LITTLE Newsletter Number 7 March 27, 1972

LITTLE for Minicomputers T. Stuart

A. Targets

This report is concerned with the application of LITTLE to small general purpose machines, or minicomputers, having 4-32K words, typically 16-bit. The range of instruction codes for these machines testifies only to the perversity of designers.

Specifically, the following target minis are considered:

1. Honeywell H-316 and DDP-516

These are accumulator machines with a pseudo index register, paging, infinite indirect addressing, and a constant instruction length of one word.

2. IBM 1130 and 1800

These are also accumulator machines with three index registers, a single level of indirect addressing, and an instruction length of one or two words. Operands may generally be either 16 or 32 bits.

3. Hewlett-Packard 2100

No index register is the distinguishing characteristic of this mini. There are two registers, one serving as an accumulator; the other appears somewhat vestigial. Indirect addressing is infinite; the memory is paged. Instruction size is one word, but in certain cases up to eight instructions can be "microprogrammed" into the single word.

4. Varian 620 Series

This one has an accumulator, an index register, and a third register which performs some of the functions of the previous two. The machine is paged, has infinite indirect addressing and single or double word instructions. This machine also comes in an 18-bit version.

5. Data General Nova and Supernova

These machines are the first in this list that are clearly out of the accumulator category. They have four general registers, two of which also function in indexing. Memory is paged, indirect addressing is infinite, and all instructions occupy

one word. The instruction set is idiosyncratic: e.g., skips and register shifts are imbedded in arithmetic instructions.

6. Digital PDP-11

More powerful than any of the above, this mini has eight general registers including a flexible program counter. Especially efficient as a stack processor and with 10 addressing modes in addition to the ability of employing any word in memory as a register, it seems to present a serious challenge in code optimization. Instructions may occupy one, two or three locations, and operations can address either 8 or 16 bit words.

7. Texas Instruments 960A

Perhaps an even tougher challenge will come from this newest minicomputer. There are 16 general registers for arithmetic, logical and indexing operations, with 8 fully and 8 partially available at any given time. Mode switching between the two sets is like an exchange jump in the 6600, but is more flexible since it may be accomplished in half a dozen ways. Instruction length is always two words and operands may be 32, 16, or just 1 bit. Yes, Virginia, there is a Santa; you can address a single bit. Some instructions have three operands. Instruction execution time appears slower than the other minis but small instruction stacks can be accommodated in the CPU. Further, though the manufacturer is not now marketing it, there may be a capability to microprogram your own instruction set.

B. Roadblocks

Despite the diversity, these machines are considered as a group specifically because each one has been found useful (though certainly not always efficient) in a number of applications: compilers for simple languages such as BASIC, time-sharing systems, communication routines, text editing, and my own special interest, graphics support. These types of programs seem clearly a good target for LITTLE, but it is also apparent that a full implementation of LITTLE is impossible on machines of 4-32K with 16 bit words; hence code generation must take place on large machines.

Since the number of application programs for which LITTLE is the best choice will be small for any given minicomputer installation, the work of producing an assembler can only be judged worthwhile, save for pedantic purposes, if there's a reasonable probability of eventual use by other installations. To consider this likely, three, partly obvious conditions must be presupposed:

1) A large computer must be convenient to the minicomputer staff. This will not be the case for many small colleges or commercial operations, but there remains a sufficient number of situations where it will be available. However, physical availability is only part of the problem. Not easy to assess are the effects on the staff of other inconveniences. First, turn around time will generally be slower on the large machone. Second, programming becomes a two pass operation: compiling on one machine, testing on another. Third, there is sometimes a resistance to learning a new language.
2) The large computer must not only be convenient, but LITTLE must already be running in a full implementation on it. No one is going to implement LITTLE on the large machine merely to generate a few programs for the minicomputer, even if the assembler for the mini has already been written on another large computer.
3) For many installations there must already be an existing assembler, since they will lack either the competence, time, or inclination to produce one. This condition will be partly mitigated if the number of desired applicaton programs coded in LITTLE is already large or the assembler is extremely easy to turn out. This last possibility was the subject of Newsletter No. 4 and is also the focus of this one. Nonetheless, no matter how successful an approach to easy assembler development is made, it is not inappropriate to suggest that a serious effort to apply LITTLE to minicomputers should imply substantial continuing aid to others for implementation.

The above three conditions bear little or no relevance to large machines, but present distinct handicaps for the small installation. It might be deemed reasonable to postpone concern

for them, focusing on the problems of large machine transfers, until a more hospitable situation develops. On the other hand, early development of mini computer output could well give additional impetus to interest in LITTLE on the part of large installations, since they invariably have problems with system programming on satellite or remote computers similar to the problems here at Courant.

C. Scope

The concern here is with the conversion of an array containing 30 basic LITTLE operations and their associated operands (the variable and operations array - VOA) into the appropriate binary code of the target machine's loader. To explore the difficulties, several examples of LITTLE code will be transformed into selected representations in the instruction sets of the seven machines described earlier. However, because there is an effect on the subsequent discussion, the first topics considered are a suggested redefinition of the goal and an enumeration of some major factors which have been nearly or completely ignored.

The suggestion is that the goal be not binary code for a loader, but symbolic code for an assembler. Clearly, this is a step backward. The VOA and its associated tables contain a great deal of information that an assembler once again must reconstruct. Nevertheless, a trio of opposing arguments should be heard from:

1) Loaders for minis seem poorly documented, probably because there is very little interest. Many, perhaps most, purchasers deal only with higher level languages. (A new owner of one of the machines treated here was quite adamant in reply when asked for the assembly manual, "BASIC is the only language this computer knows.") The only released documentation of the IBM 1130 loader is its assembler listing.
2) Debugging symbolic code is easier.
3) Special additions to programs may be desirable when the work of modifying the LITTLE compiler is not worthwhile. For example,

a reference to an absolute address, or a device controlling instruction, or special linkages to/from external routines. As with debugging, the inclusion will be easier.

Producing code for the assembler also introduces complications though.

1) Machine dependent assembler directives must be introduced.
2) The symbol length in LITTLE will in general be unacceptable. In none of the seven minis here can seven characters be handled.
3) The target machine assembler will in general require control or delimiting characters unavaiable to the host machine compiler.

Each of these objections is, however, relatively trivial to overcome, and is taken up later. On the other hand, situations may appear where the binary code is unquestionably preferable. Foremost among these possibilities is the case where an assembler does not exist, as it well might with a microprogrammable machine.

There are at least half a dozen major factors that may or will affect the quality of code produced and that are completely ignored in this newsletter.

1) The subjects of virtual memories and associated paging structures in the sense of Newsletter # 3 are not considered. (Hardware paging is a concern below.) This subject is unquestionably important given the size of the minis.
2) Interpretable code was also discussed in Newsletter # 3 and its implementation is ignored here. One can imagine applications where this could be valuable. With the cost of memory now dropping below 40¢/word, problems lacking large data files might well find it possible to pack all the code into core, obviating the need for a disk or tape. This would be more economical when the scale of this approach is great enough to make a read only memory feasible.
3) Handling operations on words of arbitrary lengths is overlooked. I had a nightmare about this one.
4) I/O and interrupts, discussed in Newsletter # 4, are not considered for the seven target machines here. This is a major omission, because in general there will be a greater need for

systems programming of this type in minis than in large machines where a great variety of I/O routines already exist.

5) Computers operating in a real time environment are common among minis. Experience indicates that the approach taken to assembly coding under a real time executive is significantly distinctive, but no thought has been given to what implications, if any, this might have for the LITTLE compiler.

6) Most of the attention in the rest of this report concentrates on code generated by a single LITTLE statement, to the near exclusion of problems with the simple block. For example, the following LITTLE equivalent to a simple FORTRAN DO loop poses strenuous work to get optimal code -

```
        I = 1;
/LAB/   A(I) = B(I); I=I+1;
        IF(I.NE.N) GO TO LAB;
```

Programming in an assembler language would probably result in 6 or 7 instructions for these target machines, while the first attempt to generate code from the VOA will likely lead to 15, 20 or more instructions.

Before any work is begun on providing LITTLE for the minicomputers, it seems advisable that the above points be thought through, even if no effort is made to include such options initially. In the case of problems 3, 4 and 6 it is especially important, and it would probably save future woe to trace typical examples through to the target machines.

D. Pot Holes

In this section the problems encountered during processing of VOA code are examined to determine the influences from three general characteristics of each machine: its instruction code, its registers, and its addressing modes. The examination is often concerned with the possibility of an intermediate, imaginary machine as discussed in LITTLE Newsletter # 4 and to that end especially focuses on the idiosyncracies of these computers, or,

to put it another way, on the cases where the targetinstruction set differs from the 30 element LITTLE set. Such cases are not hard to find. For only 3 or 4 of the 30 is there a mapping into the instruction sets of all 7 machines that is isomorphic: ADD, AND, the unconditional transfer, and perhaps the label operation.

First, let us consider the relationship between the, complexity of a processor and the quality of code it might turn out. For this purpose two of the minis have been chosen, the Honeywell and the PDP-11. Take this LITTLE statement:

```
IF((I.GE.J).OR.(K.EQ.L))GO TO NEXT
```

The VOA entries produced could be the following. (Sometimes VOA entries, both here and below, will be modified from the suggested form by previous operations in the block, and sometimes I am just guessing what is in it, but this will not be of significance for the general discussion.)

#	Operation	Operands	Explanation
1	GE	I,J	If I $\geq$ J, true. Otherwise false.
2	EQ	K,L	If K = L, true. Otherwise false.
3	OR	#1,#2	OR results of operations 1 and 2.
4	IF	#3,NEXT	If result of operation 3 true, go to NEXT.

A simple, straightforward, easy-to-write processor will be one that, lacking a single correlative to the VOA instruction, expands the original in the new basis set. Ignoring address problems for the moment, it need only keep track of the location of the results of previous operations. Such a processor could give this expansion for the Honeywell:

Label	Operation	Operands	Explanation
	LDA	I	Load accumulator with I
	SUB	J	Subtract J
	SMI		Skip if - (I ignore LITTLE rule.)
	JMP	*+3	Skip next 2 instructions

[continued]

```
Label  Operation  Operands        Explanation
       CRA                       Set acc. to false
       SKP                       Unconditional skip
       LDA         =-1           Set acc. to true
       STA         TEMP1         Save result in temp. location.
       LDA         K
       SUB         L
       SZE                       Skip if zero
       JMP         *+3           Skip 2 instructions
       LDA         =-1           Set true
       SKP
       CRA                       Set false
       STA         TEMP2         Save result in temp. location.
       ANA         TEMP1         There is no OR instruction,
       IMA         TEMP1         so these 4 perform it.
       ERA         TEMP2
       ERA         TEMP1
       SZE                       Skip on false.
       JMP         NEXT          Branch on true.
```

Now the question is, how do the above twenty-two instructions compare with code from a programmer or more intelligent processor? Perhaps this:

```
       LDA         I
       SUB         J
       SMI                       Skip if false.
       JMP         NEXT          Jump if true.
       LDA         K
       SUB         L
       SNZ                       Skip if false.
       JMP         NEXT          Jump if true.
```

Eight instructions. Now turn to expansion output for the PDP-11 with a more powerful op set.

```
CLR     R0      Set register 0 to false
CMP     I,J     Compare I and J
BMI     *+2     If I < J, skip
COM     R0      Set reg. 0 to true
CLR     R1      Reg. 1 set to false
CMP     K,L     Compare K and L
BNE     *+2     If K ≠ L, skip
COM     R1      Set reg. 1 to true
BIS     R0,R1   OR reg. 0 into reg. 1
BNZ     NEXT    If true, go to NEXT
```

Ten instructions is much better, but a PDP-11 programmer would undoubtedly do this:

```
CMP     I,J
BGE     NEXT    If I ≥ J, branch
CMP     K,L
BZE     NEXT    IF K = L, branch.
```

The bad-to-good code length ratios, 22/8 and 10/4 for the two machines, will in practice be even worse, because the longer code requires more frequent allocation of memory for addressing purposes.

However, the simple point to be made here is that it will be quite difficult to produce good code. Whereas, in the first instances for each machine, it is quite clear which VOA entry led to which machine instructions, in the second approaches, the processor must be highly aware of the syntax of the VOA. That it must also deal intimately with the target semantics is not apparent from this example, since it is obvious that any intermediate imaginary, LITTLE language that would lead to the good HOneywell code would also lead equally well to the good PDP-11 code. This approach will be examined more closely in following examples.

Another inference can be drawn from these transformations of VOA code. The reader should attempt some alternate transformation which improves on the simplest approach, but does not demand the work of the good code here. Good luck.

An objection could be made that this example is a case which is most properly a function for the syntax analysis segment of the compiler. Although that would be possible with this particular

LITTLE statement, a brief consideration of allied situations leads to the conclusion that it would only be practical if the number of types of operations in the VOA were increased. The merits or demerits of that proposal will not be discussed here.

A final, subjective statement about this example is tendered. Perhaps in a large, fast machine the longer code is acceptable, but in the minicomputer where space is precious, there should be some hope that a processor can turn out the better code, even if not initially.

For a second example, consider the LITTLE expression,

C(I) - D(I) ,

and the VOA entries,

```
1  FETOP  C,X(I)   Fetch O, indexed by I
2  FETOP  D,X(I)   Fetch D
3  -      #2,#1    Subtract result 2 from 1.
```

This is a situation in which the LITTLE machine seems applicable. For example, an index register could be used in the intermediate code as shown:

```
1  SETXR  I        Set index register to I
2  FETCH  C,#1     Fetch indexed C
3  FETCH  D,#1     Fetch D
4  -               Subtract .
```

Such an instruction set would certainly aid code production for a Nova minicomputer:

```
        LDA     2,I       Set reg. 2 to value of I.
        LDA     1,C,2     Get C, indexed by reg. 2
        LDA     0,D,2     Get D
        SUB     0,1       Subtract reg. 0 from reg. 1
```

With the PDP-11, some synthesis of the instructions is required for good code, but it is by no means difficult:

```
        MOV     I,R1      Set index in reg. 1
        MOV     A(R1),R0  Get A, indexed, into Reg. 0
        SUB     B(R1),R0  Subtract indexed B from reg. 0
```

On the other hand, for the HP-2100 mini, the only obvious way to express this is with the following horrendous code:

```
        LDA     ADDC    Get address of C in reg. A
        ADA     I       Add index
        LDA     @0      Replace address by value in reg. A
        LDB     ADDD    Get address of D in reg. B
        ADB     I       Add index
        LDB     @1      Replacement
        CMB             There is no subtraction operator,
        INB             so these 3 instructions do it.
        ADA     1
        -
        -
ADDC    DAC     C       Declare address of C
ADDD    DAC     D       Declare address of D
```

Such code arises because this machine has neither indexed nor immediate addressing modes. In some blocks, the operand addresses probably should be saved for efficiency, and the resulting code will look even worse. A similar, though not quite so bad, situation can develop with the Honeywell machines, which have poorly designed index registers.

The exercise for the reader now is to empathize with the processor that must (1) translate VOA code to the three target codes, or (2) from VOA to the intermediate to the target codes. The conclusion will be not unlike that of our next example.

The third LITTLE transformation problem comes from this statement and its VOA entries.

```
A = .F. 8, 4, B

1  EXOP   8,4,B    Extract 4 bits beginning at pos. 8 from B
2  SASS   #1,A     Assign result to A
```

This is a much simpler form of the extraction operator than the usual case, because the bit position is a constant. When it is a variable, a translation differing markedly from the following Honeywell code would be made. (I have assumed that the result of an extraction is a right justified field with zero fill.)

```
        CRA                     3 instructions to form
        SSM                     the mask.
        ARS         3
        LGR         12          Right justify mask
        STA         TEMP        Save it.
        LDA         B           Get B.
        ARS         5           Right justify desired field.
        ANA         TEMP        AND mask with field.
        STA         A           Store.
```

This is rotten code to perform the given task, but is chosen to illustrate that it would be a suitable and easy expansion of LITTLE machine code. Several of the other target machines are also amenable to this hypothetical LITTLE code:

```
1       FMASK       4           Form 4-bit mask.
2       RSHFT       #1,12       Right shift to justify.
3       RSHFT       B,5         Justify B with right shift.
4       AND         #2,#3       AND results of operations 2, 3.
5       SASS        #4,A        Assign result to A.
```

However, this code poses obstacles for three of the target machines which do not have a general shift capability. The Hewlett-Packard shifts either 1 or 4 positions, the Nova and PDP shift 1 or 8, and the following PDP code demonstrates the incompatibility.

```
        CLR         R0          Zero register 0.
        MOV         B,R1        Get B in reg. 1.
        MOV         =8,R2       Get 8 in reg. 2: counter
LOOP1   ROL         R1          Rotate left reg. 1
        DEC         R2          Decrement counter
        BNZ         LOOP1       Loop till counter zero
        MOV         =4,R2       Get 4 in reg. 2: new counter
LOOP2   ROL         R1          Rotate left reg. 1
        ADC         R0          Add carry bit to reg. 0.
        ASL         R0          Shift reg. 0 left 1 bit.
        DEC         R2          Decrement counter.
        BNZ         LOOP2       Loop till counter zero.
        MOV         R0,A        Assign result to A.
```

With trivial modifications this would be a subroutine that handles variable field widths and positions for any extraction from a 16-bit word, but the important point is that the PDP does not accommodate

itself well to masking operations, and that the processor will have a difficult job to either generate this code or a request for this subroutine when given the suggested LITTLE code. In fact, the only practical solution would involve 2 or 3 subroutines and a much longer expansion.

This example and the previous one demonstrate how we can assume too much about a machine's op code. The next two examples show how we can assume too little. The first non-zero bit operator is the first subject.

A = .FB. B

```
1   FB      B              Determine first non-zero bit in B
2   SASS    #1, A          Assign result to A
```

The Honeywell code that follows is typical of good code for most machines, except that the counter would sometimes be assigned to a temporary variable.

```
        CRA                Clear accumulator
        STA     0          Zero counter (mem. loc. 0)
        LDA     B          Get B
LOOP    LGL     1          Shift left 1 position
        IRS     0          Increment counter
        SSC                Skip if carry occurred
        JMP     LOOP       Loop
        STX     A          Assign the count
```

A simple task for you now is to choose imaginary LITTLE code from which this target code or similar target code can be generated. Whatever the choice, it is unlikely to transform easily into the following appropriate IBM 1130 statements.

```
        LD      B          Get B in accumulator
        LDX  1  16         Set index reg. 1 to 16
        SLCA 1             Shift acc. left, decrementing reg. 1
                            till bit appears in position 1
        LD      1          Get index reg 1 into acc.
        MUL     =-1        There is no complement, so *-1.
        SZE                Skip if acc. is zero.
        A       =17        Correct the count.
        ST      A          Assign result to A.
```

The fifth example involves an expression containing the bit count operator -

.NB. C

The VOA entry is simply -

```
1   NB   C      Count the one bits in C.
```

The following code for the Varian 620 is essentially duplicatable in all target machines.

```
        TZB                 Zero B register.
        LDA     C           Get C in A reg.
        LDXI    16          Set index reg. to 16.
LOOP    LRLA    1           Shift A reg. left 1 bit
        XAN                 Execute next instruction if A reg. < 0.
        IBR                 Increment B register.
        DXR                 Decrement X reg.
        JXZ     NEXT        Skip to next if X reg. = 0.
        JMP     LOOP        Loop
NEXT    -
```

Once again, as with the previous example, it is possible to express the above in LITTLE machine code that would make the processor's function simpler. Yet it would hardly be simple to transform it into the following top notch code for the Texas Instrument machine:

```
        LOT     1,C         Count the one bits in C;
                            put result in reg. 1.
```

The first five examples touched on arithmetic and logical operations, shifting, and bit manipulation. Here the discussion will be about the last major group of instructions -- those which alter program flow. The variations of instruction sets in this area create numerous problems for translation either directly from the VOA or from an intermediate language.

The major divergence occurs in the designer's choice of conditional skips or conditional branches or both; all these choices are represented in the group of machines discussed here.

The obvious implication for code optimization in the case of machines capable of conditional branches is that the use of the conditional branch as a skip will lead to two instructions where one would have sufficed. More serious problems and possibilities in optimization are raised (but not discussed) by the following machine characteristics.

1) Some instruction sets are designed for comparison of two quantities, while others test only the properties of a single quantity.
2) The Texas Instrument mini can perform branches according to the status of any designated bit in the system.
3) Whether the tests are relational or unary, the set of tests is in some cases incomplete. For example, the HP 2100 has no test for negativity.
4) Some machines perform the operations most efficiently on memory, some on registers.
5) In the Varian, one register accommodates the full set; other registers a subset.
6) 'Plus' in some machines also means zero; in others it does not.
7) The Nova possesses no distinct conditional transfer instructions: they are imbedded in arithmetic and logical operations.

The next difficulty for a processor examined here is addressing, and this is perhaps the least difficult. Basically, there are only four addressing modes: direct, indirect, immediate, and relative. Each of these may be determined or complicated by indexing, autoincrementing, paging, or displacement limitations. Even with the variations among machines, it still seems feasible to write a segment of the processor which would handle storage allocation and addressing for all of them, though it will not be an easy task. The tricky problem is the decision on where, or at just what point in processing, to introduce the addressing segment. For example, an appropriate machine instruction sometimes cannot be chosen until the addressing format is known, and the addressing format cannot be chosen until several other instructions with optional formats are determined. Nevertheless, this is not impossible, and will be discussed later in part F of this newsletter.

The eighth difficult topic is register allocation, that is, specifically, whether or not the optimization of the allocation can be accomplished independently. The hypothesis advanced is that a treatment of this process separated from code generation will be feasible only when there are a number of identical registers. The evidence comes from the definition -- a register is defined by the operations which influence it. Hence, any allocator attempting to treat non-identical registers must be aware of all the op codes which differ in all the computers considered. If this seems a trivial thesis, its consequences are not: it eliminates many of the machines here.

A second limiting influence on the practicality of a general allocator occurs when a major segment of the instruction set refers to different registers. For example, with indexed operations the following possibilities can be envisioned.

1. Index registers and general registers are separate.
2. Index registers and general registers are identical.
3. Some of the general registers are also index registers.

All three cases are represented here.

Another complication of some computers is the choice one must make for binary operations among register-register, register-memory, and memory-memory allocations to operands. The choice, when available, will depend not only on subsequent operations, which is tractable, but also on the details of the instruction sets (which may only allow the choice for a subset of the binary ops) and the timing of the machine. A register-to-register instruction is not invariably either faster or smaller.

In part A the general characteristics of each machine were given, but it seems appropriate to consider their registers again in the light of the above observations.

The Honeywell can be eliminated immediately since there are no significant allocation decisions. The Hewlett-Packard and to a greater extent the Varian present a number of choices, but all registers are different. The IBM of course has only one accumulator, but the three index registers present some possibility.

However, inspection of substantial quantities of assembler code encounters the rare use of all three registers simultaneously. This is probably due to the very limited number of instructions which modify the registers. On the other hand, indexed operations will likely occur far more frequently in LITTLE than in most assembler programs. The Texas Instrument computer has 16 registers and the instruction subsets applying to each may be labeled as shown:

Register	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Subset	A	A	A	A	B	C	D	E	F	F	F	F	G	H	I	J

The Nova, as noted earlier, possesses 4 general registers, 2 of which may also be used for indexing. It is conceivable that one could simplify this situation for a general allocator by arbitrarily designating one or both optional registers exclusively as indexers. Obviously the decision would not have entirely laudable consequences, but its benefits are diminished by a singular characteristic of the Nova. Alone among these mini-computers, this op set allows no binary operations outside registers, thus maximizing the importance of optimization.

The PDP-11 is an 8-register machine with 2 reserved for special purposes. The remaining 6 pose a classical case for optimization, modified only by the problem of how or whether to use a register at all. For example, inspection of the instruction set indicates that it is possible (though hardly desirable) to write programs without ever placing an operand in a register. Practically, this flexibility gives rise to optimization problems involving LITTLE statements such as

<*name> = <expr>

where a common occurrence is for the left-hand member also to occur in the expression, and may be factored out to give

<*name> = <expr1><op><*name>

then a register need not be allocated and a reduction in code results. Of course, this decision would be mediated by subsequent code.

The final consideration in this section is a minor point. Entries in the VOA are in triple form -

<operation><operand1><operand2>

An alternative is quadruple form -

<operation><operand1><operand2><result>

Considering particularly the evaluation of expressions, it appears that for some machines it would behoove a preliminary processor of the VOA to produce quadruples, while for others the flexibility and ambiguity of triples are more advantageous. The former case applies to minis that are essentially accumulator machines: IBM, Honeywell, Varian, and Hewlett-Packard. With them, one operand is generally in memory, one in an accumulator, and the result admits of only three possible successor actions -

1. Do nothing.
2. Assign result to a variable.
3. Assign result to a temporary.

Such decisions can be made by a preliminary analysis, thus simplifying subsequent processing for a particular computer. Triples are preferable for the PDP and Texas Instrument instruction sets, while the Nova is unclear, but probably triples are better.

E. Crossroads

It is time now to summarize the discussion, listing only very major problems.

In part B it was noted that not all minicomputer installations will be capable of using LITTLE, and that those who can may well express strategic objections having no relation to an assessment of LITTLE itself. Furthermore, it was expected that acceptance of LITTLE will be in no small measure dependent on substantial assistance.

In part C it was first argued that processors to turn out symbolic code rather than full assemblers were preferable. Then

a number of points which demand attention but did not receive it were listed. Most important among them for minicomputers were I/O and the question of block optimization in a processor.

Part D suggested that, from the beginning, code transformation requires knowledge not only of the syntax and semantics of the VOA but also the target op set, that an intermediate, general processor can assume either too much or too little about a specific machine, and that optimization of registers is complicated by many factors.

Underlying all this is a perhaps excessively rigid unwillingness to compromise the possibilities for good code with the assets of a general approach. Yet the only sure conclusion one can draw now is that LITTLE for minicomputers is a tough problem.

In part F an approach is briefly outlined, and the section has been titled AMTRAK because:

1. The service facilities of the approach need questioning.
2. It may be better to sidetrack minicomputers and concentrate on express work for large computers.
3. Another mode of transportation may reach the destination sooner.

F. AMTRAK

A processor might be divided into the following four segments.

Segment	Complexity	Independence
1. Unresolved Code Generator	Tough	40 - 60%
2. Intermediate Resolver	Moderate	95%
3. Formatter	Small	0%
4. Encoder	Trivial	95%

The complexity is an estimate of the difficulty of writing and debugging each segment. The independence is a guess as to the portion of code that is unconcerned with any particular machine. The feasibility of the whole approach will rest almost entirely on the feasibility of the first segment. Its complexity requires a more lengthy account than the other segments, so the discussion

will begin with a statement of the generator's functions to give perspective, followed immediately by the details of the rest of the processor, and then we'll return for a more extensive examination.

Segment 1 receives control from the syntax analyzer after the processing of each subroutine and, using the VOA as input, generates a new operation list specific for a single target machine. The code in the new list lacks resolution or optimization only with reference to addressing and the operator choices determined by addressing. The generator returns control to the analyzer after each subroutine, until the last one, whereupon control passes to Segment 2.

The intermediate resolver is a table driven routine with three primary functions -- storage allocation, symbol transformation, and address resolution. Input consists of the operations list, variables from the VOA, and a table descriptive of the target computer (the 5% dependent portion of this segment). Output is a single list or file needing only minor resolution by the following format segment. The descriptive table contains special characteristics used for control of the three functions, e.g., page size, symbol properties, field length and range of addressing modes.

The storage allocation routine must provide algorithms for optimized locations of constants and variables and blocking code into pages where necessary.

The symbol transformer has the trivial task of converting LITTLE names into ones acceptable to the assembler used later.

The address resolver must take each operation, and with directions from Segment 1, choose the best address form -- relative, direct, or indirect -- and create indirects when appropriate. This is the address mode vs. intermediate code problem mentioned in part D, and is solved neatly in a recently published algorithm by Ferguson. The resolver should not be concerned with addressing forms involving immediate, indexed, or incremental modes as they are either trivial to determine or highly dependent on the context, and thus are better left to the initial generator.

This segment is generally oblivious to the semantics of operators with two exceptions -- subroutine and function calls must be designated as either resolved or not with the information passed to the formatter. The other exception is a recognition of unconditional transfers to facilitate variable and constant placement.

A later version of this resolver should perhaps attempt two more optimizing problems. First is assignment of two or more variables to the same location. Though this might seem a more appropriate function of the syntax analyzer, in some machines it would result in more indirect addressing. The other possible optimization for paged machines or machines with large relative addressing ranges would be the duplication of temporaries to reduce storage and execution time by eliminating indirects. Both these attempts would require additional information, which in the former case might very properly be an appropriate function for the analyzer. All in all, this segment (given its input) should easily produce better code than the average programmer and assembler usually do.

The formatter converts the Segment 2 output file into a symbolic representation of the ultimate program. For the most part, it acts as a simple macro expander, converting one operation in a binary format to one assembler statement. Its most difficult task in this area will be the resolution of op codes which could not be tackled until addressing was complete. For example,

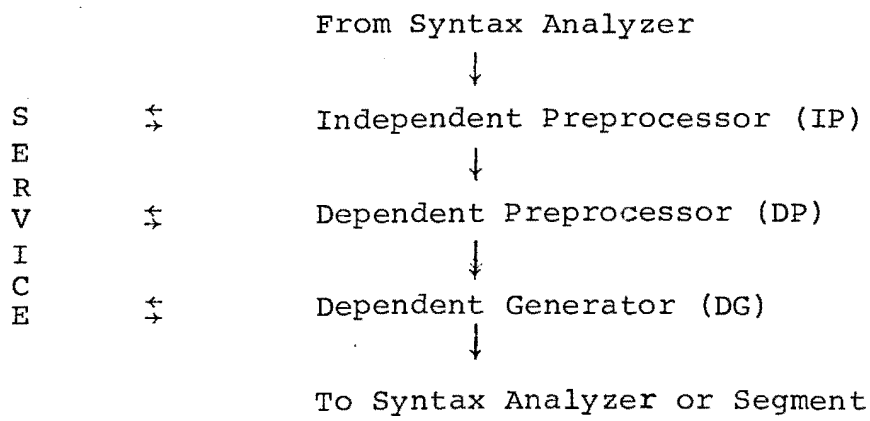
```
    JNZ  NEXT          or        SZE
                                  JMP  NEXT
(Jump if not zero)          (Skip if zero; jump)
```

This segment could undoubtedly be generalized to work from tables with machine independence, except for one other problem. At this point assembler directives and directives passed by the assembler to the loader are introduced. In any case, the formatter should be so easy to write, especially if a model for another machine is available, that independence is not an issue.

One additional function could be carried out here. A number of unsatisfied external references will exist here (e.g., an extraction operator function) and one might introduce a file of subroutines to satisfy them, yet it is unclear that this would serve any advantage over leaving the function for the loader.

The trivial Segment 4, an encoder, merely produces a file for cards, paper tape, or transmission which is acceptable to the target assembler. It uses a table with the binary representation of the target machine's characters to translate each character of the symbolic file in the large host machine. The encoder presumably terminates by transferring control to the lexical scanner. Now let us return to Segment 1.

The unresolved code generator is partitioned (not actually, just for expository reasons) into target machine dependent and independent sections. These are each subdivided. Both have preprocessing subsections; the other subsection of the independent portion is a collection of service routines which the other three subsections draw upon; the other dependent subsection performs direct, simplistic code generation. It all works like this:



The basic service functions are retrieval of information from the VOA (and associated tables) and insertion of generated code into the new operations list, but there are several other functions mentioned during the discussion of each subsection.

1. IP - Considering the thrust of this report so far, it may come as some shock that operation of the IP draws heavily on the approach in Newsletter # 4, the LITTLE imaginary machine, IM. Yet the difficulties pointed to in part D are the exceptions, not the general rule. In each of examples 2-5 there were possible choices of code for the IM that would have aided subsequent generation for the majority of minicomputers. The exceptions are considered extremely important, however, because they increase the complexity of code generation not marginally, but drastically. The solution should be apparent: turn out IM code as an alternative to, rather than a replacement for, VOA code.

Moreover, IM alternatives need neither form a complete set nor exclude redundancy. In the former category, for example, there is no possibility of improving unconditional transfers, label operations, or returns and very little possibility for subroutine and function calls, read and write statements, or the negation operator. For a quick example of the latter category, consider the two VOA entries,

```
1.  EQ    I,J          Compare I and J for equality.
2.  IF    #1,NEXT      Branch to NEXT if op. #1 is true.
```

and possible IM code,

First alternative:	Second Alternative:
1. Compare I with J	1. Subtract J from I
2. If equal, go to NEXT	2. If not zero, skip.
	3. Go to NEXT.

Some of these minis would find the first preferable, some the second, none the VOA.

Developing alternatives could easily get out of hand without some restrictions. These might include a limit on the number of new operators introduced, say 40 or 50, and allowing only expansions of a single VOA instruction and the reworking of a maximum of 2 (or 3?) sequential instructions.

Certainly not very many, because one will soon find the IM code tied to a single machine: the IP has become a DP.

Of course not every alternative need exist in a specific implementation; that would involve useless processing. The point is, however, to spend some initial effort designing IM code that has a good probability of accomplishing the most important tasks in a wide variety of minicomputers. Then after the seçond or third machine has been treated, there is essentially no work necessary on this subsection.

The astute observer may be thinking well, don't we have a charade going on here now. Why, if a set of alternatives is superior for a given machine, shouldn't one just call them and treat them as replacements, and be done with it? The answer is that there is no surety of superiority until the code environment is examined. For a simple example, take a LITTLE assignment statement, A = B. The single VOA entry might be processed to give a two-member alternative, but for the PDP-11,

```
LD      B → register
ST      register → A
```

the dependent generator, DG, would never, by itself, use this alternative, since the instruction set includes this direct correlative of the VOA entry,

```
MOV     B,A  .
```

On the other hand, the dependent preprocessor, DP, described below, might well decide on the basis of subsequent use of either A or B that the alternative is preferable, since it may save one or more machine cycles. In several of the minicomputers the choice might be reversed for commutative operators. That is, where the DP would choose the expanded alternatives, the DPs would process the VOA with greater ease when the environment calls for an operation reversal,

```
A op B  →  B op A  .
```

The same choice will usually be made by DPs in more complex optimizations, because in general the VOA entries will be semantically more inclusive.

To wind up discussion of the IP, note that the only specific function required of the service subsection is recognition of singlet, doublet and perhaps triplet code sequences. Finally, the structure of the IP must be that of sequentially executable routines to facilitate conversion of one implementation to another.

2. DP - Whereas the IP was predominantly involved with semantic manipulations, the dependent processor focuses on syntactic problems. Its simplest functions would be like those mentioned just above. Intermediate functions include register allocation and compile time calculations (e.g., in part D, third example, Honeywell code, the first five instructions should never be executed). More advanced functions would be the transformation envisioned in example 1, part D and the optimization suggested for the FORTRAN-like loop in part C.

A major characteristic of the DP is that in a first version of any target processor, it may be null. The result, naturally, will be lousy code.

Since DP development can be put off, here there will only be brief mention of service requirements and general structure.

The service needs of the IP are duplicated here, but are also extended to include recognition of larger sequential and scattered code patterns. Additionally the service subsection must be capable of answering questions concerning the frequency of reference to variables and constants, perhaps providing general algorithms for register assignment, setting up indirect triple tables to simplify reordering, carrying out the subsequent restructuring of the VOA, inserting new alternatives created by the DP, and finally marking DP output as obligatory input to DG.

DP structure should receive more thought than the other subsections as the complexity might argue for a tree-like or even a recursive form, but a simple, sequential set of routines would have at least one benefit. The DP is largely tied to a specific machine, but at least some optimizations will be nearly

identical for several. Going back again to example 1, part D we have a clear case. Just as with the IP, portions of a sequential DP are more easily stolen for transport to a new implementation.

3. DG - The generator is simple. It requests one operation at a time from the service routines. Each operation is converted to one or more target ops. The toughest task it has is to choose between VOA entries and alternatives. It only looks ahead to ask whether the result of the current operation is referenced again, and when. Even this question can be eliminated for some DGs if the service routines can supply quadruples rather than triples. The DG supplies service routines with the new operations in two parts -- one headed for the intermediate resolver, Segment 2, and one for the formatter, Segment 3. The information it must supply to each segment is implicit or explicit in previous discussion.

The only additional service function not mentioned in the accounts of IP, DP and DG that should be present is debugging code to aid new implementations of the dependent subsections.

In conclusion

It is clear that the approach outlined here, separating independent and dependent tasks into sections and subsections, will require substantially more effort than writing an assembler or processor for a single minicomputer. This effort should be balanced against the arguments given in part B.

Perhaps a gross estimate of the time involved would include 2 months for detailed design of the processor, 4 to 5 man-months for the first application (excluding DP), and about $1\frac{1}{2}$ - 2 months for each new implementation.

* * *