LITTLE Newsletter Number 4 November 28, 1971

A 'LITTLE' Machine. J. T. Schwartz

I. Introduction

The size of the machine block required to define LITTLE on a new machine can be reduced substantially at what is probably a modest cost in efficiency by using the following technique.

A. Define a parametrised family of 'LITTLE' machines, able to a reasonable degree to 'match' physical computers likely to be of interest. The imaginary machine IM used to approximate a given physical computer PM should 'underestimate' it slightly, i.e., should not permit any instructions whose emulation on the physical machine cannot be performed efficiently.

B. Compile into efficient machine code for IM. This will in particular involve register allocation for the registers of IM.

C. Reassemble the IM code into PM code. Only this last step is dependent on the actual details of PM; the 'reassembler' necessary should be short and simple, as only a local transformation of IM code is here in question. The simplest form of reassembly can amount to nothing more than a one-for-one substitution of short PM code sequences for IM instructions; a slightly more complex (and more optimal) reassembly would involve scanning for local code patterns in the IM code to be reassembled, and a slightly conditional generation of PM code sequences. To aid in the optimisation here implied, certain small items of global information can be preserved in the IM code to be reassembled; namely, each use of an IM register can be tagged as last use before reload/not last use before reload, and loads of constants can be flagged also.

In the present newsletter, a tentative set of parameters for machines IM will be suggested, and a (fixed) opcode set for these machines outlined. The first group of operations suggested will be those necessary to support the present LITTLE intermediate language. A second group, which could support a version of LITTLE

considerably extended in its power to describe operating systems, will also be outlined.

II. A family of 'LITTLE' machines IM.

Each imaginary machine will have a memory consisting of NMEM words of size WDSIZ, and will have NFULL 'full-length' registers of size WDSIZ. It will also have NINDX 'short' or 'index' registers of size INDXSIZ. The parameters NMEM, NFULL, NINDX, WDSIZ, and INDXSIZ specify a machine IM completely. It is assumed that INDXSIZ is large enough so that every word of memory can be addressed by an address contained in an index register.

III. Opcode set for the 'LITTLE' machines IM, basic part.

The opcode set of the 'LITTLE' machines IM is basically double address, with a few instructions of different format provided. The opcodes are as follows. (Full registers are denoted Fi; index registers as Ii.)

Fi←Fj	full register move
Fi←Fi.OR.Fj	boolean or
Fi←Fi.A.Fj	boolean and
Fi←Fi.EX.Fj	exclusive or
Fi←.N.Fj	inversion
Ii←.NB.Fj	number of bits of Fj
Ii←.FB.Fj	position of first non-zero bit; Ii←0 if Fj=0.
Fj←Ii	move Ii to low Fj
Ii←lowbits Fj	move lowbits Fj to Ii
Ii←0	zero Ii
Fj←G(A)	load word A to Fj
Fj←C(A+Ii)	load indexed A(Ii) to Fj
C(A)←Fj	store Fj to word A
C(A+Ii)←Fj	store Fj indexed to word A(Ii)

Ii←lowbits G(A)	load index from word A
Ii←lowbits G(A+Ij)	load index from word A(Ij)
Fj←Fj.R.Ii	rightshift Fj by amount Ij. (end-off, zero extended)
Fj←Fj.L.Ii	leftshift Fj by amount Ij (end-off, zero extended)
Fj←.M.Ii	generate Ii-bit mask in Fj
GO TO L	(unconditional go to label)
SKIP if Fi=Fj	(conditional operations)
SKIP if Ii=Ij	
SKIP if Fi≠Fj	
SKIP if Ii≠Ij	
SKIP if Fi=0	
SKIP if Ii=0	

Integer additional and subtraction have standard definitions only for positive quantities, contained in similar registers; such quantitites are distinguished by the fact that the leading bit of the register is off. If the addition or subtraction operator is applied to quantities not satisfying this restriction, the result is unpredictable. Additions may overflow, and subtractions underflow, this bit. A test on the lead bit is provided.

Similar remarks apply to the comparison tests described below, which may be thought of as being logically equivalent to a subtraction followed by an appropriate sign test.

Fi←Fi+Fj	integer addition
Fi←Fi-Fj	integer subtraction
Ii←Ii+Ij	integer addition
Ii←Ii-Ij	integer subtraction

```
SKIP if Fi lead bit 1
SKIP if Fi lead bit 0
SKIP if Ii lead bit 1
SKIP if Ii lead bit 0
```

SKIP if Fi.GE.Fj	SKIP if Ii.GE.Ij
SKIP if Fi.GT.Fj	SKIP if Ii.GT.Ij
SKIP if Fi.LE.Fj	SKIP if Ii.LE.Ij
SKIP if Fi.LT.Fj	SKIP if Ii.LT.Ij

Integer multiplication is assumed to be available for F-register quantities, provided that the quantities being multiplied have no bits on in the upper half of the register. If this condition is violated, the result of a multiplication operator is unpredictable.

Integer division and remainder are assumed to be available for for F-register quantities provided that neither operand has its lead bit on. Division by zero will give an error stop.

Fi ← Fi * Fj	integer multiply
Fi ← Fi/Fj	integer divide
Fi ← Fi//Fj	integer remainder

V. Stores.

STORE Fi TO A+Ij	indexed store
STORE Ii TO A+Ij	indexed store of index (highbits of word set to zero)

STORE Fi TO A	stores value to address A
STORE Ii TO A	stores index to address A (highbits of word set to zero)

VI. Floating arithmetic.

Floating arithmetic is not specified for the 'core' or 'machine independent' section of the LITTLE language, since floating arithmetic operations are normally highly machine dependent. It may, however, be desirable to support these operations in the LITTLE machine, since this will enable languages like FORTRAN to be compiled for the same machine, i.e., to be supported by the LITTLE intermediate language. Naturally, a program in which floating arithmetic is used may lose some of its machine independence. To limit this effect, operations are proposed which provide a narrow path of communication between integer and floating quantities. Note in particular that the bit-pattern of a floating quantity should in general not be examined.

A special single bit quantity called FLOAT ERROR is associated with the floating point operations. This quantity is turned on by various error conditions in floating point operations, such as floating point overflow, underflow, etc. Two skip operations which test this bit are described below; testing the bit turns it off.

Fi ← Fi ⊕ Fj	floating add
Fi ← Fi ⊖ Fj	floating subtract
Fi ← Fi ⊗ Fj	floating multiply
Fi ← Fi ⊘ Fj	floating divide
Fi ← .FLOAT.Fj	float and round integer
Fi ← .INTEGER.Fj	Fi becomes greatest integer less than absolute value of Fj
SKIP if Fi.FGE.Fj	(floating greater or equal)
SKIP if Fi.FGT.Fj	(floating greater than)
SKIP if Fi.FLE.Fj	(floating less or equal)
SKIP if Fi.FLT.Fj	(floating less than)

Fi ⟵ 0	(load floating zero)
SKIP if no FLOAT ERROR	
SKIP if FLOAT ERROR	

VII. Input-output, External Device Control.

The descriptions below assume a logical environment in which a central memory and a set of named files exist. A file name is assumed to have the same length in bits as a full register. Files have the nature of arrays, i.e., they consist of words sequentially addressed. Otherwise empty file addresses contain zeros. A separate set of tape read/write operations are provided. Tapes are always read and written from a nominal current position, which is always a nominal 'record gap'. Each record gap has a specified 'eor-level'; a logical tape is terminated by a record gap of maximum possible level.

Read and write operations are provided in three basic forms:

a - normal form, implying suspension of processing during i/o operation.

b - buffered form, implying continuation of processing during i/o operation, with flag posted in specified location on completion of i/o.

c - interrupt form, implying continuation of processing during i/o operation, with specified interrupt on completion of i/o.

MOVE	LOC,NEWLOC,NWDS	move N words from array beginning at LOC to array beginning at NEWLOC
READ	FILE,FLOC,LOC,NWDS	read N words beginning at location FLOC in specified file to array beginning at LOC

WRITE	FILE, FLOC, LOC, NWDS	write N words from array beginning at LOC to specified file, first file position written being FLOC.
READT	TFILE, LOC, NWDS, EORLEV	read up to N words from specified tape file to array beginning at LOC. Read until first following end record mark of level at least EORLEV. Store level of this mark in EORLEV.
WRITET	TFILE, LOC, NWDS, EORLEV	write N words to specified tape file from array beginning at LOC. At end of these words, write an end record mark of specified level.
BREAD	FILE, FLOC, LOC, NWDS, READY	operation similar to 'read', but proceeds in parallel with computation. READY set to 0 when operation initiated, to 1 when operation completed.

BWRITE	FILE, FLOC, LOC, NWDS, READY	similar 'concurrent' versions of	WRITE
BREADT	TFILE, LOC, NWDS, EORLEV, READY		READT
BWRITET	TFILE, LOC, NWDS, EORLEV, READY		WRITET

VIII. Device control.

'Devices' are considered to be files to which control signals may be written and from which control signals ('status information') may be read. Available devices will have particular file names, e.g., 'CLOCK', 'SENSOR1', 'CONSOLE3', etc. Control signals may also be associated with files to which data could reasonably be written; this will allow error conditions to be sensed, etc. Each class of device will expect control signals in some specified standard format, and will react to a 'read control status' signal by transmitting a status package having some specified standard format. The completion of a control operation may require a time depending on the control parameters supplied; this can be exploited to secure a 'WAIT K MILLISECONDS' effect. Devices absent (or files not yet attached) will deliver a characteristic status package, allowing 'device disconnected' or 'file closed' to be sensed.

As with read/write operations, so also control operations are provided in three basic forms: normal form, implying suspension of processing until operation completion, buffered form, implying the setting of a completion flag, and interrupt form.

READC	FILE,LOC,NWDS	read first N words of status information to array beginning at LOC.
WRITEC	FILE,LOC,NWDS	write N words of control operation from array beginning at LOC.
BREADC	FILE,LOC,NWDS,READY	operation similar to 'readc', but proceeds in parallel with computation. READY set to zero when operation initiated, to 1 when operation completed.
BWRITEC	similar 'concurrent' version of WRITEG.	

A special nominal device PROCESSES can be read repeatedly to obtain a list consisting of all devices for which i/o or control operations are in process. This allows programmed shutdown of these operations on the occurrence of an interrupt.

IX. Interrupts: A LITTLE interrupt causes the following actions:

a - A system interrupt-level-enabled counter is set to its maximum possible value (all interrupts disabled).

b - Control is transferred to a specific code location ('label'); note that since the actual format which code has during execution is not known, code locations lie in an address space logically distinct from that containing data locations.

c - A standard status package, recording all register contents, status of the FLOAT ERROR bit, prior interrupt level, and instruction position, is recorded at a pre-specified data location.

Return from an interrupt is then made by an IRETURN instruction addressing the stored status package.

The interrupt-related LITTLE machine instructions are

ENABLE	LEV	- enable interrupts of level exceeding LEV
IRETURN	SPAK	- interrupt return, using status package stored starting at location SPAK
WAIT		- idle waiting for next interrupt

Corresponding i/o and control i/o operations are as follows.

IREAD FILE, FLOC, LOC, NWDS, LABEL, LEVEL, STATLOC
read N words beginning at location FLOC in specified file to array beginning at address LOC. On conclusion of this input operation, an interrupt of the specified level is to take place, control being transferred to the code address LABEL, with status stored at the data address STATLOC

IWRITE	FILE, FLOC, LOC, NWDS, LABEL, LEVEL, STATLOC
IREADT	TFILE, LOC, NWDS, EORLEV, LABEL, LEVEL, STATLOC
IWRITET	TFILE, LOC, NWDS, EORLEV, LABEL, LEVEL, STATLOC
IREADC	FILE, LOC, NWDS, LABEL, LEVEL, STATLOC
IWRITEC	FILE, LOC, NWDS, LABEL, LEVEL, STATLOC

These last five instructions are corresponding 'interrupting' versions of WRITE, READT, WRITET, READC, WRITEC respectively.

X. Miscellaneous additional points for subsequent consideration.

We have assumed the 'central memory' of the LITTLE machine to be 'real' rather than 'virtual', in that no address-fault conditions are recognised and no address mapping system appears in the language. The opposite decision might be better, and it is desirable to try to work out software conventions which could make explicit within LITTLE address faults and the sequences to be used to recover from them.

In a paged environment of the type envisaged in the preceding paragraph , code might be written in a manner allowing intra-page transfers to proceed normally, with inter-page transfers going through a paging table. An 'ENCODE' instruction, which takes a data array having a known intermediate language format and converts it into true executable format (necessarily machine dependent) is necessary also. This instruction might return the number of words which the resulting code block contains. To secure the execution of such a block, one might have such a master 'system' instruction as

EXECUTE LOCODE, HICODE, LODATA, HIDATA, TIME, STATUSPAK, RETLOC, RETDATA

This instruction would begin or continue the execution of a program with code contained between LOCODE and HICODE, data contained between LODATA and HIDATA. Execution would be permitted for a stated TIME, and would begin with the status package held

at a specified address. Whenever a program initiated in this way attempted to execute a 'privileged' instruction (a READ, WRITE,WRITEC,READC,etc., or an interrupt related instruction) control would return to RETLOC, with the parameters of the request available in the RETDATA area in some standard form. An instruction of this kind should ideally be so designed as to be 'transparent', i.e., so that a program initiated by an EXECUTE has exactly the same form as any other program; this allows systems to be tested while imbedded as subprograms within other systems.

To allow data and code space to be returned to/requested from an enveloping 'operating system' program, special instructions

```
        UPDATA  NWDS
and     UPCODE  NWDS
```

might be desirable. Depending on whether the parameter NWDS is positive or negative, a certain number of pages of DATA (resp. CODE) space would be requested by (or released) by the program executing these instructions; always at the end of its currently available data (or code) space. These instructions might also be provided in 'buffered' and 'interrupt' form.