I. Idris Assembler Conventions for 8086

NAME

As.86 - The 8086 Assembly Language

FUNCTION

The as.86 language facilitates writing machine level code for the 8086/7/8 family of computers. It also accepts the additional instructions available with the 80186/286 processors. Its primary design goal is to express the readable output of the C compiler code generator p2.86 in a form that can be translated as rapidly as possible to relocatable object form; but it also includes limited facilities for aiding the hand construction of machine code. What it does not include are listing output control, macro definitions, conditional assembly directives, or branch shortening logic.

SYNTAX

Source code is taken as a sequence of zero or more lines, where each line is a sequence of no more than 511 characters terminated by either a newline or a semicolon. The slash ´/´ is a comment delimiter; all characters between the slash and the next newline are equivalent to a single newline. Each line may contain at most one command, which is specified by a sequence of zero or more tokens; an empty line is the null command, which does nothing.

A token is any one of the characters "!+-,:=&*[]", a number, a quoted string, or an identifier. A number is the ASCII value of a character x when written as ´x, or a digit string; a character x may be replaced by the escape sequence \c, where c is in the sequence "btnvfr" (in either case) for the octal values [010, 015], or where c is one to three decimal digits taken as an octal number giving the value of the character. A digit string is interpreted as a hexadecimal number if introduced by "0x" or "0X", as octal if introduced by "0", and as decimal otherwise. Octal and decimal constants may contain any decimal digits; hexadecimal constants may also contain the letters a through f, in either case, to specify the values 10 through 15. Tokens that might be otherwise confused must be separated by whitespace, which consists of the ASCII space ´ ´ and all non-printing characters.

IDENTIFIERS

An identifier is a letter or the character ´.´, followed by zero or more letters or digits; letters are characters in the set [a-zA-Z_], with uppercase distinguished from lowercase; at most nine characters of an identifier are retained.

There are many pre-defined identifiers, most of which represent machine instructions. New identifiers are defined either in a label prefix or in a defining command; the scope of a definition is from its defining occurrence through the end of the source text. A label prefix consists of an identifier followed by a colon ´:´; any number of labels may occur at the beginning of a command line. A label prefix defines the identifier as being equal to the (relocatable) address of the next byte for which code is to be generated. A defining command consists of an identifier, followed by an equals ´=´, followed by an expression; it defines the identifier as being equal to the value of the expression, which must contain no undefined identifiers.

There are also twenty numeric labels, identified by a digit sequence whose value is in the range [0, 9], followed immediately by a ´b´ or an ´f´. A definition of the form "#:" or "# = expr", where # is in the range [0, 9] makes the corresponding #b numeric label defined and renders the corresponding #f undefined; further, any previous references to #f are given this definition. Thus, #b always refers to the last definition of #, which must exist, and #f always refers to the next definition, which must eventually be defined.

EXPRESSIONS

Expressions consist of an alternation of terms and operators. A term is a number, a numeric label, or an identifier. There are three operators: plus ´+´ for addition, minus ´-´ for subtraction, and not ´!´ for bitwise equivalence. If an expression begins with an operator, an implicit zero term is prepended to the expression; if an expression ends with an operator, an error is reported. Thus plus, minus, and not have their usual unary meanings.

Expression terms consist of numbers or user-defined identifiers; register names are not allowed. Two terms may be added if at most one is undefined or relocatable; two terms may be subtracted if the right is absolute, relocatable to the same base as the left, or involves the same undefined symbol as the left; two terms may be equivalenced only if both are absolute.

RELOCATION

Relocation is always in terms of one of three code sections: the text section, which normally receives all executable instructions and is normally read only; the data section, which normally receives all initialized data areas and is normally read/write; and the bss section, which can accept only space reservations and is initialized to zeros at start of execution. The pre-defined variable dot "." is maintained as the location counter; its value is the (relocatable) address of the next byte for which code is to be generated at the start of each command line.

Switching between sections is accomplished by use of the commands ".text", ".data", and ".bss"; each sets dot to wherever code generation left off for that section and forces an even byte boundary. Initially, all three sections are of length zero and dot is in the text section. Space may be reserved by a definition of the form ". = . + expr", where expr is an absolute expression whose value advances the location counter, or by the command ".space expr", where expr serves the same function. One byte of space may be conditionally reserved by the command ".even", which ensures that dot is henceforth on an even byte boundary, relative to the start of the section it is in.

GLOBALS

The command ".globl ident [, ident]*" declares the one or more identifiers to be globally known, i.e. available for reference by other modules. The command ".extern ident [, ident]*" serves the same function, though conventionally reserved for flagging externally-defined symbols. Undefined identifiers may thus be given definitions by other modules at load time, provided there is exactly one globally known defining instance for each such identifier.

A special form of global reference is given by the command ".comm ident, expr", which makes ident globally known and treats it as undefined in the current module; if no defining instance exists at link time, however, then this command requests that at least expr bytes be reserved in the combined bss section and that the identifier be defined as the address of the first byte of that area (thus bss for "Block Started by Symbol"). As before, expr represents an expression whose value is absolute.

CODE GENERATION

Code may be generated into the current section, provided it is text or data, either a byte at a time or a word at a time. Bytes of code are generated by the command ".byte expr [, expr]*", where each expr is an absolute expression whose least significant byte defines the next byte of code to be generated. Words of code are generated by the command ".word expr [, expr]*", where each expr is a (perhaps relocatable) expression defining the next word of code to generate. A word need not (but usually should) begin on an even byte boundary.

To simplify generating ASCII strings, any sequence of character constants may be replaced by the equivalent sequence of characters, enclosed in double quotes. Thus the command "string" might be used as a shorthand for, in this case, ".byte ´s, ´t, ´r, ´i, ´n, ´g". The usual escape sequences for characters apply.

ADDRESS MODES

All remaining commands are machine instructions. Each is introduced by an identifier defined as an instruction and, depending upon the formats defined for that instruction, is followed by zero, one, or two operands. An operand may be an immediate value (absolute or relocatable), an 8086 or 8087 register name, a memory reference, or a simple memory displacement (which can be PC-relative).

Given a single, perhaps generic, mnemonic, as.86 matches the operands following against a set of templates for the allowable operand combinations. An immediate operand consists of an absolute expression, or the character ´&´ followed by any expression. A register name is any of the register identifiers defined below.

A simple memory displacement consists of a single relocatable expression, or the character '*' followed by any expression. A general memory reference consists of such a displacement, or of a series of index expressions, or of a displacement followed by index expressions. Each index expression is enclosed in square brackets "[]", and may be one of the register names "bp", "bx", "di", or "si", or an absolute or relocatable expression. In the last case, the index expression is added to its predecessors (and to the original displacement, if any), to form the displacement in the eventual binary encoding of the instruction. Thus "sym[6]", "sym+6", and "[sym][6]" are all equivalent.

as.86 chooses a size for an operation by the following rules: If neither operand has a size component (explicit or implied), then the assembler makes the operation word-sized (quadword-sized for floating-point operations). If exactly one operand has a size component, the other is given a matching one. Otherwise, both operands must have the same size. Any operand (except a relocatable immediate) may be assigned a size component by preceding it with a two-character modifier:

Name	Meaning
.b	operand is byte-sized
.s	operand is byte-sized
.w	operand is word-sized
.d	operand is doubleword
.q	operand is quadword
.t	operand is "ten-byte"

The modifier ".s" conventionally precedes short jump displacements; ".b" is used for data manipulation. Implicit sizes (as for register names) cannot be overridden by these modifiers.

Numerous instructions take dummy operands whose only effect is to set some opcode bit(s) in the binary encoding of the instruction. Without exception, as.86 follows the standard Intel conventions for their specification, to wit: the string instructions are followed by one or two operands that reference memory, only the size of which has any effect; and the shift instructions will accept the register "cl" or a "1" as their second operand (though "1" is presumed if no second operand is given). In addition, floating-point instructions operating on the stack top and next-lower element may be specified with no operands. Finally, the size of the data to be sent with an in or out instruction is given by the usual size modifier, specified alone if need be, though ax (or al) and dx may be given if desired.

Several out-of-the-way instructions are given special treatment detailed below, after the list of instruction mnemonics.

PRE-DEFINED IDENTIFIERS

Pre-defined identifiers fall into three groups: register names, command keywords, and instructions. A register name is one of the identifiers "ax", "bx", "cx", "dx", "bp", "di", "si", or "sp" for the word-sized 8086 CPU registers; "al", "bl", "cl", "dl", "ah", "bh", "ch" or "dh" for the byte-sized 8086 registers; "cs", "ds", "es", or "ss" for the segment registers; or "fr0" through "fr7" for the registers in the 8087 stack.

The keywords, and the command formats they imply, are:

```
    .bss
    .byte   expr [, expr]*
    .comm   ident, expr
    .data
    .even
    .extern ident [, ident]*
    .public ident [, ident]*
    .space  expr
    .text
    .word   expr [, expr]*
```

where expr implies an expression (perhaps constrained to be absolute), ident is an identifier, and "[x]*" implies zero or more repetitions of x.

To complete the list of commands other than instructions:

```
    ident = expr    / defines ident as expr
    "string"        / generates code for string characters
```

Instructions are named for the most part by standard mnemonics:

.cseg	fadd	fld	fstenv	jge	lidt	repnz
.dseg	faddp	fld1	fstp	jl	lldt	repz
.eseg	fbld	fldcw	fstpt	jle	lmsw	ret
.sseg	fbstp	fldenv	fstsw	jmp	lock	reti
aaa	fchs	fldl2e	fsub	jmpi	lods	rol
aad	fclex	fldl2t	fsubp	jna	loop	ror
aam	fcom	fldlg2	fsubr	jnae	loope	sahf
aas	fcomp	fldln2	fsubrp	jnb	loopne	sal
adc	fcompp	fldpi	ftst	jnbe	loopnz	sar
add	fdecstp	fldt	fwait	jnc	loopz	sbb
and	fdisi	fldz	fxam	jne	lsl	scas
arpl	fdiv	fmul	fxch	jng	ltr	sgdt
bound	fdivp	fmulp	fxtract	jnge	mov	shl
call	fdivr	fnclex	fyl2x	jnl	movs	shr
calli	fdivrp	fndisi	fyl2xp1	jnle	mul	sidt
cbw	feni	fneni	hlt	jno	neg	sldt
clc	ffree	fninit	idiv	jnp	nop	smsw
cld	fiadd	fnop	imul	jns	not	stc
cli	ficom	fnsave	in	jnz	or	std
clts	ficomp	fnstcw	inc	jo	out	sti
cmc	fidiv	fnstenv	ins	jp	outs	stos
cmp	fidivr	fnstsw	int	jpe	pop	str
cmps	fild	fpatan	into	jpo	popa	sub
cwd	fildq	fprem	iret	js	popf	test
daa	fimul	fptan	ja	jz	push	verr
das	fincstp	frndint	jae	lahf	pusha	verw
dec	finit	frstor	jb	lar	pushf	wait
div	fist	fsave	jbe	lds	rcl	xchg
enter	fistp	fscale	jc	lea	rcr	xlat
esc	fistpq	fsqrt	jcxz	leave	rep	xor
f2xm1	fisub	fst	je	les	repe	
fabs	fisubr	fstcw	jg	lgdt	repne	

Note that as.86 provides separate mnemonics (specified as instructions with no operands) for the segment override prefixes: ".cseg", ".dseg", ".eseg", and ".sseg". In addition, moves to/from the 8087 stack of quad-word integers and ten-byte (temporary) reals are specified by the special mnemonics "fildq", "fistpq", "fldt", and "fstpt". Finally, the mnemonics "calli" and "jmpi" are provided for intersegment calls and jumps. The direct version of each takes two operands: a relocatable expression giving the offset in the new segment, then a relocatable expression giving the segment (or "paragraph") address. The mnemonic "reti" calls for an intersegment return to be generated.

ERROR MESSAGES

Errors are reported by the as.86 translator in English, albeit terse, as opposed to the conventional assembler flags. Any message ending in an exclamation mark ´!´ indicates problems with the translator per se (they should "never happen") and hence should be reported to the maintainers. Here is a complete list of errors that can be produced by erroneous input or by an inadequate operating environment:

```
#b undefined                  can´t read xxx
.comm defined xxx             can´t write object file
bad #:                        illegal character xxx
bad #f or #b                  invalid size
bad .=                        missing expr
bad .comm size                missing index
bad .space value              missing term
bad command                   missing xxx
bad immediate                 redefinition of xxx
bad index register            register not allowed
bad memory ref                reloc + reloc
bad operand(s)                relocatable byte
bad register combination      size mismatch
bad temp file read            string too large
bad third operand             too many symbols
byte pc range                 undefined #f
can´t backup .                unknown instruction
can´t create temp file        x ! reloc
can´t create xxx              x - reloc
can´t load .bss
```

In all cases, "xxx" stands for an actual character, identifier, or filename supplied by as.86.