This newsletter discusses compiled SETL code style for the 6600 and the 370 and the microcode for the PUMA by defining the library linkages and general code style and giving rough timing estimates. The code is in assembler (or microcode in the case of the PUMA). Since access to different words and fields can be done quite differently at this level, to give symbolic names to offsets in the code would misleadingly imply that to change fields involves just a change in the definitions of the names. We will assume the SETL data structures as of the beginning of May and write the field names in the comments.
6600, 370, and PUMA Microcode Nubbins

6600 Nubbins

A. 6600 Nubbins.

This case is the simplest. The basic design considerations are as follows:

i) Not all nubbins are inline since some are too large.

ii) For offline nubbins at least a jump offline and a jump back are required and a certain amount of load-store work can be done in parallel with these jumps. (Actually, we will not be able to avoid a third jump.)

iii) Since library calls can occur, only registers that are used in highly stereotyped ways in the LITTLE SRTL code can be used for other than very temporary uses.

The last consideration will be addressed first. We can use X0 to contain TVALMASK. This register is normally unused by the LITTLE compiler but the compiler can be modified to use that register when the appropriate mask is required. B1 will, as required by the LITTLE system, contain the constant one. B2 will contain the address of the heap. (Actually HEAP-1.)

The first two considerations suggest a 3-address style. The inputs of the operation will normally be loaded into X4 and X5 and the output will be placed into X6. This gives the following form for a call to an offline nubbin:

\[
\begin{align*}
S14 & \quad \text{ARG1} & \quad \text{Load first arg.} \\
S15 & \quad \text{ARG2} & \quad \text{Load second arg.} \\
S16 & \quad \text{RESULT} & \quad \text{Store previous result.} \\
RJ & \quad \text{WUB} & \quad \text{Call nubbin.}
\end{align*}
\]

This occupies 2 words and takes about 2.3 microseconds.

If the result of the first operation is a "temporary" to be used immediately it need not be stored and reloaded; instead we can jump to a point at which an appropriate copy is performed. This leads to the "short form" call which is either:

\[
\begin{align*}
S14 & \quad \text{ARG1} \quad \text{or} \quad S15 & \quad \text{ARG2} \\
RJ & \quad \text{WUB1} & \quad \text{or} \quad RJ & \quad \text{WUB2}
\end{align*}
\]

This occupies one word and takes about 1.5 microseconds.

Since most of the time is spent in the RJ instruction, in a few favorable cases of short nubbins inline code may be generated.

A typical example of an offline nubbin is the following multiplication sequence:

\[
\begin{align*}
\text{MULT} & \quad \text{BSS} & \quad 1 & \quad \text{Entry word.} \\
& \quad BX1 & \quad -X0*X4 & \quad \text{Get type, value for first arg.} \\
& \quad SX7 & \quad 377777B & \quad \text{Get largest short integer.} \\
& \quad BX2 & \quad -X0*X5 & \quad \text{Get type, value for second arg.}
\end{align*}
\]
6600, 370, and PUMA Microcode Nubbins

6600 Nubbins

IX3  X1-X7  See if arg. 1 too large.
IX6  X1*X2  Do multiply.
IX2  X2-X7  See if arg. 2 too large.
IX7  X6-X7  See if result is too large.

B13  X3*X2  Now AND together the ... three test values.
B13  X3*X7  ... three test values.
B16  X3,MULT  Done if all in range.

SB3  =XLIBMUL  Else get library address.
RJ  LIBLINK  Branch to library.
RQ  MUL  Return upon exit from library.

This takes about 4.7 microseconds.

The LIBLINK sequence which is used to link to the LITTLE-written library is as follows:

LIBLINK  BSS  1  Entry word.
  BX6  X4  Copy first argument.
  SA1  P1  Point to parm. list and first parm.
  BX7  X5  Copy second argument.
  SA2  LIBLINK  Get entry word.
  SA6  X1  Store first argument.
  SA7  X1+B1  Store second argument.
  BX6  X2  Copy entry word.
  SA6  B3  Store at branch location.
  JP  B3+1  Branch to library routine.
  P1  CON  T1  First argument address.
  CON  T1+1  Second argument address.
  BSS  2  Space for the two arguments.

This takes about 3.7 microseconds to call the library.

Op-codes that merely call a library routine can have the following three-word inline "long form":

SA4  ARG1  Get first argument.
SA5  ARG2  Get second argument.
SA6  RESULT  Store previous result.
SB3  =Entry  Get appropriate entry point.
RJ  LIBLINK  Go call library.

Thus, nubbins are not required for these cases, which are fairly numerous. Of course, two-word forms are available if X6 need not be stored.

The simplest SETL jumps are compiled as inline tests. Tests, such as the general equality test, which may involve library code, can have the following treatment:

SA4  ARG1  Load first argument.
SA5  ARG2  Load second argument.
6600, 370, and PUMA Microcode Nubbins

6600 Nubbins

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BX1</td>
<td>X4-X5</td>
</tr>
<tr>
<td>SA6</td>
<td>RESULT</td>
</tr>
<tr>
<td>BX2</td>
<td>-X0*X1</td>
</tr>
<tr>
<td>ZR</td>
<td>X2,JUMPA DR</td>
</tr>
<tr>
<td>AX1</td>
<td>51</td>
</tr>
<tr>
<td>SX3</td>
<td>7</td>
</tr>
<tr>
<td>AX2</td>
<td>51</td>
</tr>
<tr>
<td>IX1</td>
<td>X1-X3</td>
</tr>
<tr>
<td>SB3</td>
<td>=XLIBEQUIV</td>
</tr>
<tr>
<td>IX2</td>
<td>X2-X3</td>
</tr>
<tr>
<td>BX1</td>
<td>X1*X2</td>
</tr>
<tr>
<td>MG</td>
<td>X1,LAB</td>
</tr>
<tr>
<td>RJ</td>
<td>LIBLINK</td>
</tr>
<tr>
<td>NZ</td>
<td>X6,JUMPA DR</td>
</tr>
</tbody>
</table>

LAB

This in-line code sequence is 7 words long and takes about 6.5 microseconds in the worst case in which the library is not called. Note that a word (and a few minor cycles) can be saved if X6 was an argument which did not have to be stored.

In-line addition is as follows:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SA4</td>
<td>ARG1</td>
</tr>
<tr>
<td>SA5</td>
<td>ARG2</td>
</tr>
<tr>
<td>SA6</td>
<td>RESULT</td>
</tr>
<tr>
<td>IX3</td>
<td>X4*X5</td>
</tr>
<tr>
<td>BX6</td>
<td>-X0*X5</td>
</tr>
<tr>
<td>BX7</td>
<td>X6</td>
</tr>
<tr>
<td>AX7</td>
<td>51</td>
</tr>
<tr>
<td>ZR</td>
<td>X7,LAB</td>
</tr>
<tr>
<td>SB3</td>
<td>=XLIBADD</td>
</tr>
<tr>
<td>RJ</td>
<td>LIBLINK</td>
</tr>
</tbody>
</table>

LAB

This is four words long and takes about 4.4 microseconds.

The in-line subtraction nubbin is as follows:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SA4</td>
<td>ARG1</td>
</tr>
<tr>
<td>SA5</td>
<td>ARG2</td>
</tr>
<tr>
<td>BX1</td>
<td>X4*X5</td>
</tr>
<tr>
<td>IX2</td>
<td>X4-X5</td>
</tr>
<tr>
<td>SA6</td>
<td>RESULT</td>
</tr>
<tr>
<td>BX1</td>
<td>X1*X2</td>
</tr>
<tr>
<td>BX6</td>
<td>-X0*X2</td>
</tr>
<tr>
<td>BX1</td>
<td>-X0*X1</td>
</tr>
<tr>
<td>AX1</td>
<td>51</td>
</tr>
<tr>
<td>SB3</td>
<td>=XLIBSUB</td>
</tr>
<tr>
<td>ZR</td>
<td>X1,LAB</td>
</tr>
<tr>
<td>RJ</td>
<td>LIBLINK</td>
</tr>
</tbody>
</table>
6600, 370, and PUMA Microcode Nubbins
6600 Nubbins

LAB

This occupies 5 words and takes about 5.1 microseconds.

As a final example, we consider the case of remote map retrieval by a quantity known to be a pointer to the relevant base. This can be done in an inline sequence as follows:

\[
\begin{align*}
& + \quad \text{SA5 ARG2 Load second argument.} \\
& + \quad \text{SA4 ARG1 Load first argument.} \\
& + \quad \text{SB4 X5+B1 Prepare HEAP (VALUE (ARG2) + OFF_EBINDX).} \\
& + \quad \text{SA1 B4+B2 Load the above word.} \\
& + \quad \text{SB5 X4+2 Prepare to get MAXINDX.} \\
& + \quad \text{SA2 B5+B2 Load maximum index word.} \\
& + \quad \text{LX1 -18 Extract EBindX field.} \\
& + \quad \text{SA6 RESULT Store last result.} \\
& + \quad \text{MX3 -45 Get mask.} \\
& + \quad \text{BX3 -X3*X1 Get EBINDX(ARG2).} \\
& + \quad \text{LX2 -18 Position MAXINDX.} \\
& + \quad \text{SB3 X3+B1 Copy index to B-register.} \\
& + \quad \text{SB2 X2+B1 Extract MAXINDX.} \\
& \text{LE B3,B2,SKIP Skip next set if index in range.} \\
& + \quad \text{SB3 B1 Else set index to zero.} \\
& \text{SK1P} \\
& + \quad \text{SB3 B3+B5 Prepare to load result.} \\
& + \quad \text{SB3 B3+B2 Get HEAP address - 1.} \\
& + \quad \text{SA1 B3+B1 Add tuple header length; load result.} \\
& + \quad \text{B16 X1 Get result.}
\end{align*}
\]

This occupies 7 words and takes about 6.5 microseconds. This, in fact, may be too long to do inline. If it were done offline, the code would be modified to put the SKIP label before the entry word.

Note that 4-6 microseconds is a typical time for these important nubbins. Thus, code that never needs to enter the library should run at approximately 1/5 - 1/10 the speed of corresponding code generated by a reasonably good FORTRAN compiler.
6600, 370, and PUMA Microcode Nubbins.

370 Nubbins.

B. 370 Nubbins.

The structure of these nubbins is different from those for the 6600 for two major reasons:

i) Jumps are a lot less expensive than on the 6600.

ii) Registers are saved across library calls so a simple register allocator could be used.

This leads to the following design:

i) All nubbins are offline and entered with a BAL instruction.

ii) Some registers will be reserved for scratch registers within the nubbins.

iii) Other registers will be used to contain needed constants and base locators.

iv) The rest of the available registers can be allocated by the generated code to reduce the number of loads and stores.

The register usage is as follows:

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0 (A1)</td>
<td>First input to nubbin and return value.</td>
</tr>
<tr>
<td>R1 (A2)</td>
<td>Second input to nubbin.</td>
</tr>
<tr>
<td>R2 (AH)</td>
<td>Address mask (X'00FEFFFC')</td>
</tr>
<tr>
<td>R3 (TVH)</td>
<td>Type/value mask (X'FFEFFFC')</td>
</tr>
<tr>
<td>R4 (HEAP)</td>
<td>Base register pointing to HEAP-1.</td>
</tr>
<tr>
<td>R5</td>
<td>Base register for offline nubbins.</td>
</tr>
<tr>
<td>R6</td>
<td>Base registers for labels.</td>
</tr>
<tr>
<td>R7 (LBL)</td>
<td>Allocatable but used for jump address in tests.</td>
</tr>
<tr>
<td>R8-R12</td>
<td>Allocatable.</td>
</tr>
<tr>
<td>R13 (WA)</td>
<td>Scratch for nubbins.</td>
</tr>
<tr>
<td>R14</td>
<td>Return address from nubbins.</td>
</tr>
<tr>
<td>R15 (WB)</td>
<td>Scratch for nubbins.</td>
</tr>
</tbody>
</table>

A "worst case" call to a nubbin when everything must be loaded and stored would be as follows:

```
ST    A1,RESULT  Store last result.
L     A1,ARG1    Load first argument.
L     A2,ARG2    Load second argument.
BAL   R14,NUB    Call the nubbin.
```

This occupies 16 bytes. (Note that we will not attempt to give timings because of the large number of models and submodels.)

In a better (and more typical) case where items are in registers, the code is as follows:

```
LR    A2,R11    Second arg. (first was output)
BAL   R14,NUB   Go call nubbin.
```
6600, 370, and PUMA Microcode Nubbins.

370 Nubbins.

Thus requires only 6 bytes. Note that, unlike in the 6600 case, both BAL's are to the same location.

To call a library routine, a nubbin sets R15 to the entry point of the routine and branches to LIBLINK which is shown below.

LIBLINK
STM A1,A2,ARGS Store arguments.
LA R13,SAVEAREA Point to save area.
LA R1,PLIST Point to parameter list.
BR R15 Call routine; it returns inline.

PLIST
DC A(ARGS,ARGS+4) Parameter list.
ARGS
DS 2F Space for arguments.
SAVEAREA
DS 16F Standard OS save area.

We will now present the 370 code for the nubbins shown in the 6600 section.

First, multiplication:

MULT
MR A1,A1 Do the multiply.
LTR A1,A1 See if too large or not integers.
BNZ LMULT Go offline if so.
SLDL A1,30 Else position result.
NR A1,TVH Mask out junk bits.
CLR A1,AM See if too large.
BNHR R14 Return if not.

LMULT
L R15,=A(LIBMULT) Else get library address.
B LIBLINK Now call library.

For the branch cases, the inline code must load the address of the "true" label into register LBL and then call the nubbin. We will show the case of the equality test nubbin below.

EQUV
NR A1,TVH Remove junk from ...
NR A2,TVH ... both inputs.
CLR A1,A2 Compare both inputs.
BER LBL Branch if equal.
SRL A1,24 Now get type codes ...
SRL A2,24 ... for both inputs.
LA WA,6 T_LATON.
CR A1,WA If greater, go offline.
BH LEQUV Go call library.
CR A2,WA If other type is OK, not equal.
BNHR R14 So return FALSE.

LEQUV
STM A1,A2,ARGS Else save arguments.
LA R1,PLIST Point to parameter list.
LA R13,SAVEAREA Point to save area.
ST R14,RET Save return address.
L R15,=A(LIBEQUV) Get library routine address.
BALR R14,R15 Call library.
LTR RO,RO Test return value.
6600, 370, and PUMA Microcode Nubbins.

370 Nubbins.

```
BNZR LBL RET
L R14,RET
BR R14
RET

Next, addition:

ADD AR A1,A2
NR A1,TVM
CLR A1,AM
BNHR R14
L R13=A (LIBADD)
B LIBLINK

Next, subtraction. This is similar to addition except that the types must also be checked before the actual operation.

SUB NR A1,TVM
NR A2,TVM
CLR A1,AM
BH LSUB
CLR A2,AM
BH LSUB
SR A1,A2
BNHR R14
L R5=A (LIBSUB)
B LIBLINK

Finally, we present the case of remote map retrieval below.

OPRSN NR A2,AM
LR WA,16(A2,HEAP)
NR A2,A1
CH WA,12(A2,HEAP)
BH SKIP
SR WA,WA
SLL WA,2
AR A2,WA
L A1,16(A2,HEAP)
BR R14

```

Return if equal.
Else load old return address.
Return FALSE.
Save location for return address.

Do addition.
Remove junk bits.
See if type still OK.
Return if OK.
Else get library routine.
Go call library.

Remove junk bits ...
...
from both inputs.
See if in range.
Go offline if not.
Check second input.
Branch if not in range.
Now subtract.
Return if not negative.
Get library address.
Go call library.

Get offset from start of heap.
Load EBINDX.
Get first arg. addressable.
Get value only.
Compare with MAXINDX.
Index in range.
Else set index to zero.
Get correct offset.
Get HEAP offset - 4.
Load result value.
Now return.
C. PUMA Microcode Nubbins.

This case is entirely different because we are dealing with a microprogrammable machine. Grossly described, what we intend to do is to emulate both the "normal" 6600 instructions (and maybe add a few for efficiency) and special SETL instructions and have the microcode handle the state switching.

These SETL instructions will correspond to calls to nubbins in the above two cases. If the nubbins do not require a call to the library, they can be done by the microcode in a manner similar to the way the microcode would execute a 6600 instruction. If the nubbins required a library call, a microcode sequence would be entered to call the library. The library would execute a special instruction to return to the SETL mode and set the result.

The PUMA has, in addition to the X, A, and B registers, 8 60-bit Y registers. These registers are used in the normal 6600 emulation as scratch registers but if we could restrict their usage as scratch registers, they could be used as registers in the "SETL machine" mode. In fact, the only place where more than one or two of the Y registers are currently used is in the multiply routine. If we were to accept a multiply which is 3 times slower, we could have the rest of the Y registers free for the SETL instructions and they would persist over the LITTLE-written library.

In addition, we need a register to hold, in 6600 mode, the return point to SETL mode. We can use Y1 for this register. That means that Y0 can be used as the scratch register in 6600 mode and those few places where a second scratch register is needed can be re-written to use only one. That leaves Y2-Y7 as registers for the SETL instructions which will persist over the library calls. We can use the X registers as scratch in SETL mode so that Y0 and Y1 can be used for data that need not persist over library calls.

The SETL instructions will have a format similar to the normal 6600 instructions. The op-code and I fields of the 6600 instruction will be used for the SETL op-code and the J and K fields will be used as usual. Bit 1 of the op-code will be the library flag. If it is on, it means that this operation is merely a call to the library and no processing can be done by microcode. This means that the microcode can simply call the library directly without having to have special code for that operation. The low-order bit of the op-code is used for operation sub-types. For binary operations which return an output this bit is used to indicate which register receives the output. If it is on, Y7 receives the output; otherwise, Y6. In other cases, it is used to differentiate such things as branch TRUE/FLASE, load/store, and give two related op-codes when there is no need for three registers. Note that branches, loads, and stores will use the long form of the instructions which is the same as for 6600 instructions.

We will have the global register usage over both 6600 and SETL modes the same as for the 6600 nubbins above. Namely, X0 will hold TVALMASK, B1 will hold the constant 1, and B2 will contain the address of HEAP(0).
There will be a table of entry points to the library at memory locations known to the microcode. At each entry point will be a special program-stop instruction which will return control to SETL mode in a case where the library routine would normally return. This instruction can be placed at the entry word by an initialization routine. Note that it is assumed that the called routine does not do funny things with its entry word other than branch to it to return. This is the case in LITTLE-written code and COMPASS routines are not supposed to do things with this word in any event.

When the microcode wants to call the library, it places into E1 the main storage address of the entry point to which it desires to branch and jumps to micro-instruction LIBLINK which is is shown below. It builds, in Y1, a value containing the parcel count into the current instruction word, the address of the current word (which is P+1 by this time), and the value to which it will branch. The latter is used for safety as follows. When a "return to SETL" instruction is encountered, it must only occur at one minus the last jump point taken to the library. Thus, the P value at that time must agree with the branch address stored in Y1. LIBLINK will also save the two input arguments into a parameter list and set A1 and X1 according to the normal calling conventions (A1 contains the address of the parameter list and X1 contains the address of the first parameter).

LIBLINK P=P-1; E0=E1; IF ~NIWEMPTY THE LLNOWRD
* An instruction fetch is in progress. Wait it out.
LLNOWRD CLEAR; AC=EO; E0=PLISTaddr; IF CDONE THEN LLNOWRD
MA=AC; READ; AC=EO; E0=T1addr; * Read up branch address.
A1=AC; AC=EO * Set parm. list address; set store address.
X1=AC; AC:MQ=SHIFT(P:MQ, R16) * Start shifting P value.
MQ=SHIFT(AC:MQ, R16) * Continue shift.
LLNOWRD CLEAR; AC=EO; E0=T1addr; IF CDONE THEN LLNOWRD
LLWAIT MIW=CHRD; IF ~CMDONE THEN LLWAIT * Read to next inst. word.
LLNOWRD CLEAR; AC=EO; E0=PLISTaddr; IF CDONE THEN LLNOWRD
MA=AC; READ; AC=EO; E0=T1addr; * Read up branch address.
A1=AC; AC=EO * Set parm. list address; set store address.
X1=AC; AC:MQ=SHIFT(P:MQ, R16) * Start shifting P value.
MQ=SHIFT(AC:MQ, R16) * Continue shift.
LLNOWRD CLEAR; AC=EO; E0=T1addr; IF CDONE THEN LLNOWRD
LLWAIT MIW=CHRD; IF ~CMDONE THEN LLWAIT * Read to next inst. word.
LLNOWRD CLEAR; AC=EO; E0=PLISTaddr; IF CDONE THEN LLNOWRD
MA=AC; READ; AC=EO; E0=T1addr; * Read up branch address.
A1=AC; AC=EO * Set parm. list address; set store address.
X1=AC; AC:MQ=SHIFT(P:MQ, R16) * Start shifting P value.
MQ=SHIFT(AC:MQ, R16) * Continue shift.
LLNOWRD CLEAR; AC=EO; E0=T1addr; IF CDONE THEN LLNOWRD
LLWAIT MIW=CHRD; IF ~CMDONE THEN LLWAIT * Read to next inst. word.
LLNOWRD CLEAR; AC=EO; E0=PLISTaddr; IF CDONE THEN LLNOWRD
MA=AC; READ; AC=EO; E0=T1addr; * Read up branch address.
A1=AC; AC=EO * Set parm. list address; set store address.
X1=AC; AC:MQ=SHIFT(P:MQ, R16) * Start shifting P value.
MQ=SHIFT(AC:MQ, R16) * Continue shift.

This takes about 1.31 microseconds. We will assume in timing estimates for the PUMA that a cycle is 45ns and memory cycle is 470ns.

We will now present the new microcode for the program stop instruction which will process the special "return to SETL" instruction. We will
6600, 370, and PUMA Microcode Nubbins.

PUMA Microcode Nubbins.

Assume that we are using an I field of one to indicate this instruction and that, for clarity, no other sub-types of program stop exist.

L00

\[ E0=2000; \text{IF} -I(0) \text{THEN ERROR} \] * Process normal PS. 
\[ E1:BUP=Y1; AC=P; \text{IF} I(1) \text{THEN ERROR} \] * Get return word. 
\[ (AC)=AC-BUP[18]; \text{IF} I(2) \text{THEN ERROR} \] * Start address check. 
\[ AC=AC-BUF[18]; \text{IF} -WNEMPTY \text{THEN LO0SK} \] * Continue test. 

LO0WT1

IF ~CMDONE THEN LO0WT1 * Wait out instruction fetch. 
\[ WN=CHR; \text{CLEAR} \] * Clear out instruction fetch. 

LO0SK

IF ~AC=0 \text{THEN ERROR} \] * Finish validity check. 
\[ AC=SHIFT(BUF:MQ, R16); BUF=X6; E0=2; CIW=WIN 

LO0SLP

AC=SHIFT(AC:MQ,R4); E0=E0-1[F]; IF EALU(0)&EALU(1) THEN LO0SLP 

LO0WT2

P=AC; IF CMDONE THEN LO0WT2 * Wait for free memory. 

LO0WT3

MA=P; READ; NIW=CHR; IF ~CMDONE THEN LO0WT3 
\[ CIW=WIN; AC=BUP; CLEAR; P=P+1; LATCH I \] * Get to return word. 
\[ =3-E1; MA=P; \text{READ}; \text{IF} EALU \text{THEN RETOUT} \] * RNI; test pos. 

LO0PLP

NEWPARCEL; LATCH I; E1=E1+1[F] * Get to correct position. 
\[ =3-E1; \text{IF} EALUP \text{THEN RETOUT} \] ELSE LO0PLP * See if done. 

This takes about 1.35 microseconds.

Next we will present the microcode which handles the return from a binary operation with a result to give an idea of the type of housekeeping needed.

RETOUT NEWPARCEL; LATCH I; IF I(0) THEN ROUT7 
ROUT6 Y6=AC; IF ICHECK THEN SICHECK ELSE SOPCODEBRANCH 
ROUT7 Y7=AC; IF ICHECK THEN SICHECK ELSE SOPCODEBRANCH 

This takes 90ns and will be included in the timings of the nubbins which jump to it (although the other overhead operations will not be included).

Next we present the microcode for some of the simpler operations. First, addition and subtraction:

ADD

\[ BUF=Y1 \] * Get first input. 
\[ AC=BUP; BUF=YK \] * Copy first; get second input. 
\[ (AC)=AC+BUP \] * Start addition. 
\[ AC=AC+BUP; BUF=X0; E1=LIBADD \] * Finish add; get mask 
\[ AC=AC8-BUF; E0:X6=AC \] * Do mask and set to check type. 
\[ =370+E0; \text{IF} EALUP \text{THEN RETOUT} \] ELSE LIBLINK * Complete.

SUB

\[ BUF=Y1 \] * Get first input. 
\[ AC=BUP; BUF=YK \] * Copy first; get second input. 
\[ (AC)=AC-BUP; E0:X6=AC \] * Start subtract; get check type. 
\[ AC=MQ; MQ=AC-BUF; =370+E0; \text{IF} -EALUP \text{THEN LIBLINK} \n\[ AC=BUP; BUF=X0 \] * Get second input; get TVALMASK. 
\[ E0:X6=AC; AC=MQ \] * Check second type; get subtract result. 
\[ AC=AC8-BUF; =370+E0; \text{IF} -EALUP \text{THEN LIBLINK} \] * Mask; check.
6600, 370, and PUMA Microcode Wubbins.

PUMA Microcode Wubbins.

E0:X6=AC
   * Get check type for output.
=370+E0; IF EALUP THEN RETOUT ELSE LIBLINK * Done.

Addition takes .36 microseconds and subtraction takes .50 microseconds.

Next, we present the case of remote map retrieval shown above. It will be very helpful in understanding the microcode below to refer to the COMPASS code for the same routine above.

```plaintext
SOFRSN BUF=YK; AC=MQ; MQ=0  * Get second input; clear MQ.
   AC=BUF; BUF=B2; E0=2; IF ~NIWEMPTY THEN SOFRWT1

SOFRWTO NIW=CMRD; IF ~CMDONE THEN SOFRWT0  * Wait for fetch.

SOFRWTO CLEAR; (AC)=AC+BUF[18][NOP]; IF CMDONE THEN SOFRWT1
   AC=AC+BUF[18][NOP]; BUFYJ  * Get addr. EBINDX; get first arg.
   MA=AC; READ; AC=E0  * Read EBINDX word; set AC to 2.
   (AC)=AC+BUF[18]  * Prepare to get HEAP offset of MAXINDX.
   AC=AC+BUF[18]; BUF=B2  * Finish add; get HEAP address.

SOFRWTS (AC)=AC+BUF[18]; IF ~CMDONE THEN SOFRWT2  * Wait for read.
   AC=AC+BUF[18]  * Get address for MAXINDX.

SOFRWTS CLEAR; MA=AC; X6=AC; IF CMDONE THEN SOFRWT3  * Wait for mem.
   AC=CMRD; READ  * Get EBINDX word; start read of MAXINDX.
   X5=AC; AC=MQ=SHIFT(-1:MQ,R16)  * Save word; build mask.
   AC=MQ=SHIFT(AC:MQ,L1); BUF=X5  * Cont. with mask; get word.
   X5=AC; AC=SHIFT(BUF:MQ,R16)  * Save mask; shift data.
   AC=SHIFT(AC:MQ,R1); BUF=X5  * Cont. shift; get mask.
   AC=SHIFT(AC:MQ,R1)  * Finish shift.
   AC=AC+BUF  * Extract EBINDX.

SOFRWTS X5=AC; IF ~CMDONE THEN SOFRWT4  * Save EBINDX; wait MAXINDX.
   AC=SHIFT(CHR:NQ,R16); CLEAR  * Position MAXINDX.
   AC=SHIFT(AC:MQ,R1)  * Continue shift.
   AC=SHIFT(AC:MQ,R1)  * Mask out MAXINDX.
   (AC)=AC+0[18]  * Start range test.
   AC=AC+BUF[18]; BUF=X5  * Complete; get EBINDX
   (AC)=AC-BUF[18]  * Else use index of zero.
   BUF=B0

SOFRSKP AC=BUF; BUF=X6
   (AC)=AC+BUF[18][NOP]  * Get index and address.
   AC=AC+BUF[18][NOP]; BUF=B2

SOFRWTS (AC)=AC+BUF[18][NOP]; IF CMDONE THEN SOFRWT5
   AC=AC+BUF[18][NOP]  * Get address to load.
   MA=AC; READ  * Read result value.

SOFRWTS IF ~CMDONE THEN SOFRWT6  * Wait for read to complete.
   AC=CMRD; CLEAR; NEWPARCEL; LATCH I; IF I(0) THEN ROUT7 ELSE ROUT6
```

This takes about 2.61 microseconds.

For tests of one input that input is given in the J field of the branch instruction. For tests of two inputs, a "compare" instruction is used and a "condition code" is saved in the output register. Then, a
6600, 370, and PUMA Microcode Nubbins.

PUMA Microcode Nubbins.

"branch on condition" is used to branch TRUE or FALSE depending on the "condition code" stored in its input register. As an example of the two-input test, we present the equality test which was shown earlier in 6600 and 370 assembly code.

EQUV BUF=YJ * Get first input.
   AC=BUF; BUF=YK * Copy first input; get second input.
   AC=AC/BUF; BUF=XO * Do exclusive OR; get TVALMASK.
   AC=AC-E-BUF; EO:BUF=YK; E2=1 * Mask; get type; get TRUE.
   =67-E0; E1:BUF=YK; IF AC=0 THEN TRUE * Check; get type; true
   =67-E0; IF EALU(11) THEN LIBLINK * This is long type.
   =67-E1; BUF=BO * Start other test; get FALSE.
   =67-E1; AC=BUF; IF EALU(11) THEN LIBLINK ELSE RETOUT

TRUE AC=E2; NEWPARCEL; LATCH I; IF I(0) THEN ROUT7 ELSE ROUT6

This takes about .50 microseconds.

Therefore, we see that the simpler nubbins which do not do memory accesses are about 10-11 times faster than those running on the 6600 and those which access memory or involve library linkages are about 3 times faster.