I. Idris Assembler Conventions for PDP-11

I - 1 **As.11** The PDP-11 Assembly Language

NAME

As.11 - The PDP-11 Assembly Language

FUNCTION

The as.11 language facilitates writing machine level code for the PDP-11 family of computers. Its primary design goal is to express the readable output of the C compiler code generator p2.11 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 "()!^+-,:=$*<", the sequence "-(", the sequence ")+", 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 as "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 one or more decimal digits, which are taken as an octal number unless immediately followed by a point '.', in which case the number is taken as decimal.

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 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 four operators: plus '+' for addition, minus '-' for subtraction, not '!' for bitwise equivalence, and lookslike '^' for instruction definition. 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; lookslike may only be applied between two absolute terms or between an absolute on the left and an instruction on the right.

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 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 ". = . + #", where # is an absolute expression whose value advances the location counter. 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. 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, #", 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 # 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, # 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 # [, #]*", where each # is an absolute expression whose least significant byte defines the next byte of code to be generated. Words of code are generated simply by writing an expression, which is taken as a command to generate a word of code. A word must begin on an even byte boundary.

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, or two address modes. Some address modes are constrained to be absolute expressions, others must be relocatable addresses within some narrow range of the current location counter, others must be certain machine registers, but most address modes on the PDP-11 are quite general.

General address modes are built from expressions and the pre-defined register identifiers "r0", "r1", "r2", "r3", "r4", "r5", "sp", and "pc". A register address mode consists of simply a register expression. Autoincrement is indicated by the form "(reg)+", autodecrement by "-(reg)", indexing by "expr(reg)", program counter relative by "expr", and immediate by "$expr", where expr is any expression and reg is any register expression. Any of the above address modes may be preceded by an asterisk '*' to signal indirection. By special dispensation, "*reg" may be written "(reg)" and "*0(reg)" may be written "*(reg)".

INSTRUCTION FORMATS

Each instruction is characterized by a value, which is its basic encoding as a machine word, and a format, which specifies how its operands are to be combined with the code word. A format of zero means the instruction may take no operands. If the 07700 bits of the format are nonzero, then these six bits determine the legitimate modes for the left operand and the low six bits are for the right operand; otherwise the low six bits are for the single operand that is permitted.

If the six-bit operand code has its low three bits zero, then no registers are acceptable as an operand; a value of 1 means that [r0, r3] are acceptable; 3 means that [r0, r5] are acceptable; and 7 means that all registers are acceptable. If the high three bits of the operand code are zero, then only a register is acceptable as the operand. A value of 1 means that the low three bits of an absolute number are used (as in spl); 2 means that the low six bits of an absolute number are used (as in mark); 3 means that the operand must be within reach of an sob instruction, and formatted as such; 4 means that the low eight bits of an absolute number are used (as in emt); 5 means that the operand must be within reach of a conditional branch instruction, and formatted as such; and 6 or 7 means any address mode is permitted.

If the 010000 bit is set in a format, then the left operand has its mode bits added to the low end of the code word and the right operand has its mode bits shifted left six bits before being added to the code word; otherwise the left operand is shifted and the right is left alone. If the 020000 bit is set, then the instruction is assumed to be movf and the instructions ldd and std are tried in turn.

New instructions may be introduced by the lookslike operator. A command of the form "fadd = 075000 ^ rts" defines fadd as an instruction whose code word is 075000 and whose format matches that of rts, i.e. 000007. Alternatively, a restricted mov that cannot clobber sp or pc could be defined with an explicit format as "move = 010000 ^ 07773". It is not possible to specify the 020000 bit except by inheritance.

PRE-DEFINED IDENTIFIERS

Pre-defined identifiers fall into three groups: registers, command keywords, and instructions. The registers are, in order, "r0", "r1", "r2", "r3", "r4", "r5", "sp", and "pc"; other names for registers may only be formed by definitions of the form "temp = r3" or by expressions of the form "r2 + 1". The keywords, and the command formats they imply are:

```
.bss
.byte   # [, #]*
.comm   ident, #
.data
.even
.globl  ident [, ident]*
.text
```

where # implies an absolute expression, 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 characterized by a base code word and a format. Those pre-defined in as.11 are:

OP	CODE	FORMAT	OP	CODE	FORMAT	OP	CODE	FORMAT
absf	0170600	000063	clr	0005000	000067	mul	0070000	017707
adc	0005500	000067	clrb	0105000	000067	mulf	0171000	017301
adcb	0105500	000067	clrf	0170400	000063	neg	0005400	000067
add	0060000	007777	clv	0000242	000000	negb	0105400	000067
addf	0172000	017301	clz	0000244	000000	negf	0170700	000063
ash	0072000	017707	cmp	0020000	007777	reset	0000005	000000
ashc	0073000	017707	cmpb	0120000	007777	rol	0006100	000067
asl	0006300	000067	cmpf	0173400	017301	rolb	0106100	000067
aslb	0106300	000067	com	0005100	000067	ror	0006000	000067
asr	0006200	000067	comb	0105100	000067	rorb	0106000	000067
asrb	0106200	000067	dec	0005300	000067	rti	0000002	000000
bcc	0103000	000050	decb	0105300	000067	rts	0000200	000007
bcs	0103400	000050	div	0071000	017707	rtt	0000006	000000
beq	0001400	000050	divf	0174400	017301	sbc	0005600	000067
bge	0002000	000050	emt	0104000	000040	sbcb	0105600	000067
bgt	0003000	000050	halt	0000000	000000	scc	0000277	000000
bhi	0101000	000050	inc	0005200	000067	sec	0000261	000000
bhis	0103000	000050	incb	0105200	000067	sen	0000262	000000
bic	0040000	007777	iot	0000004	000000	setd	0170011	000000
bicb	0140000	007777	jmp	0000100	000060	setf	0170001	000000
bis	0050000	007777	jsr	0004000	000770	seti	0170002	000000
bisb	0150000	007777	ldfps	0170100	000067	setl	0170012	000000
bit	0030000	007777	mark	0006400	000020	sev	0000263	000000
bitb	0130000	007777	mfpd	0106500	000067	sez	0000264	000000
ble	0003400	000050	mfpi	0006500	000067	sob	0077000	000730
blo	0103400	000050	mfps	0106700	000067	spl	0002300	000010
blos	0101400	000050	modf	0171400	017301	stfps	0170200	000067
blt	0002400	000050	mov	0010000	007777	stst	0170300	000067
bmi	0100400	000050	movb	0110000	007777	sub	0160000	007777
bne	0001000	000050	movei	0175000	000177	subf	0173000	017301
bpl	0100000	000050	movf	0172400	020000	swab	0000300	000067
bpt	0000003	000000	movfi	0175400	000177	sxt	0006700	000067
br	0000400	000050	movfo	0176000	000173	sys	0104400	000040
bvc	0102000	000050	movie	0176400	017701	tst	0005700	000067
bvs	0102400	000050	movif	0177000	017701	tstb	0105700	000067
ccc	0000257	000000	movof	0177400	007301	tstf	0170500	000063
cfcc	0170000	000000	mtpd	0106600	000067	wait	0000001	000000
clc	0000241	000000	mtpi	0006600	000067	xor	0074000	000777
cln	0000250	000000	mtps	0106400	000067			

movf is taken as 0172400 with a format of 017301, if that is acceptable, otherwise 0174000 with a format of 000173. Note that the UNIX assembler mnemonics are favored over those used by DEC, and that the LSI-11 floating instructions are not defined. Both of these lapses are easily remedied. The DEC standard mnemonics for the nonstandard pre-defined identifiers are:

as.11	MACRO-11
movei	stexp
movf	ldf or stf
movfi	stcfi
movfo	stcfd
movie	ldexp
movif	ldcif
movof	ldcdf

ERROR MESSAGES

Errors are reported by the as.11 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
.comm defined xxx
bad #:
bad #f or #b
bad .comm size
bad command
bad dest
bad src
bad temp file read
bxx range
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 ^ operand
missing )
missing identifier
missing immediate
missing indirect
missing newline
missing operand
missing term
missing xxx
redefinition of xxx
register required
reloc + reloc
reloc on odd boundary
reloc ^ x
relocatable byte
sob range
string too large
too many symbols
undefined #f
x ! reloc
x - reloc
```

In all cases, "xxx" stands for an actual character, identifier, or filename supplied by as.11.