I. IDRIS Assembler Conventions for IBM/370

NAME

As.370 - The IBM/370 Assembly Language

FUNCTION

The as.370 language facilitates writing machine level code for the IBM/370 family of computers. Its primary design goal is to express the symbolic output of the C compiler code generator p2.370 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. It does not include 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 of the characters "()!^+-,@:=<>", a number, or an identifier. A number is the ASCII value of a character x when written as 'x, or the ASCII value of the two characters xy when written "xy, 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. If a decimal number is followed by a '#', it will be used as a base for a digit string immediately following, which represents the corresponding number. Any base between 2 and 16 inclusive may be used in constructing a based number. Octal or decimal constants may contain any decimal digit; hexadecimal constants may also contain the letters a through f, in either case, to specify the values 10 to 15. Based constants may contain only the decimal digits and letters that are defined for the base.

IDENTIFIERS

An identifier is one or more letters and digits, beginning with a letter; letters are characters in the set [a-zA-Z._~], with uppercase distinguished from lowercase; at most nine characters of an identifier are retained. Tokens that might be otherwise confused must be separated by whitespace, which consists of the ASCII space ' ' and all non-printing characters.

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 ocurrence through the end of the source text. A label prefix consists of an identifer 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 provided.

EXPRESSIONS

Expressions consist of an alternation of terms and operators. A term is a number, a numeric label, or an identifier. There are four operators: plus '+' for addition, minus '-' for subtraction, not '!' for bitwise equivalence, and the at-sign '@' for address generation (very special). 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.

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; the at-sign may be used only between a byte constant on the left and an address on the right (useful in generating CCW and other things that use a 24-bit address on the right and a byte field on the left).

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 zero 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.

Swithching 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. Space may be conditionally reserved by the command ".align expr", where expr specifies the number of low-order bits that must be zero in the value of the location counter (e.g., ".align 2" forces a four-byte boundary).

GLOBALS

The command ".globl ident [, ident]*" declares the one or more identifiers to be globally known, i.e., available for reference by other modules. 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, one, two, or four bytes at a time, with one of the commands ".byte expr [, expr]*", ".word expr [, expr]*", or ".long expr [, expr]*". For byte or word generation, each expr is an absolute expression whose least significant byte or word defines the next code to be generated; longword generation may also involve relocatable expressions.

To simplify generating ASCII strings, the command "<string>" may 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 format defined for that instruction, is followed by zero, one, two, or three address modes. Some address modes are constrained to be absolute expressions, others must be addresses within some narrow range of the current base register, others must be certain machine registers.

General address modes are built from expressions. A register address mode consists of a constant expression with a value between 0 and 15 inclusive. Indexing is indicated by the expression "disp(reg)", base relative by "name_expr", and base relative and indexed by "name_expr(reg)" or "disp(reg,reg)", where reg is a register expression, name_expr is a "name expression" (as defined below), and disp is a constant expression with a value between 0 and 4095 inclusive. By special dispensation, "0(reg)" may be written "(reg)".

A name expression name_expr is a relocatable expression that can be resolved to a reference to a base register plus a 12-bit displacement. Base registers are specified by the statement "using expr, reg [, reg]*", which tells the assembler that the value of the relocatable expression expr has been loaded into register reg (and that this value, plus successive multiples of 4096, is in the succeeding registers specified). All of the registers currently specified by a "using" statement are scanned in resolving a name expression, until one is found within the appropriate distance from name_expr. If no appropriate base register is defined, an error results. The statement "drop reg [, reg]*" is used to remove a register from the list of base registers scanned, when its value is no longer usable.

PRE-DEFINED IDENTIFIERS

Pre-defined identifiers fall into two groups: command keywords, and instructions. The keywords, and the command formats they imply are:

```
.align  expr
.bss
.byte   expr [, expr]*
.comm   ident, expr
.data
.globl  ident [, ident]*
.long   expr [, expr]*
.text
.word   expr [, expr]*
drop    reg [, reg]*
using   expr , reg [, reg]*
```

where expr implies an expression (perhaps constrained to be absolute), ident is an identifier, reg is a register name, 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 by standard mnemonics. Those pre-defined in as.370 are:

a	bxh	hdv	m	rrb	stck
ad	bxle	her	mc	rrbe	stckc
adr	bz	hio	md	s	stcm
ae	c	ic	mdr	sac	stctl
aer	cd	iac	me	sck	std
ah	cdr	icm	mer	sckc	ste
al	cds	ipk	mh	sd	sth
alr	ce	ipte	mp	sdr	stidc
ap	cer	isk	mr	se	stidp
ar	ch	iske	mvc	ser	stm
au	cl	ivsk	mvcin	sh	stnsm
aur	clc	l	mvck	sigp	stosm
aw	clcl	la	mvcl	sio	stpt
awr	cli	lasp	mvcp	siof	stpx
axr	clm	lcdr	mvcs	sl	su
b	clr	lcer	mvi	sla	sur
bal	clrch	lcr	mvn	slda	svc
balr	clrio	lctl	mvo	sldl	sw
bas	concs	ld	mvz	sll	swr
basr	cp	ldr	mxd	slr	sxr
bc	cr	le	mxdr	sp	tb
bcr	cs	ler	mxr	spka	tch
bct	cvb	lh	n	spm	tio
bctr	cvd	lm	nc	spt	tm
be	d	lndr	ni	spx	tprot
bh	dd	lner	nop	sr	tr
bl	ddr	lnr	nopr	sra	trt
bm	de	lpdr	nr	srda	ts
bne	der	lper	o	srdl	unpk
bnh	discs	lpr	oc	srl	wrd
bnl	dp	lpsw	oi	srp	x
bnm	dr	lr	or	ssar	xc
bno	ed	lra	pack	ssk	xi
bnp	edmk	lrdr	pc	sske	xr
bnz	epar	lrer	pt	ssm	zap
bo	esar	ltdr	ptlb	st	
bp	ex	lter	rdd	stap	
br	hdr	ltr	rio	stc	

ERROR MESSAGES

Errors are reported by the as.370 translator in English, albeit tersely, as opposed to the conventional assembler flags. Any messages ending in an exclamation mark '!' indicate 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
.comm defined xxx
bad #:
bad #f or #b
bad .comm size
bad alignment
bad boundary
bad byte constant
bad command
bad temp file read
can't backup .
can't create temp file
can't create xxx
can't load .bss
can't load .bss
can't read xxx
can't write object file
extra operand
illegal .=
illegal character xxx
illegal using
missing )
missing identifier
missing newline
missing operand
missing term
missing xxx
odd boundary
redefinition of xxx
register required
reloc + reloc
reloc on odd boundary
relocatable byte
relocatable word
string too large
too many symbols
undefined #f
using error
x ! reloc
x - reloc
```

In all cases, "xxx" stands for an actual character, identifier, or filename supplied by as.370.