 1.3.

The label can be treated as constant of value equal to offset of labelled code or data. For example when you define data using the labelled directive `char db 224`, to put the offset of this data into `bx` register you should use `mov bx,char` instruction, and to put the value of byte addressed by `char` label to `dl` register, you should use `mov dl,[char]` (or `mov dl,ptr char`). But when you try to assemble `mov ax,[char]`, it will cause an error, because fasm compares the sizes of operands, which should be equal. You can force assembling that instruction by using size override: `mov ax,word [char]`, but remember that this instruction will read the two bytes beginning at `char` address, while it was defined as a one byte.

The last and the most flexible way to define labels is to use `label` directive. This directive should be followed by the name of label, then optionally size operator and then – also optionally `at` operator and the numerical ex-

pression defining the address at which this label should be defined. For example `label wchar word at char` will define a new label for the 16–bit data at the address of `char`. Now the instruction `mov ax,[wchar]` will be after compilation the same as `mov ax,word [char]`. If no address is specified, `label` directive defines the label at current offset. Thus `mov [wchar],57568` will copy two bytes while `mov [char],224` will copy one byte to the same address.

The label whose name begins with dot is treated as local label, and its name is attached to the name of last global label (with name beginning with anything but dot) to make the full name of this label. So you can use the short name (beginning with dot) of this label anywhere before the next global label is defined, and in the other places you have to use the full name. Label beginning with two dots are the exception - they are like global, but they don't become the new prefix for local labels.

The `@@` name means anonymous label, you can have defined many of them in the source. Symbol `@b` (or equivalent `@r`) references the nearest preceding anonymous label, symbol `@f` references the nearest following anonymous label. These special symbol are case–insensitive.

The `load` directive allows to define constant with binary value loaded from a file during the assembly process. This directive should be followed by the name of constant, then optionally size operator, then `from` operator and quoted file name, which can be also followed by a colon and numerical expression specifying offset in file. The size operator has unusual meaning in case – it states how many bytes (up to 8) have to be loaded to form the binary value of constant. If no size operator is specified, one byte is loaded (thus value is in range from 0 to 255). With giving only the offset value instead of quoted file name you can also load a value from the already assembled code. The given offset should be a valid address in currently generated code space, the loaded data cannot exceed current offset.

1.2.4 Numerical expressions

In the above examples all the numerical expressions were the simple numbers, constants or labels. But they can be more complex, by using the arithmetical or logical operators for calculations at compile time. All these operator with their priority values are listed in table 1.4. The operations with higher priority value will be calculated first, you can of course change this behaviour by putting some parts of expression into parenthesis. The `+`, `-`, `*` and `/` are standard arithmetical operations, `mod` calculates the remainder from division. The `and`, `or`, `xor`, `shl`, `shr` and `not` perform the same logical operations as assembly instructions of those names. The `rva` is specific to

PE output format and performs the conversion of an address into the RVA.

Priority	Operators
0	`+` `-`
1	`*` `/`
2	`mod`

3	`and`
	`or`
	`xor`

4	`shl` `shr`
5	`not`
6	`rva`

Table 1.4: Arithmetical and logical operators by priority.

The numbers in the expression are by default treated as a decimal, binary numbers should have the `b` letter attached at the end, octal number should begin with `0` digit (like in C language) or end with `o` letter, hexadecimal numbers should begin with `0x` characters (like in C language) or with the `$` character (like in Pascal language) or they should end with `h` letter. Also quoted string, when encountered in expression, will be converted into numbers – the first character will become the least significant byte of number.

The numerical expression used as an address value can also contain any of general registers used for addressing, they can be added and multiplied by appropriate values, as it is allowed for Intel Architecture instructions.

There are also two special symbols that can be used inside the numerical expression. First is `$`, which is always equal to the value of current offset. Second is `%`, which is the number of current repeat in parts of code that are repeated using some special directives (see 2.2).

The numerical expression can also consist of single floating point value (flat assembler does not allow any floating point operations at compilation time) in the scientific notation, they can end with the `f` letter to be recognized, otherwise they should contain at least one of the `.` or `E` characters.

1.2.5 Jumps and calls

The operand of any jump or call instruction can be preceded not only by the size operator, but also by one of the operators specifying type of the jump: `near` of `far`. For example, when assembler is in 16–bit mode, instruction `jmp dword [0]` will become the far jump and when assembler is in 32-bit mode, it will become the near jump. To force this instruction to be treated differently, use the `jmp near dword [0]` or `jmp far dword [0]` form.

When operand of near jump is the immediate value, assembler will generate the shortest variant of this jump instruction if possible (but won't create 32–bit instruction in 16–bit mode nor 16–bit instruction in 32–bit mode, unless there is a size operator stating it). By specifying the size operator you can force it to always generate long variant (for example `jmp word 0` in 16–bit mode and `jmp dword 0` in 32–bit mode) or to always generate short variant and terminate with an error when it's impossible (for example `jmp byte 0`).

1.2.6 Size settings

When instruction uses some memory addressing, by default the shorter 8–bit form is generated if only address value fits in range, but it can be overridden using the `word` or `dword` operator before the address inside the square brackets (or after the `ptr` operator).

Instructions `adc`, `add`, `and`, `cmp`, `or`, `sbb`, `sub` and `xor` with first operand being 16–bit or 32–bit are by default generated in shortened 8–bit form when the second operand is immediate value fitting in the range for signed 8-bit values. It also can be overridden by putting the `word` or `dword` operator before the immediate value.

Immediate value as an operand for `push` instruction without a size operator is by default treated as a word value if assembler is in 16–bit mode and as a double word value if assembler is in 32–bit mode, shorter 8–bit form of this instruction is used if possible, `word` or `dword` size operator forces the `push` instruction to be generated in longer form for specified size.

Chapter 2

Instruction set

This chapter provides the detailed information about the instructions and directives supported by flat assembler. Directives for defining constants and labels were already discussed in 1.2.3, all other directives will be described later in this chapter.

2.1 Intel Architecture instructions

In this section you can find both the information about the syntax and purpose the assembly language instructions. If you need more technical information, look for the Intel Architecture Software Developer's Manual.

Assembly instructions consist of the mnemonic (instruction's name) and from zero to three operands. If there are two or more operands, usually first is the destination operand and second is the source operand. Each operand can be register, memory or immediate value (see 1.2 for details about syntax of operands). After description of each instruction there are provided examples of different combinations of operands (if the instruction has any).

2.1.1 Data movement instructions

`mov` transfers a byte, word or double word from the source operand to the destination operand. It can transfer data between general registers, from the general register to memory, or from memory to general register, but it cannot move from memory to memory. It can also transfer an immediate value to general register or memory, segment register to general register or memory, general register or memory to segment register, control or debug register to general register and general register to control or debug register. The `mov` can be assembled only if the size of source operand and size of

destination operand are the same. Below are the examples for each of the allowed combinations:

```
mov bx,ax       ; general register to general register
mov [char],al   ; general register to memory
mov bl,[char]   ; memory to general register
mov dl,32       ; immediate value to general register
mov [char],32   ; immediate value to memory
mov ax,ds       ; segment register to general register
mov [bx],ds     ; segment register to memory
mov ds,ax       ; general register to segment register
mov ds,[bx]     ; memory to segment register
mov eax,cr0     ; control register to general register
mov cr3,ebx     ; general register to control register
```

`xchg` swaps the contents of two operands. It can swap two byte operands, two word operands or two double word operands. Order of operands is not important. The operands may be two general registers, or general register with memory. For example:

```
xchg ax,bx      ; swap two general registers
xchg al,[char]  ; swap register with memory
```

`push` decrements the stack frame pointer (`esp` register), then transfers the operand to the top of stack indicated by `esp`. The operand can be memory, general register, segment register or immediate value of word or double word size. If operand is an immediate value and no size is specified, it is by default treated as a word value if assembler is in 16–bit mode and as a double word value if assembler is in 32–bit mode. If more operands follow in the same line (separated only with spaces, not commas), compiler will assemble chain of the `push` instructions with these operands. The examples are with single operands:

```
push ax         ; store general register
push es         ; store segment register
push [bx]       ; store memory
push 1000h      ; store immediate value
```

`pusha` saves the contents of the eight general register on the stack. This instruction has no operands. There are two version of this instruction, one 16–bit and one 32–bit, assembler automatically generates the right version for current mode, but it can be overridden by using `pushaw` or `pushad` mnemonic

to always get the 16–bit or 32–bit version. The 16–bit version of this instruction pushes general registers on the stack in the following order: `ax`, `cx`, `dx`, `bx`, the initial value of `sp` before `ax` was pushed, `bp`, `si` and `di`. The 32–bit version pushes equivalent 32–bit general registers in the same order.

`pop` transfers the word or double word at the current top of stack to the destination operand, and then increments `esp` to point to the new top of stack. The operand can be memory, general register or segment register. If more operands separated with spaces follow in the same line, compiler will assemble chain of the `pop` instructions with these operands.

```
pop bx          ; restore general register
pop ds          ; restore segment register
pop [si]        ; restore memory
```

`popa` restores the registers saved on the stack by `pusha` instruction, except for the saved value of `sp` (or `esp`), which is ignored. This instruction has no operands. To force assembling 16–bit or 32–bit version of this instruction use `popaw` or `popad` mnemonic.

2.1.2 Type conversion instructions

The type conversion instructions convert bytes into words, words into double words, and double words into quad words. These conversion can be done using the sign extension or zero extension. The sign extension fills the extra bits of the larger item with the value of the sign bit of the smaller item, the zero extension simply fills them with zeros.

`cwd` and `cdq` double the size of value `ax` or `eax` register respectively and store the extra bits into the `dx` or `edx` register. The conversion is done using the sign extension. These instructions have no operands.

`cbw` extends the sign of the byte in `al` throughout `ax`, and `cwde` extends the sign of the word in `ax` throughout `eax`. These instruction also have no operands.

`movsx` converts a byte to word or double word and a word to double word using the sign extension. `movzx` does the same, but it uses the zero extension. The source operand can be general register or memory, while the destination operand must be a general register. For example:

```
movsx ax,al         ; byte register to word register
movsx edx,dl        ; byte register to double word register
movsx eax,ax        ; word register to double word register
movsx ax,byte [bx]  ; byte memory to word register
movsx edx,byte [bx] ; byte memory to double word register
movsx eax,word [bx] ; word memory to double word register
```

2.1.3 Binary arithmetic instructions

`add` replaces the destination operand with the sum of the source and destination operands and sets CF if overflow has occurred. The operands may be bytes, words or double words. The destination operand can be general register or memory, the source operand can be general register or immediate value, it can also be memory if the destination operand is register.

```
add ax,bx       ; add register to register
add ax,[si]     ; add memory to register
add [di],al     ; add register to memory
add al,48       ; add immediate value to register
add [char],48   ; add immediate value to memory
```

`adc` sums the operands, adds one if CF is set, and replaces the destination operand with the result. Rules for the operands are the same as for the `add` instruction. An `add` followed by multiple `adc` instructions can be used to add numbers longer than 32 bits.

`inc` adds one to the operand. It does not affect CF. The operand can be general register or memory, size of operand can be byte, word or double word.

```
inc ax          ; increment register by one
inc byte [bx]   ; increment memory by one
```

`sub` subtracts the source operand from the destination operand and replaces the destination operand with the result. If a borrow is required, the CF is set. Rules for the operands are the same as for the `add` instruction.

`sbb` subtracts the source operand from the destination operand, subtracts one if CF is set, and stores the result to the destination operand. Rules for the operands are the same as for the `add` instruction. A `sub` followed by multiple `sbb` instructions may be used to subtract numbers longer than 32 bits.

`dec` subtracts one from the operand. It does not affect CF. Rules for the operand are the same as for the `inc` instruction.

`cmp` subtracts the source operand from the destination operand. It updates the flags as the `sub` instruction, but does not alter the source and destination operands. Rules for the operands are the same as for the `sub` instruction.

`neg` subtracts a signed integer operand from zero. The effect of this instructon is to reverse the sign of the operand from positive to negative or from negative to positive. Rules for the operand are the same as for the `inc` instruction.

`xadd` exchanges the destination operand with the source operand, then loads the sum of the two values into the destination operand. Rules for the operands are the same as for the `add` instruction.

All the above binary arithmetic instruction update SF, ZF, PF and OF flags. SF is always set to the same value as the sign bit of the result, ZF is set when all bits of result are zero, PF is set when low order eight bits of result contain an even number of set bits, OF is set if result is too large a positive number or too small a negative number (excluding sign bit) to fit in destination operand.

`mul` performs an unsigned multiplication of the operand and the accumulator. If the operand is a byte, the processor multiplies it by the contents of `al` and returns the 16–bit result to `ah` and `al`. If the operand is a word, the processor multiplies it by the contents of `ax` and returns the 32–bit result to `dx` and `ax`. If the operand is a double word, the processor multiplies it by the contents of `eax` and returns the 64–bit result in `edx` and `eax`. `mul` sets CF and OF when the upper half of the result is nonzero, otherwise they are cleared. Rules for the operand are the same as for the `inc` instruction.

`imul` performs a signed multiplication operation. This instruction has three variations. First has one operand and behaves in the same way as the `mul` instruction. Second has two operands, in this case destination operand is multiplied by the source operand and the result replaces the destination operand. Destination operand must be a general register, it can be word or double word, source operand can be general register, memory or immediate value. The immediate value can be a byte, in this case processor automatically does the sign extension to it before performing the multiplication. Third form has three operands, the destination operand must be a general register, word or double word in size, source operand can be general register or memory, and third operand must be an immediate value. The source operand is multiplied by the immediate value and the result is stored in the destination register. All the three forms calculate the product to twice the size of operands and set CF and OF when the upper half of the result is nonzero, but second and third form truncate the product to the size of operands. So second and third forms can be also used for unsigned operands because, whether the operands are signed or unsigned, the lower half of the product is the same. Below are the examples for all three forms:

```
imul bl          ; accumulator by register
imul word [si]   ; accumulator by memory
imul bx,cx       ; register by register
imul bx,[si]     ; register by memory
imul bx,10       ; register by immediate value
```

```
imul ax,bx,10   ; register by immediate value to register
imul ax,[si],10 ; memory by immediate value to register
```

`div` performs an unsigned division of the accumulator by the operand. The dividend (the accumulator) is twice the size of the divisor (the operand), the quotient and remainder have the same size as the divisor. If divisor is byte, the dividend is taken from `ax` register, the quotient is stored in `al` and the remainder is stored in `ah`. If divisor is word, the upper half of dividend is taken from `dx`, the lower half of dividend is taken from `ax`, the quotient is stored in `ax` and the remainder is stored in `dx`. If divisor is double word, the upper half of dividend is taken from `edx`, the lower half of dividend is taken from `eax`, the quotient is stored in `eax` and the remainder is stored in `edx`. Rules for the operand are the same as for the `mul` instruction.

`idiv` performs a signed division of the accumulator by the operand. It uses the same registers as the `div` instruction, and the rules for the operand are the same.

2.1.4 Decimal arithmetic instructions

Decimal arithmetic is performed by combining the binary arithmetic instructions (already described in the prior section) with the decimal arithmetic instructions. The decimal arithmetic instructions are used to adjust the results of a previous binary arithmetic operation to produce a valid packed or unpacked decimal result, or to adjust the inputs to a subsequent binary arithmetic operation so the operation will produce a valid packed or unpacked decimal result.

`daa` adjusts the result of adding two valid packed decimal operands in `al`. `daa` must always follow the addition of two pairs of packed decimal numbers (one digit in each half–byte) to obtain a pair of valid packed decimal digits as results. The carry flag is set if carry was needed. This instruction has no operands.

`das` adjusts the result of subtracting two valid packed decimal operands in `al`. `das` must always follow the subtraction of one pair of packed decimal numbers (one digit in each half–byte) from another to obtain a pair of valid packed decimal digits as results. The carry flag is set if a borrow was needed. This instruction has no operands.

`aaa` changes the contents of register `al` to a valid unpacked decimal number, and zeroes the top four bits. `aaa` must always follow the addition of two unpacked decimal operands in `al`. The carry flag is set and `ah` is incremented if a carry is necessary. This instruction has no operands.

`aas` changes the contents of register `al` to a valid unpacked decimal number, and zeroes the top four bits. `aas` must always follow the subtraction of one unpacked decimal operand from another in `al`. The carry flag is set and `ah` decremented if a borrow is necessary. This instruction has no operands.

`aam` corrects the result of a multiplication of two valid unpacked decimal numbers. `aam` must always follow the multiplication of two decimal numbers to produce a valid decimal result. The high order digit is left in `ah`, the low order digit in `al`. The generalized version of this instruction allows adjustment of the contents of the `ax` to create two unpacked digits of any number base. The standard version of this instruction has no operands, the generalized version has one operand – an immediate value specifying the number base for the created digits.

`aad` modifies the numerator in `ah` and `ah` to prepare for the division of two valid unpacked decimal operands so that the quotient produced by the division will be a valid unpacked decimal number. `ah` should contain the high order digit and `al` the low order digit. This instruction adjusts the value and places the result in `al`, while `ah` will contain zero. The generalized version of this instruction allows adjustment of two unpacked digits of any number base. Rules for the operand are the same as for the `aam` instruction.

2.1.5 Logical instructions

`not` inverts the bits in the specified operand to form a one's complement of the operand. It has no effect on the flags. Rules for the operand are the same as for the `inc` instruction.

`and`, `or` and `xor` instructions perform the standard logical operations. They update the SF, ZF and PF flags. Rules for the operands are the same as for the `add` instruction.

`bt`, `bts`, `btr` and `btc` instructions operate on a single bit which can be in memory or in a general register. The location of the bit is specified as an offset from the low order end of the operand. The value of the offset is the taken from the second operand, it either may be an immediate byte or a general register. These instructions first assign the value of the selected bit to CF. `bt` instruction does nothing more, `bts` sets the selected bit to 1, `btr` resets the selected bit to 0, `btc` changes the bit to its complement. The first operand can be word or double word.

```
bt  ax,15        ; test bit in register
bts word [bx],15 ; test and set bit in memory
btr ax,cx        ; test and reset bit in register
btc word [bx],cx ; test and complement bit in memory
```

`bsf` and `bsr` instructions scan a word or double word for first set bit and store the index of this bit into destination operand, which must be general register. The bit string being scanned is specified by source operand, it may be either general register or memory. The ZF flag is set if the entire string is zero (no set bits are found); otherwise it is set. If no set bit is found, the value of the destination register is undefined. `bsf` from low order to high order (starting from bit index zero). `bsr` scans from high order to low order (starting from bit index 15 of a word or index 31 of a double word).

```
bsf ax,bx         ; scan register forward
bsr ax,[si]       ; scan memory reverse
```

`shl` shifts the destination operand left by the number of bits specified in the second operand. The destination operand can be byte, word, or double word general register or memory. The second operand can be an immediate value or the `cl` register. The processor shifts zeros in from the right (low order) side of the operand as bits exit from the left side. The last bit that exited is stored in CF. `sal` is a synonym for `shl`.

```
shl al,1          ; shift register left by one bit
shl byte [bx],1   ; shift memory left by one bit
shl ax,cl         ; shift register left by count from cl
shl word [bx],cl  ; shift memory left by count from cl
```

`shr` and `sar` shift the destination operand right by the number of bits specified in the second operand. Rules for operands are the same as for the `shl` instruction. `shr` shifts zeros in from the left side of the operand as bits exit from the right side. The last bit that exited is stored in CF. `sar` preserves the sign of the operand by shifting in zeros on the left side if the value is positive or by shifting in ones if the value is negative.

`shld` shifts bits of the destination operand to the left by the number of bits specified in third operand, while shifting high order bits from the source operand into the destination operand on the right. The source operand remains unmodified. The destination operand can be a word or double word general register or memory, the source operand must be a general register, third operand can be an immediate value or the `cl` register.

```
shld ax,bx,1      ; shift register left by one bit
shld [di],bx,1    ; shift memory left by one bit
shld ax,bx,cl     ; shift register left by count from cl
shld [di],bx,cl   ; shift memory left by count from cl
```

`shrd` shifts bits of the destination operand to the right, while shifting low order bits from the source operand into the destination operand on the left. The source operand remains unmodified. Rules for operands are the same as for the `shld` instruction.

`rol` and `rcl` rotate the byte, word or double word destination operand left by the number of bits specified in the second operand. For each rotation specified, the high order bit that exits from the left of the operand returns at the right to become the new low order bit. `rcl` additionally puts in CF each high order bit that exits from the left side of the operand before it returns to the operand as the low order bit on the next rotation cycle. Rules for operands are the same as for the `shl` instruction.

`ror` and `rcr` rotate the byte, word or double word destination operand right by the number of bits specified in the second operand. For each rotation specified, the low order bit that exits from the right of the operand returns at the left to become the new high order bit. `rcr` additionally puts in CF each low order bit that exits from the right side of the operand before it returns to the operand as the high order bit on the next rotation cycle. Rules for operands are the same as for the `shl` instruction.

`test` performs the same action as the `and` instruction, but it does not alter the destination operand, only updates flags. Rules for the operands are the same as for the `and` instruction.

`bswap` reverses the byte order of a 32–bit general register: bits 0 through 7 are swapped with bits 24 through 31, and bits 8 through 15 are swapped with bits 16 through 23. This instruction is provided for converting little–endian values to big–endian format and vice versa.

```
bswap edx         ; swap bytes in register
```

2.1.6 Control transfer instructions

`jmp` unconditionally transfers control to the target location. The destination address can be specified directly within the instruction or indirectly through a register or memory, the acceptable size of this address depends on whether the jump is near or far (it can be specified by preceding the operand with `near` or `far` operator) and whether the instruction is 16–bit or 32–bit. Operand for near jump should be `word` size for 16–bit instruction or the `dword` size for 32–bit instruction. Operand for far jump should be `dword` size for 16–bit instruction or `pword` size for 32–bit instruction. A direct `jmp` instruction includes the destination address as part of the instruction, the operand specifying address should be the numerical expression for near jump, or two numerical expressions separated with colon for far jump, the first specifies

selector of segment, the second is the offset within segment. An indirect `jmp` instruction obtains the destination address indirectly through a register or a pointer variable, the operand should be general register or memory. See also 1.2.5 for more details.

```
jmp 100h          ; direct near jump
jmp 0FFFFh:0      ; direct far jump
jmp ax            ; indirect near jump
jmp pword [ebx]   ; indirect far jump
```

`call` transfers control to the procedure, saving on the stack the address of the instruction following the `call` for later use by a `ret` (return) instruction. Rules for the operands are the same as for the `jmp` instruction, but the `call` has no short variant of direct instruction and thus it not optimized.

`ret`, `retn` and `retf` instructions terminate the execution of a procedure and transfers control back to the program that originally invoked the procedure using the address that was stored on the stack by the `call` instruction. `ret` is the equivalent for `retn`, which returns from the procedure that was executed using the near call, while `retf` return from the procedure that was executed using the far call. These instructions default to the size of address appropriate for the current code setting, but the size of address can be forced to 16–bit by using the `retw`, `retnw` and `retfw` mnemonics, and to 32–bit by using the `retd`, `retnd` and `retfd` mnemonics. All these instructions may optionally specify an immediate operand, by adding this constant to the stack pointer, they effectively remove any arguments that the calling program pushed on the stack before the execution of the `call` instruction.

`iret` returns control to an interrupted procedure. It differs from `ret` in that it also pops the flags from the stack into the flags register. The flags are stored on the stack by the interrupt mechanism. It defaults to the size of return address appropriate for the current code setting, but it can be forced to use 16–bit or 32–bit address by using the `iretw` or `iretd` mnemonic.

The conditional transfer instructions are jumps that may or may not transfer control, depending on the state of the CPU flags when the instruction executes. The mnemonics for conditional jumps may be obtained by attaching the condition mnemonic (see table 2.1) to the `j` mnemonic, for example `jc` instruction will transfer the control when the CF flag is set. The conditional jumps can be near and direct only, and can be optimized (see 1.2.5), the operand should be an immediate value specifying target address.

The `loop` instructions are conditional jumps that use a value placed in `cx` (or `ecx`) to specify the number of repetitions of a software loop. All `loop` instructions automatically decrement `cx` (or `ecx`) and terminate the loop (don't transfer the control) when `cx` (or `ecx`) is zero. It uses `cx` or `ecx`

Mnemonic	Condition tested	Description
o	OF = 1	overflow
no	OF = 0	not overflow
c b nae	CF = 1	carry below not above nor equal
nc ae nb	CF = 0	not carry above or equal not below
e z	ZF = 1	equal zero
ne nz	ZF = 0	not equal not zero
be na	CF or ZF = 1	jump if below or equal not above
a nbe	CF or ZF = 0	above not below nor equal
s	SF = 1	sign
ns	SF = 0	not sign
p pe	PF = 1	parity parity even
np po	PF = 0	not parity parity odd
l nge	SF xor OF = 1	less not greater nor equal
ge nl	SF xor OF = 0	greater or equal not less
le ng	(SF xor OF) or ZF = 1	less or equal not greater
g nle	(SF xor OF) or ZF = 0	greater not less nor equal

Table 2.1: Conditions.

whether the current code setting is 16–bit or 32–bit, but it can be forced to use `cx` with the `loopw` mnemonic or to use `ecx` with the `loopd` mnemonic. `loope` and `loopz` are the synonyms for the same instruction, which acts as the standard `loop`, but also terminates the loop when ZF flag is set. `loopew` and `loopzw` mnemonics force them to use `cx` register while `looped` and `loopzd` force them to use `ecx` register. `loopne` and `loopnz` are the synonyms for the same instructions, which acts as the standard `loop`, but also terminate the loop when ZF flag is not set. `loopnew` and `loopnzw` mnemonics force them to use `cx` register while `loopned` and `loopnzd` force them to use `ecx` register. Every `loop` instruction needs an operand being an immediate value specifying target address, it can be only short jump (in the range of 128 bytes back and 127 bytes forward from the address of instruction following the `loop` instruction).

`jcxz` branches to the label specified in the instruction if it finds a value of zero in `cx`, `jecxz` does the same, but checks the value of `ecx` instead of `cx`. Rules for the operands are the same as for the `loop` instruction.

`int` activates the interrupt service routine that corresponds to the number specified as an operand to the instruction, the number should be in range from 0 to 255. The interrupt service routine terminates with an `iret` instruction that returns control to the instruction that follows `int`. `int3` mnemonic codes the short (one byte) trap that invokes the interrupt 3. `into` instruction invokes the interrupt 4 if the OF flag is set.

`bound` verifies that the signed value contained in the specified register lies within specified limits. An interrupt 5 occurs if the value contained in the register is less than the lower bound or greater than the upper bound. It needs two operands, the first operand specifies the register being tested, the second operand should be memory address for the two signed limit values. The operands can be `word` or `dword` in size.

```
bound ax,[bx]     ; check word for bounds
bound eax,[esi]   ; check double word for bounds
```

2.1.7 I/O instructions

`in` transfers a byte, word, or double word from an input port to `al`, `ax`, or `eax`. I/O ports can be addressed either directly, with the immediate byte value coded in instruction, or indirectly via the `dx` register. The destination operand should be `al`, `ax`, or `eax` register. The source operand should be an immediate value in range from 0 to 255, or `dx` register.

```
in al,20h         ; input byte from port 20h
in ax,dx          ; input word from port addressed by dx
```

`out` transfers a byte, word, or double word to an output port from `al`, `ax`, or `eax`. The program can specify the number of the port using the same methods as the `in` instruction. The destination operand should be an immediate value in range from 0 to 255, or `dx` register. The source operand should be `al`, `ax`, or `eax` register.

```
out 20h,ax        ; output word to port 20h
out dx,al         ; output byte to port addressed by dx
```

2.1.8 Strings operations

The string operations operate on one element of a string. A string element may be a byte, a word, or a double word. The string elements are addressed by `si` and `di` (or `esi` and `edi`) registers. After every string operation `si` and/or `di` (or `esi` and/or `edi`) are automatically updated to point to the next element of the string. If DF (direction flag) is zero, the index registers are incremented, if DF is one, they are decremented. The amount of the increment or decrement is 1, 2, or 4 depending on the size of the string element. Every string operation instruction has short forms which has no operands and use `si` and/or `di` when the code type is 16–bit, and `esi` and/or `edi` when the code type is 32–bit. `si` and `esi` by default addresses data in the segment selected by `ds`, `di` and `edi` always addresses data in the segment selected by `es`. Short form is obtained by attaching to the mnemonic of string operation letter specifying the size of string element, it should be `b` for byte element, `w` for word element, and `d` for double word element. Full form of string operation needs operands providing the size operator and the memory addresses, which can be `si` or `esi` with any segment prefix, `di` or `edi` always with `es` segment prefix.

`movs` transfers the string element pointed to by `si` (or `esi`) to the location pointed to by `di` (or `edi`). Size of operands can be `byte`, `word` or `dword`. The destination operand should be memory addressed by `di` or `edi`, the source operand should be memory addressed by `si` or `esi` with any segment prefix.

```
movs byte [di],[si]         ; transfer byte
movs word [es:di],[ss:si]   ; transfer word
movsd                       ; transfer double word
```

`cmps` subtracts the destination string element from the source string element and updates the flags AF, SF, PF, CF and OF, but it does not change any of the compared elements. If the string elements are equal, ZF is set, otherwise it is cleared. The first operand for this instruction should be the source string element addressed by `si` or `esi` with any segment prefix, the

second operand should be the destination string element addressed by `di` or `edi`.

```
cmpsb                       ; compare bytes
cmps word [ds:si],[es:di]   ; compare words
cmps dword [fs:esi],[edi]   ; compare double words
```

`scas` subtracts the destination string element from `al`, `ax`, or `eax` (depending on the size of string element) and updates the flags AF, SF, ZF, PF, CF and OF. If the values are equal, ZF is set, otherwise it is cleared. The operand should be the destination string element addressed by `di` or `edi`.

```
scas byte [es:di]           ; scan byte
scasw                       ; scan word
scas dword [es:edi]         ; scan double word
```

`lods` places the source string element into `al`, `ax`, or `eax`. The operand should be the source string element addressed by `si` or `esi` with any segment prefix.

```
lods byte [ds:si]           ; load byte
lods word [cs:si]           ; load word
lodsd                       ; load double word
```

`stos` places the value of `al`, `ax`, or `eax` into the destination string element. Rules for the operand are the same as for the `scas` instruction.

`ins` transfers a byte, word, or double word from an input port addressed by `dx` register to the destination string element. The destination operand should be memory addressed by `di` or `edi`, the source operand should be the `dx` register.

```
insb                        ; input byte
ins word [es:di],dx         ; input word
ins dword [edi],dx          ; input double word
```

`outs` transfers the source string element to an output port addressed by `dx` register. The destination operand should be the `dx` register and the source operand should be memory addressed by `si` or `esi` with any segment prefix.

```
outs dx,byte [si]           ; output byte
outsw                       ; output word
outs dx,dword [gs:esi]      ; output double word
```

The repeat prefixes `rep`, `repe/repz`, and `repne/repnz` specify repeated string operation. When a string operation instruction has a repeat prefix, the operation is executed repeatedly, each time using a different element of the string. The repetition terminates when one of the conditions specified by the prefix is satisfied. All three prefixes automatically decrease `cx` or `ecx` register (depending whether string operation instruction uses the 16–bit or 32–bit addressing) after each operation and repeat the associated operation until `cx` or `ecx` is zero. `repe/repz` and `repne/repnz` are used exclusively with the `scas` and `cmps` instructions (described below). When these prefixes are used, repetition of the next instruction depends on the zero flag (ZF) also, `repe` and `repz` terminate the execution when the ZF is zero, `repne` and `repnz` terminate the execution when the ZF is set.

```
rep  movsd       ; transfer multiple double words
repe cmpsb       ; compare bytes until not equal
```

2.1.9 Flag control instructions

The flag control instructions provide a method for directly changing the state of bits in the flag register. All instructions described in this section have no operands.

`stc` sets the CF (carry flag) to 1, `clc` zeroes the CF, `cmc` changes the CF to its complement. `std` sets the DF (direction flag) to 1, `cld` zeroes the DF, `sti` sets the IF (interrupt flag) to 1 and therefore enables the interrupts, `cli` zeroes the IF and therefore disables the interrupts.

`lahf` copies SF, ZF, AF, PF, and CF to bits 7, 6, 4, 2, and 0 of the `ah` register. The contents of the remaining bits are undefined. The flags remain unaffected.

`sahf` transfers bits 7, 6, 4, 2, and 0 from the `ah` register into SF, ZF, AF, PF, and CF.

`pushf` decrements `esp` by two or four and stores the low word or double word of flags register at the top of stack, size of stored data depends on the current code setting. `pushfw` variant forces storing the word and `pushfd` forces storing the double word.

`popf` transfers specific bits from the word or double word at the top of stack, then increments `esp` by two or four, this value depends on the current code setting. `popfw` variant forces restoring from the word and `popfd` forces restoring from the double word.

2.1.10 Conditional operations

The instructions obtained by attaching the condition mnemonic (see table 2.1) to the `set` mnemonic set a byte to one if the condition is true and set the byte to zero otherwise. The operand should be an 8–bit be general register or the byte in memory.

```
setne al          ; set al if zero flag cleared
seto byte [bx]    ; set byte if overflow
```

`setalc` instruction sets the all bits of `al` register when the carry flag is set and zeroes the `al` register otherwise. This instruction has no arguments.

The instructions obtained by attaching the condition mnemonic to the `cmov` mnemonic transfer the word or double word from the general register or memory to the general register only when the condition is true. The destination operand should be general register, the source operand can be general register or memory.

```
cmove ax,bx       ; move when zero flag set
cmovnc eax,[ebx]  ; move when carry flag cleared
```

`cmpxchg` compares the value in the `al`, `ax`, or verb+eax+ register with the destination operand. If the two values are equal, the source operand is loaded into the destination operand. Otherwise, the destination operand is loaded into the `al`, `ax`, or `eax` register. The destination operand may a general register or memory, the source operand must be a general register.

```
cmpxchg dl,bl     ; compare and exchange with register
cmpxchg [bx],dx   ; compare and exchange with memory
```

`cmpxchg8b` compares the 64–bit value in `edx` and `eax` registers with the destination operand. If the values are equal, the 64–bit value in `ecx` and `ebx` registers is stored in the destination operand. Otherwise, the value in the destination operand is loaded into `edx` and `eax` registers. The destination operand should be a quad word in memory.

```
cmpxchg8b [bx]    ; compare and exchange 8 bytes
```

2.1.11 Miscellaneous instructions

`nop` instruction occupies one byte but affects nothing but the instruction pointer. This instruction has no operands and doesn't perform any operation.

`ud2` instruction generates an invalid opcode exception. This instruction is provided for software testing to explicitly generate an invalid opcode. This is instruction has no operands.

`xlat` replaces a byte in the `al` register with a byte indexed by its value in a translation table addressed by `bx` or `ebx`. The operand should be a byte memory addressed by `bx` or `ebx` with any segment prefix. This instruction has also a short form `xlatb` which has no operands and uses the `bx` or `ebx` address in the segment selected by `ds` depending on the current code setting.

`lds` transfers a pointer variable from the source operand to `ds` and the destination register. The source operand must be a memory operand, and the destination operand must be a general register. The `ds` register receives the segment selector of the pointer while the destination register receives the offset part of the pointer. `les`, `lfs`, `lgs` and `lss` operate identically to `lds` except that rather than `ds` register the `es`, `fs`, `gs` and `ss` is used respectively.

```
lds bx,[si]       ; load pointer to ds:bx
```

`lea` transfers the offset of the source operand (rather than its value) to the destination operand. The source operand must be a memory operand, and the destination operand must be a general register.

```
lea dx,[bx+si+1] ; load effective address to dx
```

`cpuid` returns processor identification and feature information in the `eax`, `ebx`, `ecx`, and `edx` registers. The information returned is selected by entering a value in the `eax` register before the instruction is executed. This instruction has no operands.

`enter` creates a stack frame that may be used to implement the scope rules of block–structured high–level languages. A `leave` instruction at the end of a procedure complements an `enter` at the beginning of the procedure to simplify stack management and to control access to variables for nested procedures. The `enter` instruction includes two parameters. The first parameter specifies the number of bytes of dynamic storage to be allocated on the stack for the routine being entered. The second parameter corresponds to the lexical nesting level of the routine, it can be in range from 0 to 31. The specified lexical level determines how many sets of stack frame pointers the CPU copies into the new stack frame from the preceding frame. This list of stack frame pointers is sometimes called the display. The first word (or double word when code is 32–bit) of the display is a pointer to the last stack frame. This pointer enables a `leave` instruction to reverse the action of the previous `enter` instruction by effectively discarding the last stack frame. After `enter` creates the new display for a procedure, it allocates the dynamic storage space for that procedure by decrementing `esp` by the number of bytes specified in the first parameter. To enable a procedure to address its display, `enter` leaves `bp` (or `ebp`) pointing to the beginning of the new stack frame. If

the lexical level is zero, `enter` pushes `bp` (or `ebp`), copies `sp` to `bp` (or `esp` to `ebp`) and then subtracts the first operand from `esp`. For nesting levels greater than zero, the processor pushes additional frame pointers on the stack before adjusting the stack pointer.

```
enter 2048,0      ; enter and allocate 2048 bytes on stack
```

2.1.12 System instructions

`lmsw` loads the operand into the machine status word (bits 0 through 15 of `cr0` register), while `smsw` stores the machine status word into the destination operand. The operand should be a word, it can be a general register or memory.

```
lmsw ax           ; load machine status from register
smsw [bx]         ; store machine status to memory
```

`lgdt` and `lidt` instructions load the values in operand into the global descriptor table register or the interrupt descriptor table register respectively. `sgdt` and `sidt` store the contents of the global descriptor table register or the interrupt descriptor table register in the destination operand. The operand should be a 6 bytes in memory.

```
lgdt [ebx]        ; load global descriptor table
```

`lldt` loads the operand into the segment selector field of the local descriptor table register and `sldt` stores the segment selector from the local descriptor table register in the operand. `ltr` loads the operand into the segment selector field of the task register and `str` stores the segment selector from the task register in the operand. Rules for operand are the same as for the `lmsw` instruction.

`lar` loads the access rights from the segment descriptor specified by the selector in source operand into the destination operand and sets the ZF flag. The operands can be both words or double words. The source operand may be a general register or memory. The destination operand should be a general register.

```
lar ax,[bx]       ; load access rights into word
lar eax,edx       ; load access rights into double word
```

`lsl` loads the segment limit from the segment descriptor specified by the selector in source operand into the destination operand and sets the ZF flag. Rules for operand are the same as for the `lar` instruction.

`verr` and `verw` verify whether the code or data segment specified with the operand is readable or writable from the current privilege level. The operand should be a word, it can be general register or memory. If the segment is accessible and readable (for `verr`) or writable (for `verw`) the ZF flag is set, otherwise it's cleared. Rules for operand are the same as for the `lldt` instruction.

`arpl` compares the RPL (requestor's privilege level) fields of two segment selectors. The first operand contains one segment selector and the second operand contains the other. If the RPL field of the destination operand is less than the RPL field of the source operand, the ZF flag is set and the RPL field of the destination operand is increased to match that of the source operand. Otherwise, the ZF flag is cleared and no change is made to the destination operand. The destination operand can be a word general register or memory, the source operand must be a general register.

```
arpl bx,ax        ; adjust RPL of selector in register
arpl [bx],ax      ; adjust RPL of selector in memory
```

`clts` clears the TS (task switched) flag in the `cr0` register. This instruction has no operands.

`lock` prefix causes the processors bus–lock signal to be asserted during execution of the accompanying instruction. In a multiprocessor environment, the bus–lock signal insures that the processor has exclusive use of any shared memory while the signal is asserted. The `lock` prefix can be prepended only to the following instructions and only to those forms of the instructions where the destination operand is a memory operand: `add`, `adc`, `and`, `btc`, `btr`, `bts`, `cmpxchg`, `cmpxchg8b`, `dec`, `inc`, `neg`, `not`, `or`, `sbb`, `sub`, `xor`, `xadd` and `xchg`. If the `lock` prefix is used with one of these instructions and the source operand is a memory operand, an undefined opcode exception may be generated. An undefined opcode exception will also be generated if the `lock` prefix is used with any instruction not in the above list. The `xchg` instruction always asserts the bus–lock signal regardless of the presence or absence of the `lock` prefix.

`hlt` stops instruction execution and places the processor in a halted state. An enabled interrupt, a debug exception, the BINIT, INIT or the RESET signal will resume execution. This instruction has no operands.

`invlpg` invalidates (flushes) the TLB (translation lookaside buffer) entry specified with the operand, which should be a memory. The processor determines the page that contains that address and flushes the TLB entry for that page.

`rdmsr` loads the contents of a 64–bit MSR (model specific register) of the address specified in the `ecx` register into registers `edx` and `eax`. `wrmsr` writes

the contents of registers `edx` and `eax` into the 64–bit MSR of the address specified in the `ecx` register. `rdtsc` loads the current value of the processors time stamp counter from the 64–bit MSR into the `edx` and `eax` registers. The processor increments the time stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. `rdpmc` loads the contents of the 40–bit performance monitoring counter specified in the `ecx` register into registers `edx` and `eax`. These instructions have no operands.

`wbinvd` writes back all modified cache lines in the processors internal cache to main memory and invalidates (flushes) the internal caches. The instruction then issues a special function bus cycle that directs external caches to also write back modified data and another bus cycle to indicate that the external caches should be invalidated. This instruction has no operands.

2.1.13 FPU instructions

[...]

2.1.14 MMX instructions

[...]

2.1.15 SSE instructions

[...]

2.2 Control directives

This section describes the directives that control the assembly process, they are processed during the assembly and may cause some blocks of instructions to be assembled differently or not assembled.

2.2.1 Repeating blocks of instructions

`times` directive repeats one instruction specified number of times. It should be followed by numerical expression specifying number of repeats and the instruction to repeat (optionally colon can be used to separate number and instruction). When special symbol `%` is used inside the instruction, it is equal to the number of current repeat. For example `times 5 db %` will define five bytes with values 1, 2, 3, 4, 5. Recursive use of `times` directive is also allowed, so `times 3 times % db %` will define six bytes with values 1, 1, 2, 1, 2, 3.

`repeat` directive repeats the whole block of instructions. It should be followed by numerical expression specifying number of repeats. Instructions to repeat are expected in next lines, ended with the `end repeat` directive, for example:

```
repeat 8
    mov byte [bx],%
    inc bx
end repeat
```

The generated code will store byte values from one to eight in the memory addressed by `bx` register.

Number of repeats can be zero, in that case the instructions are not assembled at all.

2.2.2 Conditional assembly

`if` directive causes come block of instructions to be assembled only under certain condition. It should be followed by logical expression specifying the condition, instructions in next lines will be assembled only when this condition is met, otherwise they will be skipped. The optional `else if` directive followed with logical expression specifying additional condition begins the next block of instructions that will be assembled if previous conditions were not met, and the additional condition is met. The optional `else` directive begins the block of instructions that will be assembled if all the conditions were not met. The `end if` directive ends the last block of instructions.

The logical expression consist of logical values and logical operators. The logical operators are `~` for logical negation, `&` for logical and, `|` for logical or. Logical value can be a numerical expression, it will be false if it is equal to zero, otherwise it will be true. Two numerical expression can be compared using one of the following operators to make the logical value: `=` (equal), `<` (less), `>` (greater), `<=` (less or equal), `>=` (greater or equal), `<>` (not equal). The `eq` compares any two symbols whether they are exact the same. The `in` operator checks whether given symbol is a member of the list of symbols following this operator, the list should be enclosed between `<` and `>` characters, its members should be separated with commas.

The following simple example uses the `count` constant that should be defined somewhere in source:

```
if count>0
    mov cx,count
    rep movsb
end if
```

These two assembly instructions will be assembled only if the `count` constant is greater than 0.

The next example is more complex and assumes that the symbolic constant `reg` is defined:

```
if reg in <cs,ds,es,fs,gs,ss>
    mov dx,reg
    add ax,dx
    shl ax,1
else if reg eq ax
    shl ax,2
else
    add ax,reg
    shl ax,1
end if
```

The first block of instructions will be assembled only if the value of `reg` is segment register, otherwise the second or third block will assembled whether the value of `reg` is `ax` register or not.

2.2.3 Other directives

`virtual` defines virtual data at specified address. This data won't be included in the output file, but labels defined there can be used in other parts of source. This directive can be followed by `at` operator and the numerical expression specifying the address for virtual data, otherwise is uses current address, the same as `virtual at $`. Instructions defining data are expected in next lines, ended with `end virtual` directive.

This directive can be used to create union of some variables, for example:

```
GDTR dp ?
virtual at GDTR
    GDT_limit dw ?
    GDT_address dd ?
end virtual
```

It defines two labels for parts of the 48–bit variable at `GDTR` address.

It can be also used to define labels for some structures addressed by a register, for example:

```
virtual at bx
    LDT_limit dw ?
    LDT_address dd ?
end virtual
```

With such definition instruction `mov ax,[LDT_limit]` will be assembled to `mov ax,[bx]`.

Declaring defined data values or instructions inside the virtual block would also be useful, because the `load` directive can be used to load the values from the virtually generated code into a constants. This directive should be used after the code it loads but before the virtual block ends, because it can only load the values from the same code space. For example:

```
virtual at 0
    xor eax,eax
    and edx,eax
    load zeroq dword from 0
end virtual
```

The above piece of code will define the `zeroq` constant containing four bytes of the machine code of the instructions defined inside the virtual block.

`display` directive displays the message at the assembly time. It should be followed by the quoted strings or byte values, separated with commas. It can be used to display values of some constants, for example:

```
d1 = '0'+ $ shr 12 and 0Fh
d2 = '0'+ $ shr 8 and 0Fh
d3 = '0'+ $ shr 4 and 0Fh
d4 = '0'+ $ and 0Fh
if d1>'9'
    d1 = d1 + 'A'-'9'-1
end if
if d2>'9'
    d2 = d2 + 'A'-'9'-1
end if
if d3>'9'
    d3 = d3 + 'A'-'9'-1
end if
if d4>'9'
    d4 = d4 + 'A'-'9'-1
end if
display 'Current offset is 0x',d1,d2,d3,d4,13,10
```

Instructions before the `display` directive calculate four digits of 16–bit value and convert them into characters for displaying.

2.3 Preprocessor directives

All preprocessor directives are processed before the main assembly process, and therefore are not affected by the control directives. At this time also all comments are stripped out.

2.3.1 Including source files

`include` directive includes the specified source file at the position where it is used. It should be followed by the quoted name of file that should be included, for example:

```
include 'macros.inc'
```

The whole included file is preprocessed before preprocessing the lines next to the line containing the `include` directive. There are no limits to the number of included files as long as they fit in memory.

The quoted path can contain environment variables enclosed within `%` characters, they will be replaced with their values inside the path, both the `\` and `/` characters are allowed as a path separators. It concerns also paths given with the `file` and `load` directives or in the command line.

2.3.2 Symbolic constants

The symbolic constants are different from the numerical constants, before the assembly process they are replaced with their values everywhere in source lines after their definitions, and anything can become their values.

The definition of symbolic constant consists of name of the constant followed by the `equ` directive. Everything that follows this directive will become the value of constant. If the value of symbolic constant contains other symbolic constants, they are replaced with their values before assigning this value to the new constant. For example:

```
d equ dword
NULL equ d 0
d equ edx
```

After there three definitions the value of `NULL` constant is `dword 0` and the value of `d` is `edx`. So, for example, `push NULL` will be assembled as `push dword 0` and `push d` will be assembled as `push edx`.

`restore` directive allows to get back previous value of redefined symbolic constant. It should be followed by one more names of symbolic constants, separated with commas. So `restore d` after the above definitions will give

`d` constant back the value `dword`. If there was no constant defined of given name, `restore` won't cause an error, it will be just ignored.

Symbolic constant can be used to adjust the syntax of assembler to personal preferences. For example the following set of definitions provides the handy shortcuts for all the size operators:

```
b equ byte
w equ word
d equ dword
p equ pword
f equ fword
q equ qword
t equ tword
x equ dqword
```

Because symbolic constant may also have an empty value, it can be used to allow the syntax with `offset` word before any address value:

```
offset equ
```

After this definition `mov ax,offset char` will be valid construction for copying the offset of `char` variable into `ax` register, because `offset` is replaced with an empty value, and therefore ignored.

2.3.3 Macroinstructions

`macro` directive allows you to define your own complex instructions, called macroinstructions, using which can greatly simplify the process of programming. In its simplest form it's similar to symbolic constant definition. For example the following definition defines a shortcut for the `test al,0xFF` instruction:

```
macro tst {test al,0xFF}
```

After the `macro` directive there is a name of macroinstruction and then its contents enclosed between the `{` and `}` characters. You can use `tst` instruction anywhere after this definition and it will be assembled as `test al,0xFF`. Defining symbolic constant `tst` of that value would give the similar result, but the difference is that the name of macroinstruction is recognized only as an instruction mnemonic. Also, macroinstructions are replaced with corresponding code even before the symbolic constants are replaced with their values. So if you define macroinstruction and symbolic constant of the same name, and use this name as an instruction mnemonic, it will be replaced

with the contents of macroinstruction, but it will be replaced with value if symbolic constant if used somewhere inside the operands.

The definition of macroinstruction can consist of many lines, because `{` and `}` characters don't have to be in the same line as `macro` directive. For example:

```
macro stos0
 {
    xor al,al
    stosb
 }
```

The macroinstruction `stos0` will be replaced with these two assembly instructions anywhere it's used.

Like instructions which needs some number of operands, the macroinstruction can be defined to need some number of arguments separated with commas. The names of needed argument should follow the name of macroinstruction in the line of `macro` directive and should be separated with commas if there is more than one. Anywhere one of these names occurs in the contents of macroinstruction, it will be replaced with corresponding value, provided when the macroinstruction is used. Here is an example of a macroinstruction that will do data alignment for binary output format:

```
macro align value { rb (value-1)-($+value-1) mod value }
```

When the `align 4` instruction is found after this macroinstruction is defined, it will be replaced with contents of this macroinstruction, and the `value` will there become 4, so the result will be `rb (4-1)-($+4-1) mod 4`.

If a macroinstruction is defined that uses an instruction with the same name inside its definition, the previous meaning of this name is used. Useful redefinition of macroinstructions can be done in that way, for example:

```
macro mov op1,op2
 {
  if op1 in <ds,es,fs,gs,ss> & op2 in <cs,ds,es,fs,gs,ss>
    push  op2
    pop   op1
  else
    mov   op1,op2
  end if
 }
```

This macroinstruction extends the syntax of `mov` instruction, allowing both operands to be segment registers. For example `mov ds,es` will be assembled as `push es` and `pop ds`. In all other cases the standard `mov` instruction will be used. The syntax of this `mov` can be extended further by defining next macroinstruction of that name, which will use the previous macroinstruction:

```
macro mov op1,op2,op3
 {
  if arg3 eq
    mov   op1,op2
  else
    mov   op1,op2
    mov   op2,op3
  end if
 }
```

It allows `mov` instruction to have three operands, but it can still have two operands only, because when macroinstruction is given less arguments than it needs, the rest of arguments will have empty values. When three operands are given, this macroinstruction will become two macroinstructions of the previous definition, so `mov es,ds,dx` will be assembled as `push ds`, `pop es` and `mov ds,dx`.

`purge` directive allows removing the last definition of specified macroinstruction. It should be followed by one or more names of macroinstructions, separated with commas. If such macroinstruction has not been defined, you won't get any error. For example after having the syntax of `mov` extended with the macroinstructions defined above, you can disable syntax with three operands back by using `purge mov` directive. Next `purge mov` will disable also syntax for two operands being segment registers, and all the next such directives will do nothing.

If after the `macro` directive you enclose some group of arguments' names in square brackets, it will allow giving more values for this group of arguments when using that macroinstruction. Any more argument given after the last argument of such group will begin the new group and will become the first argument of it. That's why after closing the square bracket no more argument names can follow. The contents of macroinstruction will be processed for each such group of arguments separately. The simplest example is to enclose one argument name in square brackets:

```
macro stoschar [char]
 {
    mov al,char
```

```
    stosb
}
```

This macroinstruction accepts unlimited number of arguments, and each one will be processed into these two instructions separately. For example `stoschar 1,2,3` will be assembled as the following instructions:

```
mov al,1
stosb
mov al,2
stosb
mov al,3
stosb
```

There are some special directives available only inside the definitions of macroinstructions. `local` directive defines local names, which will be replaced with unique values each time the macroinstruction is used. It should be followed by names separated with commas. This directive is usually needed for the constants or labels that macroinstruction defines and uses internally. For example:

```
macro movstr
 {
    local move
  move:
    lodsb
    stosb
    test al,al
    jnz move
 }
```

Each time this macroinstruction is used, `move` will become other unique name in its instructions, so you won't get an error you normally get when some label is defined more than once.

`forward`, `reverse` and `common` directives divide macroinstruction into blocks, each one processed after the processing of previous is finished. They differ in behavior only if macroinstruction allows multiple groups of arguments. Block of instructions that follows `forward` directive will be processed for each group of arguments, from first to last – exactly like the default block (not preceded by any of these directives). Block that follows `reverse` directive will be processed for each group of argument in reverse order – from last to first. Block that follows `common` directive is processed only once, commonly for all groups of arguments. Local name defined in one of the blocks

is available in all the following blocks when processing the same group of arguments as when it was defined, and when it is defined in common block it is available in all the following blocks not depending on which group of arguments is processed.

Here is an example of macroinstruction that will create the table of addresses to strings followed by these strings:

```
macro strtbl name,[string]
 {
  common
    label name dword
  forward
    local label
    dd label
  forward
    label db string,0
 }
```

First argument given to this macroinstruction will become the label for table of addresses, next arguments should be the strings. First block is processed only once and defines the label, second block for each string declares its local name and defines the table entry holding the address to that string. Third block defines the data of each string with the corresponding label.

The directive starting the block in macroinstruction can be followed by the first instruction of this block in the same line, like in the following example:

```
macro stdcall proc,[arg]
 {
  reverse push arg
  common call proc
 }
```

This macroinstruction can be used for calling the procedures using STDCALL convention, arguments are pushed on stack in the reverse order. For example `stdcall foo,1,2,3` will be assembled as:

```
push 3
push 2
push 1
call foo
```

If some name inside macroinstruction has multiple values (it is either one of the arguments enclosed in square brackets or local name defined in the block following `forward` or `reverse` directive) and is used in block following the `common` directive, it will be replaced with all of its values, separated with commas. For example the following macroinstruction will pass all of the additional arguments to the previously defined `stdcall` macroinstruction:

```
macro invoke proc,[arg]
 { common stdcall [proc],arg }
```

It can be used to call indirectly (by the pointer stored in memory) the procedure using STDCALL convention.

Inside macroinstruction also special operator `#` can be used. This operator causes two names to be concatenated into one name. It can be useful, because it's done after the arguments and local names are replaced with their values. The following macroinstruction will generate the conditional jump according to the `cond` argument:

```
macro jif op1,cond,op2,label
 {
    cmp op1,op2
    j#cond label
 }
```

For example `jif ax,ae,10h,exit` will be assembled as `cmp ax,10h` and `jae exit` instructions.

To make macroinstruction behaving differently when some of the arguments are a quoted strings or not, you can utilize the fact that assembler distinguishes the separate quoted strings from the quoted strings inside numerical expressions, but it doesn't distinguish the numerical expression preceded by the `+` sign from the same expression without a sign. So the string preceded by the `+` sign will be treated as a numerical expression and so won't be symbolically equal to the same string without any sign, while any other value will be symbolically equal to the same expression preceded by the `+` sign. Here's an example macroinstruction utilizing this feature:

```
macro message arg
 {
  if arg eq +arg
    mov   dx,arg
  else
    local str
    jmp   @f
```

```
    str   db arg,0Dh,0Ah,24h
    @@:
    mov   dx,str
  end if
    mov   ah,9
    int   21h
}
```

The above macro is designed for displaying messages in DOS programs. When the argument of this macro is some number of label, the string from at address is displayed, but when the argument is a quoted string, the created code will display that string followed by the carriage return and line feed.

2.3.4 Structures

`struc` directive is a special variant of `macro` directive that is used to define data structures. Macroinstruction defined using the `struc` directive must be preceded by a label (like the data definition directive) when it's used. This label will be also attached at the beginning of every name starting with dot in the contents of macroinstruction. The macroinstruction defined using the `struc` directive can have the same name as some other macroinstruction defined using the `macro` directive, structure macroinstruction won't prevent the standard macroinstruction being processed when there is no label before it and vice versa. All the rules concerning standard macroinstructions apply to structure macroinstructions.

Here is the sample of structure macroinstruction:

```
struc point x,y
 {
    .x dw x
    .y dw y
 }
```

For example `my point 7,11` will define structure labelled `my`, consisting of two variables: `my.x` with value 7 and `my.y` with value 11.

Next example shows how to extend the data definition directive `db` with ability to calculate the size of defined data by using the structure macroinstruction:

```
struc db [data]
 {
  common
```

```
        label .data byte
        db data
        .size = $-.data
    }
```

With such definition for example `msg db 'Hello!',13,10` will define also `msg.size` constant, equal to the size of defined data in bytes and also additional label `msg.data`, which will be recognized as a label for data of byte size.

Defining data structures addressed by registers or absolute values should be done using the `virtual` directive with structure macroinstruction (see 2.2.3).

2.4 Formatter directives

`format` directive followed by the format identifier allows to select the output format. This directive should be put at the beginning of the source. Default output format is a flat binary file, it can also be selected by using `format binary` directive.

`use16` and `use32` directives force the assembler to generate 16–bit or 32–bit code, omitting the default setting for selected output format.

`org` directive sets address at which the following code is expected to appear in memory. It should be followed by numerical expression specifying the address.

Below are described different output formats with the directives specific to these formats.

2.4.1 MZ executable

To select the MZ output format, use `format MZ` directive. The default code setting for this format is 16–bit.

`segment` directive defines a new segment, it should be followed by label, which value will be the number of defined segment, optionally `use16` or `use32` word can follow to specify whether code in this segment should be 16–bit or 32–bit. The origin of segment is aligned to paragraph (16 bytes). All the labels defined then will have values relative to the beginning of this segment.

`entry` directive sets the entry point for MZ executable, it should be followed by the far address (name of segment, colon and the offset inside segment) of desired entry point.

`stack` directive sets up the stack for MZ executable. It can be followed by numerical expression specifying the size of stack to be created automatically

or by the far address of initial stack frame when you want to set up the stack manually. When no stack is defined, the stack of default size 4096 bytes will be created.

`heap` directive should be followed by a 16–bit value defining maximum size of additional heap in paragraphs (this is heap in addition to stack and undefined data). Use `heap 0` to always allocate only memory program really needs. Default size of heap is 65535.

2.4.2 Portable Executable

To select the Portable Executable output format, use `format PE` directive, it can be followed by additional format settings: use `console`, `GUI` or `native` operator selects the target subsystem (floating point value specifying subsystem version can follow), `DLL` marks the output file as a dynamic link library. Then can follow the `at` operator and the numerical expression specifying the base of PE image and then optionally `on` operator followed by the quoted string containing file name selects custom MZ stub for PE program (when specified file is not a MZ executable, it is treated as a flat binary executable file and converted into MZ format). The default code setting for this format is 32–bit.

`section` directive defines a new section, it should be followed by quoted string defining the name of section, then one or more section flags can follow. Available flags are: `code`, `data`, `readable`, `writeable`, `executable`, `shareable`, `discardable`. Among with flags also on of special PE data identifiers can be specified to mark the whole section as a special data, possible identifiers are `export`, `import`, `resource` and `fixups`. If the section is marked to contain fixups, they are generated automatically and no more data needs to be defined in this section. The origin of section is aligned to page (4096 bytes).

`entry` directive sets the entry point for Portable Executable, the value of entry point should follow.

`stack` directive sets up the size of stack for Portable Executable, value of stack reserve size should follow, optionally value of stack commit separated with comma can follow. When stack is not defined, it's set by default to size of 4096 bytes.

`heap` directive chooses the size of heap for Portable Executable, value of heap reserve size should follow, optionally value of heap commit separated with comma can follow. When no heap is defined, it is set by default to size of 65536 bytes, when size of heap commit is unspecified, it is by default set to zero.

`data` directive begins the definition of special PE data, it should be followed by one of the data identifiers (`export`, `import`, `resource` or `fixups`) or by the number of data entry in PE header. The data should be defined in next lines, ended with `end data` directive. When fixups data definition is chosen, they are generated automatically and no more data needs to be defined there.

2.4.3 Common Object File Format

To select Common Object File Format, use `format COFF` or `format MS COFF` directive whether you want to create simple or Microsoft COFF file. The default code setting for this format is 32–bit.

`section` directive defines a new section, it should be followed by quoted string defining the name of section, then one or more section flags can follow. Available flags are: `code` and `data` for both COFF variants, `readable`, `writeable`, `executable`, `shareable` and `discardable` only for Microsoft COFF variant. The origin of section is aligned to page (4096 bytes).

`extrn` directive defines the external symbol, it should be followed by the name of symbol and optionally the size operator specifying the size of data labelled by this symbol.

`public` directive declares the existing symbol as public, it should be followed by the name of symbol.

Chapter 3

Tutorials

[...]