**Porting the UNIX Implementation of Icon; Version 5.10***

*William H. Mitchell*

TR 85-20a

*ABSTRACT*

This document explains how to port the UNIX implementation of Version 5.10 of the Icon programming language. The Icon system is composed of a translator, a linker, and a run-time system. Procedures for porting each system component are described in detail. This document is meant to be a companion to the Icon "tour" (TR 85-19) and the source code for the system.

August 31, 1985; Revised October 21, 1985

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant DCR-8401831.

**Porting the UNIX Implementation of Icon; Version 5.10**

## 1. Introduction

This document describes how to port the Version 5.10 Icon interpreter to a UNIX* environment. The Icon system has three major components: a translator, a linker, and a run-time system. The translator and the linker are entirely written in C and porting them is primarily a matter of setting constant values that are appropriate for the target machine. Portions of the run-time system are written in assembly language and thus must be written anew for each machine. The run-time system also contains a very small amount of C code that must be written on a per-machine basis.

The sections of this document that describe the porting of the translator and the linker are straightforward, being merely a description of a process. While porting the translator and the linker are tasks of following instructions, porting the run-time system is a task of design and programming. The approach taken is to describe what function each routine must perform and how it is implemented in the VAX† version of Icon. The porter's job is to determine how to implement the various routines on the target machine.

This document is a companion document of the Icon "tour"[1] and should be studied with the source code for Version 5.10 of Icon at hand. In particular, the porter should be familiar with the information contained in the tour.

The sections of this document that describe the VAX assembly language code attempt to explain the operation of instructions when the operation is not obvious. However, this document does assume that the porter has a rudimentary familiarity with the basic concepts of the VAX-11 architecture[2].

## 2. Software Requirements

Icon has been ported to about a half-dozen systems thus far and for those ports the C compilers encountered have been able to accommodate the system source code without difficulty. A "production quality" C compiler is the basic requirement.

In addition to fundamental reliability, the compiler must support both assignment and call-by-value for structures.

An implicit assumption in this implementation of Icon is that C integers and pointers are of the same size. This is because the algorithms that access run-time data structures were designed with the assumption that these data structures are composed of a number of words, some of which may hold integer values and some that may hold pointer values. It may be possible to port this implementation of Icon using such a C compiler, but no serious investigation of the feasibility of such has been made.

Machines whose stacks grow upward rather than downward (that is, toward larger memory addresses) present additional complications that are explained later on.

Machines that have non-conventional stacks, for example, those that map the top of the stack into a register bank, may present a considerable challenge.

In light of the increasing popularity of the C language and the availability of C compilers for non-UNIX environments, ports of Icon to non-UNIX environments may be attempted. Because the matter of porting a UNIX program to a non-UNIX environment is a problem in itself, it is not addressed in this document. Rather, this document assumes that the target environment is UNIX. This is not to say that porting Icon to a non-UNIX environment is not feasible. Icon is not strongly bound to UNIX, the primary association being that Icon is written in C. Most C systems that are available for non-UNIX environments provide most of the

*UNIX is a trademark of AT&T Bell Laboratories

†VAX is a trademark of Digital Equipment Corporation.

UNIX-independent C standard functions as part of a library. If such a library is available, it should be possible to port Icon without great difficulty.

### 3. Porting Overview

Porting Icon starts out in the same way as installing an existing implementation of Icon. Read the documentation on the installation process [3] before beginning a port.

It is important to understand the organization of the Icon system. See the Appendix, which shows the major components of the Icon file hierarchy. The portions related to source code (src) are particularly important.

At a point in the installation process, the installer becomes a porter and provides the code needed for the new computer. The porter then becomes an installer again and completes the port by completing the installation. The instructions in [3] should be carried out up to Section 1.4 (*Configuring the Icon System*). At this point, select a name associated with the computer to which the port is being made. This name should be brief, suitable for a file name and for a *cpp* symbol. For example, the VAX-11 implementation uses vax and the Ridge 32 implementation uses ridge. In the root directory, referred to as v5, configure Icon using

make Setup-port HOST=*name*

where *name* is selected name in lowercase.

This command runs Icon-setup with appropriate parameters. One of the actions taken by Icon-setup is to copy prototypes of machine-dependent files from src/proto to src/sys. The porter's work is basically confined to the files in src/sys. In lieu of qualification, references to source files in this document refer to files located in the sys directory.

In addition to assembly-language code that has to be provided for every port, some additional assembly-language support routines may be needed in the run-time system. The file src/sys/special.s, which initially is empty, is provided as a place to put such routines. Routines in src/sys are automatically included by files in the run-time system.

Although most of the work of porting is conducted in src/sys, it may be necessary to make some modifications to source files for the components of Icon that are generally machine-independent. For example, it may be necessary to include port-specific code for checking division by zero in src/ops/div.c, the routine that implements Icon division. Any modifications to a source code file that is not in src/sys should be done under control of conditional preprocessor commands. One of the byproducts of

make Setup-port HOST=*name*

is to define the uppercase version of *name* to be 1 in a header file that is included by all source-code files. Thus, if *name* is ibm370, code specific to this implementation might be included as

```
#if IBM370
      :  code specific to the IBM 370
      .
#endif IBM370
```

If this convention is used carefully, such modifications will not corrupt the source code in non-sys directories and the resulting code from the port can be merged into the central version of the source that is maintained at the University of Arizona. This offers several advantages: the code for the port can be maintained at a central location, copies of the port can be made available to other sites, and modifications of the central source code will not obsolete the port.

There are a number of Makefiles in src directories that are used to build various parts of the Icon system. These Makefiles are parameterized so that they can be adapted to the needs of specific systems. The shell script sys/Setup edits these files in a non-destructive way, so that parameters in Makefiles can be changed in the local environment. If the parameters supplied in sys/Setup are not satisfactory for the port, they can be changed as needed. Look at Setup scripts in existing implementations, such as vax, for examples. Do not edit existing Makefiles by hand, if at all possible, since this will make it more difficult to incorporate the port as part of the central source maintained at the University of Arizona.

Because files in the src/sys directory are used for development, inadvertently running Icon-setup and thus destroying the files in src/sys would be disastrous. To protect against this, the file src/sys/.protected is created and Icon-setup does not perform the set-up tasks if this file exists.

## 4. Macro Definitions

The first step in the porting process is to supply a number of C preprocessor definitions. The file params.h contains these per-system definitions. The porter edits this file to define constant definitions and macro expansions as described below. Some definitions can not be determined until preliminary work on the port has taken place. Information on supplying the definitions for these macros is deferred until the appropriate time. In some cases, there are recommended definitions or definitions that will prove to be correct on most systems. In such cases, the value is pre-supplied in params.h.

### 4.1 Data Structure Sizing

A number of data structures used throughout the translator, linker, and run-time system are sized by means of constants whose values vary between implementations. Experience has indicated that for the purposes of the Icon system, machines can be characterized as having a small memory or a large memory. Machines such as the PDP-11, which has a 64K data space, clearly need conservatively sized structures, while machines such as the VAX, which has a virtual address space, can easily accommodate larger data structures.

The porter may characterize the target system as having a large or small memory by a single #define and this in turn selects a set of appropriate constants. The large memory model is selected by

```
#define LargeMem
```

in params.h, while

```
#define SmallMem
```

selects the small memory model. If for some reason neither the definitions selected by LargeMem or SmallMem are suitable, the porter may select appropriate values for the constants defined in h/memsize.h. Note that if this route is taken, the porter must define values for *all* of the constants in h/memsize.h. In this case, the definitions for the constants below should be at the end of params.h, replacing the inclusion of ../h/memsize.h.

The following constants are defined in h/memsize.h. Unless otherwise noted, the values are used by the run-time system.

TSIZE
: The size of the translator's parse tree space.

SSIZE
: The size of the translator's string space.

MaxCode
: The maximum number of bytes code that can be generated by the linker for a single procedure.

MaxAbrSize
: The initial size in bytes of the allocated block region.

MaxStrSpace
: The initial size in bytes of the string space.

StackSize
: The size, in words, of co-expression stacks.

MaxStacks
: The number of co-expressions stacks initially allocated.

NumBuf
: The number of i/o buffers available. When a file is opened, a buffer is assigned to the file if one is available.

**SSlots and TSlots**
The number of hash table slots for sets and tables respectively. These values should be prime numbers that are not close to a power of 2.

**MaxListSize**
The largest list element block that can be made. This value is only applicable on machines with address spaces of 64 kbytes and the value in h/memsize.h should be used.

### 4.2 Machine Characteristics

These definitions describe aspects of the target system that are related to the properties of the CPU.

**IntSize**
The number of bits in an *int*.

**LogIntSize**
The base 2 log of IntSize. That is, LogIntSize answers the question "What power of 2 is IntSize?".

**LONGS**
Icon has an integer data type whose range of values is from $-2^{31}$ to $2^{31}-1$. On the VAX, C *int*s and *long*s are both 32 bits wide. On the PDP-11, C *int*s are 16 bits wide while *long*s are 32 bits wide. The PDP-11 Icon system makes an internal distinction between integers that "fit" in 16 bits and integers that require 32 bits. The former are stored in two-word descriptors (the actual value being in the second of the two 16-bit words), while the latter have a value descriptor that points to a block in the heap that holds the two-word, 32-bit value. On the other hand, the VAX uses two 32-bit words for descriptors and thus the second word of a descriptor can hold the largest possible integer value used by Icon. Rather than having an internal distinction between integer types on the VAX, integers are always represented by two-word integer descriptors. There are places in the code where special provisions are made if C *int*s are not the same size as C *long*s.

If sizeof(int) != sizeof(long) for the C compiler in use, define LONGS. (LONGS need not be given a value, #define LONGS is sufficient.) If LONGS must be defined, the minimum and maximum values that can be represented by an *int* must also be defined. Define MinShort to be the smallest value that an *int* can hold and define MaxShort to be the largest value that an *int* can hold.

**MaxLong and MinLong**
The largest and smallest values representable by a *long*.

**LogHuge**
The highest base-10 exponent plus 1 representable by a *float*. For example, on the VAX, the highest number representable by a *float* is about $1.7 \times 10^{38}$. Thus, LogHuge is 39 on the VAX.

**WordSize**
The size in bytes of a word. This should be defined as sizeof(int).

**Descriptor Flags**
The symbols F_Nqual, F_Var, F_Tvar, and F_Ptr should be defined as a set of bit masks with one bit set in each. F_Nqual should have the leftmost bit set; F_Var should have the second bit from the left set, F_Tvar the third, and F_Ptr the fourth. For example, on the VAX, these are:

```
#define F_Nqual     0x80000000
#define F_Var       0x40000000
#define F_Tvar      0x20000000
#define F_Ptr       0x10000000
```

On a 16-bit machine they should be:

```
#define F_Nqual     0x8000
#define F_Var       0x4000
#define F_Tvar      0x2000
#define F_Ptr       0x1000
```

### 4.3 Procedure and Operator Declarations

The macro definitions in the section of **params.h** denoted by the comment **Procedure and Operator Declarations** are entwined with procedure frame layouts and appropriate values for these definitions are discussed in Section 7.2 (*Procedure Frame Layout*).

### 4.4 Source Code Tailoring Definitions

**cset_display**

This is a rather complicated macro that is used to initialize the values of csets such as **&cset** and **&lcase**. If the target machine has *int*s with 32 or 16 bits, then one of the definitions of **cset_display** in **params.h** may be used. If this is not the case, **cset_display** will have to be hand-crafted and the various uses of it will have to be altered for the machine in question. Briefly, **cset_display** specifies which of the 256 bits that comprise a cset are to be set to 1. For example, the **cset_display** for **&cset** has all the bits set to 1, while **&ascii** has the first 128 bits set to 1. Csets are accessed using the **setb** and **tstb** macros, which are also defined in **params.h**. Uses of **cset_display** appear in **iconx/init.c**, **fncs/bal.c**, and **fncs/trim.c**. In certain cases, for example on a machine with 36-bit words, it may be necessary to modify the definitions of **CsetSize**, **setb**, and **tstb**.

**SetBound** and **ClearBound**

See Section 7.4.2 (Boundary Setting and Clearing) to determine the correct values for these definitions.

**Return**

This macro provides a "hook" at the return point of built-in Icon functions. In most circumstances this macro should be defined as **return**, but if for some reason the porter needs an action performed when built-in functions are ready to return, an alternate definition for **Return** can be used.

**DclSave**

This definition is used in an elaborate way in some earlier implementations. This document describes implementation techniques that render this macro obsolete and it should be defined as /* */.

**VarArgs**

This is another "hook" macro. A call of this macro appears as the first executable statement in **fncs/stop.c**, **fncs/write.c**, and **fncs/write.c** This can be used to perform an operation when one of these functions is called. Under usual circumstances however, it should be defined as /* */.

**UpStack**

At certain points, supposedly machine-independent C code must deal with values on the stack. Defining **UpStack** causes such code to assume that the stack grows up rather than down. Porters on such systems should examine the various points in the source code where **UpStack** is used to be sure that the supplied code will work on the target system.

**Arg(n)**, **ArgType(n)**, and **ArgVal(n)**

See Section 7.2 (*Procedure Frame Layout*) to determine appropriate definitions.

### 4.5 Miscellaneous Definitions

**PFMarkerHigh, GFMarkerHigh, EFMarkerHigh**

This is the offset in words from procedure, generator, and expression frame pointers (respectively) to the high word of the associated frame. These values are dependent on frame design and should be specified when the frame layout has been completed.

**GranSize**

The granularity of memory allocations. Calls to *sbrk*(2) are used to expand the main memory that is being used. When *sbrk* is given an address to expand to, it rounds it to a multiple of a certain number. That value should be used for **GranSize**. The man page for *sbrk*(2) should state what value is used on the target system.

**StkBase**

This value represents the approximate base of the stack when execution begins. One machines such as the VAX, where the stack grows down from high memory, **StkBase** should have a high value, where on the machines where the stack grows up from low memory, **StkBase** should have a low value. The man page for exec(2) usually specifies the initial value for the stack pointer where program execution begins. If uncertain, be extreme with this value.

MaxHdr

This value specifies the maximum expected size of bin/iconx.hdr. Do *make iconx.hdr* in src/icont. Round the size of iconx.hdr (in bytes) up to the next multiple of 1024 and use this value for MaxHdr. This value is not used on systems that support direct execution of interpretable *icode* files as described in [3].

OpSize

See Section 6 (*Porting the Linker*) to determine this value.

OpndSize

This value specifies the size in bytes of icode operands. This should be defined to be WordSize.

## 5. Porting the Icon Translator

### 5.1 Overview

The Icon translator, known as itran, is the first logical component of the Icon system. The translator takes Icon source files as input and produces two *ucode* output files for each input file. The Icon program in the file hello.icn may be translated by:

```
itran hello.icn
```

This produces two ascii files, hello.u1 and hello.u2. hello.u1 contains instructions and data in a printable format. hello.u2 contains information about global symbols and scope.

The translator is written entirely in C and is the most machine independent major system component of Icon. No serious problems should be encountered in porting it. If difficulties are encountered, they probably indicate that there are serious deficiencies in the C compiler being used.

### 5.2 Porting Procedure

The only system-specific material in the translator is related to the sizing of data structures and specification of SmallMem or LargeMem in params.h causes these structures to be sized appropriately. Thus, the translator may be compiled by changing to the v5 directory and issuing the command:

```
make tran
```

### 5.3 Testing The Translator

Once the translator has been successfully constructed with *make*, change to v5 (if not already there) and test it by

```
make Test-tran
```

This runs the translator on a number of Icon programs, produces ucode output in .u1 and .u2 files, and uses *diff* to compare the results to output that is known to be correct. Since the translator is machine-independent and written entirely in C, there should be no differences.

## 6. Porting the Icon Linker

### 6.1 Overview

The Icon linker, known as ilink, is the second logical component of the Icon system. The linker takes .u1 and .u2 files produced by the translator and binds them together to form an icode file. The icode file serves as input for the Icon run-time system.

For example,

```
ilink hello.u1
```

reads hello.u1 and hello.u2 and produces a file hello, which can be executed by the run-time system.

The linker is written entirely in C and is a comparatively small and simple program. However, the interpretable files produced by the linker are not machine independent. Because of this, the porter must make some decisions.

Icode files contain two distinct types of data: opcodes and associated operands that the interpreter "understands", and data that is directly mapped into run-time data structures. By "mapping", it is meant that the data is loaded into memory and then C structure references are used to access elements of the object at a certain location in memory. The formats of the opcodes and operands must conform to what the interpreter is expecting. The data that is directly mapped must conform to the format of the C data structures used by the run-time system.

The opcodes, operands, and mapped data are accumulated in memory during the linking process. This conglomerate is referred to as the code section. Several routines are used to add data to the code section. These routines are parameterized so that porting the linker to a new machine is merely a matter of setting the parameters correctly. Four primitive data units compose the code section. These are *opcodes*, *operands*, *words*, and *blocks*.

opcodes
: are instructions for the interpreter. An opcode may direct the interpreter to push a value on the stack, branch to a location, perform an arithmetic operation, etc. The size of opcodes is specified by the porter.

operands
: are associated with some opcodes. For example, the `goto` instruction has a location to branch to as its single operand. Operands are defined to be `WordSize` bytes in length.

words
: compose mapped data structures. For example, the data blocks for Icon procedures are a series of words. Obviously, words are `WordSize` bytes in length.

blocks
: are merely some number of bytes. For example, a `cset` constant is loaded into the code section as a block of `sizeof(struct b_cset)` bytes.

### 6.2 Porting Procedure

The per-system parameterization required for the linker is almost completely specified by the definitions made earlier in `params.h`, but the porter must define the opcode size, which is specified by `OpSize` in `params.h`.

The interpreter treats opcodes as unsigned quantities. One byte (8 bits) is large enough to accommodate all opcodes and a value of 1 is strongly recommended for `OpSize`. It is possible to use larger opcodes; two or four bytes may prove to be a convenient choice on a machine that requires memory accesses to be on two- or four-byte boundaries. It should be noted, however, that there is no way to put the extra bytes to use. The `outop` routine in `lcode.c` assumes that opcodes are one byte; if a larger size is used, `outop` must be recoded.

The constant `OpndSize`, which defines the size of interpreter operands is defined to be `WordSize` in `params.h` and this value should not be changed under normal circumstances.

Compile the linker by changing to `v5` and

```
make link
```

### 6.3 Testing the Linker

When the linker is successfully compiled, change to `v5` and build the Icon command processor:

```
make icont
```

Then test the linker by

```
make Test-link
```

which runs the Icon linker on the files produced by the translator during the preceding test and produces linker debugging output in `.ux` files. This process is comparatively slow because of the generation of debugging output. The format of .ux files is somewhat dependent on computer and operating system details. Consequently, there are likely to be differences — even extensive ones — between the locally generated .ux files and the distributed ones. Differences are not checked by make Test-link, but they can be determined separately by

```
make Test-linkcheck
```

If extensive differences are encountered, it may be necessary to examine the output in v5/port/local manually.

## 7. Porting the Icon Run-Time System

The run-time system, known as *iconx*, is the third major logical component of the system. The run-time system takes an icode file produced by the linker and "executes" it. A program is run by:

```
iconx hello
```

where hello has been produced by the linker.

The run-time system has four logical components:

start-up code
an interpreter
primary routines
support routines

The start-up code initializes the run-time system and passes control to the interpreter. The interpreter fetches icode instructions and executes them. An icode instruction may be entirely performed by the interpreter or the interpreter may call a *primary routine* to perform the operation. In turn, a primary subroutine may call a number of *support routines* that in turn may call other support routines. Each primary routine has a direct correspondence to a source language construct of some type. Primary routines are also referred to as *top-level routines.*

### 7.1 Overview of the Porting Process

The following steps are to be followed when porting the run-time system:

(1) Determination of layout of procedure, generator, and expression markers and selection of associated frame pointers.

(2) Definition of remaining macros in params.h and definition of macros in defs.s.

(3) Complete system compilation.

(4) Coding of a "basis" of routines for the run-time system, consisting of start.s, invoke.s, interp.s, efail.s, pfail.s.

(5) Testing of the basis routines for the run-time system.

(6) Coding and testing of

```
arith.s
fail.s
pret.s
esusp.s
lsusp.s
psusp.s
suspend.s
display.c
```

in an incremental fashion. Test programs are provided to test the system after adding each routine.

(7) Coding of gcollect.s and sweep.c. Testing of garbage collection.

(8) Complete system testing.

This document does not explain how to port the sections of the system that are related to co-expressions. The files involved are coact.s, cofail.s, coret.s, create.c, and refresh.c. Icon works properly with these sections of code left unimplemented, provided no attempt is made to use co-expressions, in which case the system notes it as a run-time error.

### 7.2 Porting Procedure

**Determining Frame Layouts**

This implementation of Version 5 of Icon shares the stack between the C and Icon run-time environments. The essential ramification of this is that the system contains code that is intimately entwined with both the machine itself and with the C run-time environment. This requires that the porter be familiar with the architecture of the target machine and with various aspects of code generated by the C compiler being used.

The first step is to determine the frame layout and call/return protocol used by C functions. Occasionally, the reference material for a system will contain this information, but usually this information must be gathered empirically.

In one form or another, the following actions (perhaps in a slightly different order) comprise the call/return protocol used by most C compilers:

**Argument Set-Up**

This typically involves pushing the various argument values on a stack of some sort. This is usually done in an accumulative fashion, i.e., the argument expressions are evaluated in turn (typically from right to left) and while the evaluations may use the stack, the end result of each is a value on the top of the stack that is not disturbed until the routine is called.

**Routine Entry**

This is the machine-level transfer of control from the series of instructions comprising the call to the routine to the routine itself. On many machines, this is nothing more complicated than pushing the current program counter value on the stack and then branching to the first instruction of the routine being called.

**Register Save**

Most machines have one or more general-purpose registers that are available for use by the programmer (and hence, code generated by the C compiler). By convention, certain registers are preserved across subroutine calls, i.e., these registers are guaranteed to have the same value when a subroutine returns that they had when a subroutine was called. The subroutine and subsequently-called routines may use the registers, but each routine ensures that when it is finished, the register values are what they were when the routine was called.

C compilers save either a fixed or variable number of registers. Saving a fixed number of registers is usually more straightforward, but on a machine with many registers, it is obviously inefficient to save many more registers than necessary. Methods of accomplishing the register save are diverse, but the registers usually end up on the stack.

Note that if a variable number of registers are saved, this must be done under the control of the routine being entered since in general, the caller cannot know which registers are used by the callee. If a fixed set of registers are saved however, it is possible for the caller to save the registers.

**Local Space Allocation**

Local C variables that are dynamically allocated usually lie on the stack above the saved registers. Allocating space for the locals is simply a matter of making space on the stack by subtracting from, or adding to the stack pointer as appropriate.

**Routine Execution**

This is simply the execution of the routine being called.

**Register Restoration**

This step restores the registers that were saved when the routine was entered. If the number of registers saved is variable, there is obviously coordination of some type that ensures that the registers saved are those that are restored.

**Routine Exit**

This is the machine-level transfer of control back to the point of call. This is often as simple as popping a previously-pushed program counter value and jumping to it.

**Post-Call Cleanup**

This takes place in the routine that initiated the call and typically consists of nothing more than the adjusting the stack pointer to pop the arguments the routine was called with. Note that this can only be done by the caller because, while not recommended, a C routine can be called with more or fewer arguments than it is expecting.

### An Example — The Sun Workstation's MC68000

The porter must familiarize himself with the above steps as instantiated on the target machine. The MC68000 C compiler used by the Sun Workstation provides a good example of empirically determining a call/return protocol for a particular machine. Consider the following program:

```
main()
{
        f(1,2);
}
f(a,b)
int a, b;
{
        register int i = 1;
        register char *p = 0;
        int x, y;

        x = a;
        y = b;
}
```

Compiling this with cc -S on the Sun yields a .s file that is similar* to:

*The actual compiler output is somewhat stylized, for pedagogical purposes, the text shown here is a sanitized version Also note that the Sun assembler uses an unusual syntax for operands involving register displacements and these operands have been rewritten in a more commonly used format

```
_main:
                ...
[1]             pea         2
[2]             pea         1
[3]             jbsr        _f
[4]             addqw       #8,sp
                ...
_f:
[5]             link        a6,#-16
[6]             moveml      #0x2080,(sp)
[7]             moveq       #1,d7
[8]             movl        #0,a5
[9]             movl        8(a6),-4(a6)
[10]            movl        12(a6),-8(a6)
[11]            moveml      -16(a6),#0x2080
[12]            unlk        a6
[13]            rts
```

The first point of interest is the evaluation of the expression f(1,2). In this C compiler, argument evaluation is from right to left. At lines 1 and 2 respectively, the constants 2 and 1 are pushed on the stack using peas (push effective address).

Line 3 enters f with the jbsr instruction. This instruction pushes the pc value on the stack and transfers control to the address named by the operand, _f in this case.

Execution proceeds with line 5. link pushes a6 on the stack and points a6 at the word just pushed. This is used to form a chain of frames. At any point during execution, a6 points to a word that contains the previous value of a6. The second operand of link specifies a value to add to sp to make space for later use. In this case, the value of -16 causes four words of space to be reserved.

The next action taken is to save the registers. The moveml instruction at line 6 does this. moveml accepts two operands: a register mask and a starting address. The Sun has 16 general-purpose registers, a0–a7 and d0–d7. The rightmost bit in the register mask corresponds to d0; the leftmost bit corresponds to a7. The mask 0x2080 selects d7 and a5 (from right to left). The mask is scanned from right to left and the selected registers are stored in successively higher words beginning at the starting address. Thus, d7 is stored at -16(a6) and a5 is at -12(a6).

In this case, note that the registers are above the space allocated for the local variables and that the link instruction created space for both.

At this point, the stack is

| | | |
|---|---|---|
| sp → | -16 | Saved d7 |
| | -12 | Saved a5 |
| | -8 | Local y |
| | -4 | Local x |
| a6 → | 0 | Saved a6 value |
| | 4 | Saved pc value |
| | 8 | Formal parameter a (1) |
| | 12 | Formal parameter b (2) |

The declarations and accompanying assignments for i and p in the C source are present to force registers to be used and thus, saved. Lines 7 and 8 perform the requested assignments.

Line 9 performs the assignment x = a. movl transfers the left operand into the right operand and thus x is at -4(a6) and a is at 8(a6).

Similarly, line 10 performs y = b, and shows that y is at -8(a6) and that b is at 12(a6).

It is now time for f to return to its caller. A moveml is used to restore the appropriate registers. Note that the operands are reversed from what was used to save the registers. In this case, the mask is again scanned from right to left, but the selected registers are loaded from successive words beginning at the named address.

The unlk (unlink) instructions loads a6 from the word that a6 points at and then points the stack pointer at the prior word, leaving the stack in the state it was in just before the link at the start of the routine was performed.

At line 13, the return pc value is on the top of the stack and rts (return from subroutine) pops this word from the stack and branches to the addresses named by it.

Execution is now at line 4. The arguments for f (1 and 2) are still on the stack and the addqw instruction adds 8 to the stack pointer, effectively removing them and leaving the stack in the same state it was in before the evaluation of f(1,2) began.

In summary, the generated code shows the following about the call/return protocol:

(1) The frame pointer is a6. The previous a6 value is at 0(a6).

(2) Local variables start at -4(a6) and extend toward lower addresses.

(3) Arguments start at 8(a6) and extend toward higher addresses.

(4) The return pc is at 4(a6).

(5) A variable number of registers are saved and it not possible to determine which registers are present in a frame without examining the code that created the frame.

**Another Example — The VAX-11**

Compiling the same program segment on a 4.2bsd VAX yields the assembly code:

```
_main:
            ...
[1]         pushl       $2
[2]         pushl       $1
[3]         calls       $2,_f
            ...
_f:
[4]         .word       0xc00
[5]         subl2       $8,sp
[6]         movl        $1,r11
[7]         clrl        r10
[8]         movl        4(ap),-4(fp)
[9]         movl        8(ap),-8(fp)
[10]        ret
```

Evaluation of f(1,2) begins with line 1. As on the Sun, argument evaluation is right to left and pushls (push longword) are used to push the constants 2 and 1 on the stack.

Entry to f is accomplished with the calls instruction. The first argument of calls is the number of words in the argument list; the second is the location of the routine being called. The VAX calls instruction and its counterpart, ret, entirely implement the call/return protocol.

The first action of calls is to push the word count of the argument list (the first operand of calls) onto the stack. This word is often referred to as the nwords value.

The next step is to examine the halfword (two bytes) at the start of the routine being called; this word is a mask that indicates which registers should be saved. The VAX has twelve general-purpose registers, r0–r11, and the lower twelve bits of the mask correspond to these with the rightmost bit representing r0. Thus, the mask 0xc00 indicates that r10 and r11 should be saved. The mask is scanned from left to right, in this case pushing first r11 and then r10 onto the stack.

After the registers have been saved (note that in some cases, no registers are saved), the values of pc, fp (frame pointer), and ap (argument pointer) are pushed on the stack in turn. Then a word containing the current program status word and the register mask of the routine being entered is pushed on the stack. Finally, a condition handler address, which is not used by the C compiler and is always zero, is pushed on the stack.

fp is pointed at the word on the top of the stack. ap is pointed at the nwords word. On the Sun, both the arguments and the locals lie at a fixed distance from the frame pointer. On the VAX, however, because the registers are saved between the arguments and the rest of the frame, ap is used to reference the arguments and fp is used for the other frame pointer duties.

At line 5, the stack pointer is decremented by 8 to make space for local variables. At this point, the stack is:

| | | |
|---|---|---|
| sp → | -8 | Local variable y |
| | -4 | Local variable x |
| fp → | 0 | Condition handler |
| | 4 | Program status word and register mask |
| | 8 | Saved ap |
| | 12 | Saved fp |
| | 16 | Saved pc |
| | 20 | Saved r10 |
| | 24 | Saved r11 |
| ap → | 0 | Nwords value (2) |
| | 4 | Argument a (1) |
| | 8 | Argument b (2) |

Lines 6 and 7 perform the assignments to i and p.

Line 8 performs the assignment x = a. movl transfers the left operand into the right operand and thus x is at -4(fp) and a is at 4(ap).

Similarly, line 9 performs y = b, and thus y is at -8(fp) and b is at 8(ap).

f can now return to its caller, and this is entirely handled by the ret instruction. Values for ap, fp, and pc are restored from the frame. The register mask saved in the frame is used to pop the saved registers from the stack. This leaves the nwords value on top of the stack and this value is popped and then the number of words in the argument list, as indicated by nwords, are popped. This leaves the stack in the state it was in before the evaluation of f(1,2) began and execution continues after the calls at line 3.

In summary, the generated code demonstrates the following:

(1) The frame pointer is fp and due to the placement of the saved registers, a second register, ap, is used to point at the argument list.

(2) Local variables start at -4(fp) and extend toward lower addresses.

(3) Arguments start at 4(ap) and extend toward higher addresses.

(4) Previous values of ap, fp, and pc are stored at 8(fp), 12(fp), and 16(fp) respectively.

(5) A variable number of registers are saved, but by examining the register mask present in the frame, it is possible to determine which registers were saved and where they are located.

The porter should generate assembly-language output on the target system for the above C code and gain an understanding of the code produced as has been described for the Sun and the VAX. Once the porter has gained such an understanding, the run-time system frames for the target system can be determined.

**Determining Register Preservation Conventions**

The set of registers that the C compiler assumes are preserved across calls must be known by the porter. Obviously, on a machine that saves a fixed set of registers, examining the assembly code for any routine should provide this information. On machines that save a variable set of registers, this information can usually be determined by creating a routine that forces all available registers to be used. Assuming that the C compiler in use heeds the register attribute of declarations, the assembly code generated for a routine that contains a number of such declarations usually indicates which registers are preserved across calls. For example, start with the code:

```
f()
{
    register int i1 = 0;
}
```

Compile it and note which register was saved. Then add i2 =0, i3 = 0, and so forth until registers stop being saved. In most cases, the registers being saved at this point comprise the set of registers that the C compiler assumes is preserved across calls. On machines such as the Sun that have more than one type of register, a series of declarations that use each of the register types is required. For example,

```
register char *s1 = 0, *s2 = 0, ...;
```

causes the Sun's address register to be used.

On the VAX, r6–r11 are preserved and on the Sun a2–a5 and d2–d7 are preserved.

**Procedure Frame Layout**

With one exception, the AT&T 3B, on all machines that Icon has been ported to, the C run-time stack grows downward, from higher memory addresses to lower memory addresses. Furthermore, the argument evaluation on these systems is such that for the function call f(a,b,c,d), the argument d is pushed first, and the argument a is pushed last. Thus, in the routine f, the arguments from left to right lie in sequentially increasing locations.

The execution of Icon programs is stack-based and computations use one or more operands on the top of the stack and leave a result on the top of the stack. C routines that implement Icon primitives are declared as:

routine-name(isb, nargs, arg*n*, ..., arg*0*)

isb is the *istate block*; it is discussed in more detail later. nargs is simply an int. The various arg*i* are struct descrips.

For the calculation 1 + 2 + 4 in Icon, the generated icode does the following:

push a null value
push a null value
push the constant 1
push the constant 2
call the routine plus to perform addition
push the constant 4
call the routine plus to perform addition

plus is declared as

```
plus(isb, nargs, arg2, arg1, arg0)
```

When plus is first called, arg0 has the null value, arg1 has the value 1, and arg2 has the value 2. Note that one of the actions taken when plus is called is to push values for nargs and isb. plus performs the addition and places the result, 3, in arg0, replacing its null value (the second null value pushed). plus was called from the interpreter loop and when it returns, isb, nargs, arg2, and arg1 are popped from the stack, leaving arg0, with the value 3 on top of the stack.

Next, the constant 4 is pushed on the stack, and plus is called again. This time in plus, arg0 has the null value, arg1 has the value 3 and arg2 has the value 4. plus performs the addition and places the result, 7, in arg0, replacing the first null value pushed. When plus returns, all but arg0 is popped from the stack, leaving it on the top.

This simple paradigm is used for all computations and interacts perfectly with the code generated by the C compiler.

Now consider the case of the AT&T 3B, where the call f(a,b,c,d) causes a to be pushed first and d to be pushed last. Conversely, the arguments are popped from the stack one at a time, a is the last argument to come off. Considering the example again, it is obvious that in this case, the routine should be declared as

```
plus(arg0, arg1, arg2, nargs, isb)
```

Assuming this, then things proceed as before: The null value of arg0 is replaced by the sum of the value of arg1 (1) and arg2 (2). plus returns and arg1 through the isb, which lie at higher memory locations than arg0 are popped, leaving arg0 on the top of the stack.

To cope with the problem of needing two different argument list forms, macros are used to generate the C routine declarations. There are several different top-level macros to deal with the various classes of C routines that implement Icon primitives:

ProcDcl(name, nargs)
: Declare built-in function name with nargs arguments. Also declare a procedure data block for name.

ProcDclV(name, nargs, var)
: Same as ProcDcl, but with var as a dummy parameter.

OpDcl(name, nargs, print-name)
: Declare operator name with nargs arguments. print-name is the special character representation of the operator. For example, the print-name of plus is +. Also declare a procedure data block for name.

OpDclV(name, nargs, print-name, var)
: Same as OpDcl, but with var as a dummy parameter.

LibDcl(name, nargs)
: Declare library routine name with nargs arguments.

On machines with down-growing stacks, the definitions of these macros and sub-macros that are used on the VAX should work and appear in params.h. On other machines, the porter must supply appropriate alternative definitions.

A problem related to argument ordering is that of argument access in built-in routines such as write that accept a variable number of arguments but have no declarations for the arguments themselves. (Since the number of arguments to such functions is arbitrary, it is not practical to supply a sufficient number of formal arguments.) For example, on the VAX, write is declared as

```
write(isb, nargs)
```

The macro

```
Arg(n)
```

is used to access the *value* (not the address) of the *n*th argument. Because it is known that the arguments lie in ascending locations directly after nargs, Arg(n) is defined as:

```
*((struct descrip *)(&nargs+1)+(nargs-n))
```

Two other argument-access macros, ArgType(n) and ArgVal(n), are used to access the first and second words, respectively, of the indicated argument. As with the declaration macros, the VAX values are supplied in params.h; appropriate values will need to be supplied for machines with stacks that grow up.

Note that on machines with up-growing stacks, the routines fncs/stop.c, fncs/write.c, fncs/writes.c, and lib/llist.c will probably need to be declared (via appropriate macro definitions) as

```
routine(arg0)
```

with the argument access macros using arg0 instead of nargs as the point of reference.

Continuing with the procedure frame layout, the istate block is a three-word structure that is used to hold the istate register values present in the callers environment. (Selection of the istate registers is explained below.) The VAX uses

```
struct isb_b {
        int isb_ipc, isb_gfp, isb_efp;
        };
```

No machine-independent code uses this structure, so the porter can order the words as desired or add words, but the above structure should prove adequate except under unusual circumstances.

The first task is to determine the layout of Icon procedure frames. The basic structure of a procedure frame is:

Icon local variables
_file
_line
C routine frame
istate register block
nargs
$arg_n$
. . .
$arg_1$
$arg_0$

The exact frame format chosen depends on the target system, and while various permutations of the above are possible, it is highly recommended that this format be used. Note that the same basic layout holds for both machines with down- and up-growing stacks; in the former, the local variables are at lower memory addresses, while in the latter, they are at higher memory addresses. On the VAX, the frame format is:

| | | | |
|---|---|---|---|
| | | Icon local variables | |
| | -8 | saved value of _file | |
| | -4 | saved value of _line | |
| fp → | 0 | 0 (condition handler address) | |
| | 4 | program status word and register mask | |
| | 8 | saved ap | |
| | 12 | saved fp | |
| | 16 | saved pc | |
| ap → | 0 | number of words in argument list (nwords) | |
| | 4 | saved ipc (r9) | |
| | 8 | saved gfp (r10) | istate block |
| | 12 | saved efp (r11) | |
| | 16 | number of arguments (nargs) | |
| | | arguments | |

The first argument is at 20(ap) and the first local is at -16(fp).

Procedure frames on the Sun look like:

| | | | |
|---|---|---|---|
| | | Icon local variables | |
| | -8 | saved value of _file | |
| | -4 | saved value of _line | |
| a6 → | 0 | saved a6 | |
| | 4 | saved pc | |
| | 8 | saved ipc (a3) | |
| | 12 | saved gfp (a4) | istate block |
| | 16 | saved efp (a5) | |
| | 20 | number of arguments (nargs) | |
| | | arguments | |

The first argument is at 24(a6) and the first local is at -16(a6).

As can be seen, determining the frame layout for the target machine is largely a matter of building around C routine frames.

**Selection of Istate Registers**

The porter must select which general-purpose registers are to be used as the Icon interpreter program counter (ipc), generator frame pointer (gfp), and expression frame pointer (efp). Any three registers that are preserved across subroutine calls should do. By convention, the registers are consecutive and the lowest numbered register is used as the ipc, but this is not required. These registers are collectively referred to as the *istate registers.*

If the target machine does not have enough registers, one or more of the istate "registers" can be located in memory. This of course requires special actions on the part of the porter that will become apparent upon reading the descriptions of the routines.

The VAX uses r9 for the ipc, r10 for the gfp, and r11 for the efp. The Sun uses a3 for the ipc, a4 for the gfp, and a5 for the efp.

### C Routine Frames

The frame format used by C routines that implement built-in procedures and operators is a subset of the Icon procedure frame. These frames do not include the saved values for _line and _file, and obviously do not include the region of Icon local variables. Not surprisingly, these frames are created by performing a subset of the operations used when activating an Icon procedure.

### Generator Frame Layout

While procedure frames provide part of the execution environment for Icon procedures, generator frames provide a means to reactivate execution. There are two types of generators: Icon and C. Icon generators reactivate execution in an Icon context while C generators reactivate execution in a C context.

Generator frames contain saved values for _file, _line, _k_level, and _boundary. Also included are machine-specific values such as the frame pointer and program counter. Values of general-purpose registers are also contained in the generator frames. In Icon contexts, the only register values that need to be restored are those of the istate registers, while in C contexts, the registers that the C compiler preserves across calls must be restored.

Generator frames should be designed to contain the registers necessary to restore a C context. This results in wasted frame space for Icon generators, but the simplicity realized by this approach outweighs the unutilized space.

On the VAX, the convenient variable-sized frames allow generator frames to contain only the necessary registers, i.e., frames for Icon generators contain only the istate registers (r9–r11) while C generator frames contain r6–r11. The exact frame format is:

| | | |
|---|---|---|
| | -12 | saved value of _file |
| | -8 | saved value of _line |
| | -4 | saved value of _k_level |
| gfp → | 0 | saved value of _boundary |
| | 4 | condition handler address |
| | 8 | program status word and register mask |
| | 12 | saved ap |
| | 16 | saved fp |
| | 20 | saved pc |
| | -24 | saved r6 |
| | -20 | saved r7 |
| | -16 | saved r8 |
| | -12 | saved ipc |
| | -8 | saved gfp |
| | -4 | saved efp |
| ap → | 0 | |

The routines that create Icon generator frames, esusp, lsusp, and psusp, have entry masks that direct the istate registers to be saved while the routine that creates C generator frames, suspend, has an entry mask that directs r6–r11 to be saved.

On the Sun, this format is used:

| | | | |
|---|---|---|---|
| | -52 | saved value of _file |
| | -48 | saved value of _line |
| | -44 | saved value of _k_level |
| | -40 | saved d2 |
| | -36 | saved d3 |
| | -32 | saved d4 |
| | -28 | saved d5 |
| | -24 | saved d6 |
| | -20 | saved d7 |
| | -16 | saved a2 |
| | -12 | saved a3 |
| | -8 | saved a4 |
| | -4 | saved a5 |
| gfp → | 0 | saved value of _boundary |
| | 4 | saved a6 |
| | 8 | saved pc |

**Expression Frame Markers**

Expression frame markers are essentially machine-independent. The format is:

| | | |
|---|---|---|
| | -8 | failure address |
| | -4 | saved gfp |
| efp → | 0 | saved efp |

On some machines (the Ridge-32 is a case in point), the stack must be always aligned on an 8-byte boundary. In such cases, an extra word should be added to the marker at -12(efp).

At this time, values for PFMarkerHigh, GFMarkerHigh, and EFMarkerHigh in params.h should be defined. On the VAX, these values are 2, 3, and 2, respectively. On the Sun, they are 2, 13, and 2, respectively. Note that these values are word counts. For example, the high word of a generator frame marker on the Sun is -52(gfp) and this corresponds to the Sun's GFMarkerHigh value of 13 (words).

**Assembly-Language Macro Definitions**

The file defs.s contains a set of definitions that are used by assembly language source files. While it is not strictly necessary that the porter use any of these definitions, they do help with readability and the sections of this document that deal with the VAX-specific code often refer to these definitions. Of course, the porter may add other definitions that are of use.

WordSize
: The size in bytes of an *int*.

DescSize
: The size in bytes of a descrip structure.

Isb_size
: The size in bytes of an isb_b structure.

fp
: The name of the register used by C as the frame pointer.

ipc, gfp, and efp
: The names of the registers selected to serve as the interpreter program counter, the generator frame pointer, and the expression frame pointer.

There are a number of values in procedure frames that are accessed at different times. On the VAX, a set of macros are used to name the appropriate location. Most instruction sets have an operand form that names a location in terms of a register value and an offset from that location. For example, on the VAX, the nargs word is the fourth word below the word ap points at and there is the definition:

```
#define Nargs_loc (4*WordSize)(ap)
```

Similarly,

```
#define Line_loc (-1*WordSize)(fp)
```

indicates that the word of the procedure frame that contains the saved value of _line is the word below the word pointed at by fp. These values are defined as follows:

| | |
|---|---|
| File_loc | saved _file value |
| Line_loc | saved _line value |
| Ap_loc | saved value of ap |
| Fp_loc | saved value of fp |
| Pc_loc | saved value of pc (the return pc) |
| Ipc_loc | saved value of ipc |
| Gfp_loc | saved value of gfp |
| Efp_loc | saved value of efp |
| Nargs_loc | location of nargs word |
| Argn_loc | location of first word of descriptor for $arg_n$ |

In addition to the *name*_loc values, which have a register associated with them, there are corresponding definitions for *name*_off values. For example, Nargs_off is (4*WordSize). These values are just offsets and are used when another register holds the fp value (or on the VAX, ap value) of interest. As a simple example, to put the value of the nargs word in r0, one might use

```
movl    Nargs_loc,r0
```

or equivalently,

```
movl    ap, r2
movl    Nargs_off(r2),r0
```

Ep_Off
: The offset in bytes from the address of a routine to the first instruction after the routine's entry sequence. See the section on invoke.s for details on the use of this value.

Arg_desc, Arg_dword, and Arg_vword
: See the section on CallCtl in interp.s to determine these definitions.

The Icon keywords &trace, &pos, and &subject are represented as trapped variables, but their actual values are used in the assembly language routines. Since these values lie at a fixed location in b_tvkywd structures, rather than explicitly naming the location, values are defined for _k_trace, _k_pos, and _k_subject. _k_trace and _k_pos are expected to have integer values and thus the second word of the descriptor b_tvkywd.kyval is the desired word. Thus, defining _k_trace and _k_pos as (respectively)

```
_tvky_trc+(3*WordSize)
_tvky_pos+(3*WordSize)
```

should work on all machines. _k_subject should name b_tvkywd.kyval itself and thus is defined as

```
_tvky_subject+(2*WordSize)
```

Several definitions in defs.s expand into one or more instructions:

PushNull
: This should expand into one or more instructions that push a null value on the stack. On the VAX, this could be written as:

```
movq    _nulldesc,-(sp)
```

but because it is done very often, a more efficient two-instruction sequence is used:

```
pushl    $0
pushl    $D_Null
```

**Push_isb**

This should push the istate registers on the stack, forming the istate block. On the VAX, this could be written as:

```
pushl    efp
pushl    gfp
pushl    ipc
```

but the pushr instruction is used instead. pushr takes a register mask as its operand and pushes words onto the stack containing the values of the registers indicated by the mask. For example,

```
pushr  $0x0005
```

pushes r2 and then pushes r0. Push_isb is defined as

```
pushr    $StdSv
```

where StdSv is defined as 0xe00 to select r9–r11.

**Pop_isb**

This is a companion of Push_isb that is used to pop registers that were pushed by Pop_isb. The popr instruction is the counterpart of pushr and pops registers according to a mask. On the VAX, Pop_isb is

```
popr     $StdSv
```

**CallPush(nargs, address)**

This is a subordinate macro that is the front-end of Call and CallName and is used to create a procedure frame and enter a routine. Obviously, the actions of this macro must be coordinated with the desired procedure frame layout. On the VAX, this pushes *nargs* (always a constant) on the stack, follows that with the istate block, and then calls the routine at *address*. The exact code is:

```
pushl    $nargs
Push_isb
calls    $0,address
```

A zero-length argument list is used for calls to allow the istate block to be left on the stack. If this were not done, there would be no way to restore the istate registers.

**CallPush_R(reg, address)**

This is identical to CallPush with the exception that the register reg contains the number of arguments rather than it being specified by a constant value. Thus,

```
pushl    reg
```

is used rather than

```
pushl    $nargs
```

**CallPop**

This routine is the back-end of the various call macros. It restores the istate registers and then pops the appropriate number of arguments from the stack, leaving the result value produced by the routine on top of the stack. On the VAX, it is:

```
Pop_isb
movl     (sp)+,r0
movaq    (sp)[r0],sp
```

Pop_isb restores the istate registers and removes them from the stack. This leaves the nargs word on top and it is popped and moved into r0. Finally, movaq performs the calculation:

```
sp = sp + (nargs * DescSize)
```

to pop the arguments from the stack.

CallName(nargs, name)
On the VAX, this expands to:

```
CallPush(nargs, name)
CallPop
```

CallName_R(reg, name)
On the VAX, this expands to:

```
CallPush_R(reg, name)
CallPop
```

DummyFcn(name)
Initially, each of the assembly language routines that must be filled in consist of a single line of the form DummyFcn(name). DummyFcn should be defined to generate *assembly* language statements that form a dummy routine with the label name. This can be as simple as a label and a global declaration. It is advisable to include as part of the definition something that will cause a program abort. A halt instruction usually does the job. Thus, the system can be built and will function normally unless an incomplete routine is called.

DummyDcl(x)
A macro that should expand into an assembly language declaration that allocates a word of storage for a variable named x.

DummyRef(x)
A macro that should expand into an assembly language reference to the symbol x. That is, the desired effect is to have x referenced in a particular routine so that the loader considers it to be a symbol that needs to be resolved.

Global(x)
A macro that should expand into an assembly language declaration of x as a global symbol.

A number of values that are defined in params.h must also be defined in defs.s. These values are F_*flagtype*, T_*typename*, TypeMask, and MaxStrLen.

defs.s also contains macro calls to Global for the various global symbols that expected to be used in the assembly language routines. If the porter needs additional global declarations, they can be added in defs.s or in the file containing the reference.

### 7.3 Complete System Compilation

In order to determine if there are serious C compiler problems with the run-time system source, the entire system should be made at this point. Do a

```
make iconx
```

in v5. The entire system should compile without any problems. The resulting interpreter will be disfunctional, but if it is built without any problems, it provides further evidence that the C compiler is up to the task.

### 7.4 Porting the Assembly-Language Routines

The porting of the assembly language routines is the most difficult part of porting Icon. This document has a section for each assembly language routine and each routine is described in three ways:

overview
generic operation
the routine on the VAX

The overview section briefly describes the action of the routine and how the routine may be encountered during the course of execution. The generic operation section tells what steps the routine takes to perform its given task. Each major step that the routine takes is described. These steps should be very similar from machine to machine. The section about the routine on the VAX details the operation of the routine on the VAX. This section complements the comments contained in the source code for the routine and should be read with the source code at hand. This section is very machine specific.

Each routine must be formulated for the target machine. For the most part, the best approach is to take the same steps that are taken on the VAX. It is important to select the right level for modeling the VAX routines. Try to recognize the steps that are made rather than following the operations on a per-instruction basis. The most important thing is to have a good understanding of what actions are performed and how these can be done on the target machine.

### A Simple Program

The first goal is to get a very simple Icon program working. This first program is v5/port/hello.icn. It is quite short:

```
procedure main()
   write("Hello world")
end
```

The basis of routines mentioned above (start.s, invoke.s, interp.s, efail.s, and pfail.s) must be implemented for even a very simple Icon program to work. However, all these routines do not need to be written to make hello *begin* to work.

Translate and link hello by running the translator and the linker:

```
bin/itran hello.icn
bin/ilink hello.u1
```

This creates an interpretable file named hello. Just to get the feel of things, execute the run-time system with the file:

```
bin/iconx hello
```

A message of some type and a core dump should be produced.

As start.s et al. are written, try stepping through them with a debugger to be sure the correct actions are being performed. Most of the assembly language source files are straight-line code with a branch or two, and it is possible to do a large amount of verification of the assembly code by single stepping through it.

When a routine has been completed, it may be added to the run-time system by:

```
make iconx
```

in v5.

## 7.4.1 start.s

### Overview

When the Icon interpreter is executed, the C routine main passes control to mstart, merely serving as a front-end for it. The routine mstart in start.s is used to get Icon started.

### Generic Operation

(1) Call the routine init with the name of the file to interpret as its argument.

(2) Make an Icon list out of the command line arguments using the llist function.

(3) Invoke the main procedure of the Icon program.

### mstart on the VAX

There is a short main program in iconx/main.c that calls mstart with two arguments:

```
main(argc, argv)
int argc;
char **argv;
{
mstart(argc, argv);
}
```

The number of command line arguments is in argc, and argv is a pointer to an array of pointers to strings representing the arguments. argv[0] is the command used to invoke the interpreter and argv[1] is the name of the file being interpreted. Additional command line arguments are passed along to the main procedure of the Icon program. When mstart gets control, 4(ap) is the argc value and 8(ap) is the argv value.

The first action taken by mstart is to call init to initialize the Icon run-time system. init loads the header and code portions of the interpretable file into memory, so init needs the name of the interpretable file. The word at 8(ap) is loaded into r9, pointing it at argv[0]. Then the name of the file to interpret (argv[1]), residing at 4(r9), is pushed on the stack as the argument for init, which is then called.

A troublesome point is the deactivation of the main procedure. This occurs when the Icon procedure main fails, suspends, or returns. One of these always happens unless a run-time error is encountered.

The case of failure is handled by creating an expression frame for the main procedure. An expression frame marker is pushed on the stack. This marker has efp and gfp values of 0 and a failure address of mterm. mterm is the address of a quit opcode (just a 0) for the interpreter. Thus, if main fails, the marker is removed and icode execution continues with the execution of the quit opcode at mterm and this terminates execution of the program. The handling of return and suspend by main is described below.

The next task is to push the descriptor for the procedure main on the stack for later use by invoke. The variable _globals contains the address of the list of Icon global identifier descriptors. The first global identifier descriptor is always the one for the main procedure; if no procedure named main was found when the program was linked, the descriptor has the null value. The value of _globals is loaded into r0 and the word then referenced by r0 is checked to see if it is equal to D_Proc. (The first word of a descriptor for a procedure is always equal to D_Proc.) If the word is not equal to D_Proc, a branch is made to nomain, which generates the appropriate run-time error. Otherwise, the descriptor for main is pushed onto the stack with

```
movq (r0),-(sp)
```

which moves 8 bytes (the size of a descriptor on the VAX) starting at the address referenced by r0 to the 8 bytes referenced by the sp after subtracting 8 from the sp.

The main procedure is to be invoked with a list consisting of the command line arguments (if any). The Icon run-time routine llist is used to make the list that is passed to the main procedure. llist stores the descriptor for the list that it creates in the descriptor above its first argument descriptor, so to accommodate the result, a null descriptor is pushed on the stack using PushNull.

At the beginning of mstart, r9 was set to point at the first word of the argument list. Neither the name of the Icon interpreter nor the name of the interpretable file should appear in the argument list passed to main, so 8 is added to added to r9 to point it at argv[2], the first actual argument.

The next step is to construct the argument list for llist. For each command line argument, the address of the string and then its length (determined by a call to strlen()) is pushed on the stack. The length and address pairs form string descriptors that llist makes an Icon list from. A count of the arguments is maintained in r8. After a descriptor has been pushed for each command line argument, llist is called with

```
CallNameR(r8,_llist)
```

When llist returns and the istate block and arguments have been popped (recall that CallNameR does all this), the stack looks like this:

sp → descriptor for list of command line arguments
descriptor for main procedure

Note that the null descriptor pushed earlier received the result of the llist function.

Because llist allocates storage, it sets and clears the boundary. Thus, it is not possible to execute all of mstart at this stage of the port until a means of managing the boundary has been determined. This is described in the next section.

At this point, the main procedure is ready to be invoked. The descriptor for the main procedure is $arg_0$ and the descriptor for the list of command line arguments is $arg_1$. Before invoking the main procedure, the procedure frame pointer and the generator frame pointer are cleared.

A bit of hackery is used to invoke the main procedure. For all other procedure invocations, the code at the op_invoke case of interp is used and it is neither suitable to enclose this code in a subroutine or duplicate the necessary code in mstart. Instead, control branches to the _invk_start label in the op_invoke case of interp. The code at _invk_start assumes that the number of arguments supplied to the procedure is in interp's Op register and that the arguments are on the stack. On the VAX, r1 is used for the Op register. The porter should consult the section on interp to determine a suitable register to use for Op.

Before the branch to _invk_start is made, ipc is loaded with the address of mterm. Thus, if main returns or suspends, execution will continue with the icode opcode at mterm, which is a quit.

There is a block of code labeled nomain that is executed when no main procedure is found. This calls the routine runerr to produce an error message. The actual call is runerr(117, 0).

The last portion of executable code in start.s is the subroutine _c_exit. The routine __cleanup is then called to shut down the i/o system. Finally, _exit is called with the argument of _c_exit to terminate execution of the Icon interpreter.

There are several data declarations in start.s. The first data declaration is a .space 60. This is an accommodation for the garbage collector. It insures that enough of the start of the data section is used up to force the addresses of other data objects to be greater than the defined constant MaxType in params.h.

Some assorted declarations are next. mterm is referenced by the interpreter if the main procedure terminates normally. It should be OpSize bytes in length and have a 0 value. _boundary must be a word long and contain a 0. _environ must be a word long; its contents are unimportant as it is written into at the beginning of start.s.

The _tended array is also used in conjunction with garbage collection. It must declare space for five descriptors (two words per descriptor) that are initialized to 0. The label _etended is used to mark the end of the _tended array.

### 7.4.2 Boundary Setting and Clearing

**Overview**

As described in [1], when a C routine is active, i.e., when execution is in a C context, _boundary holds the value of the frame pointer for the top-level C routine originally entered from the Icon context. Conversely, when execution is in an Icon context, _boundary should be 0.

The source code for various primary routines contains calls to the macro SetBound at points where the boundary should be set and ClearBound at points where the boundary should be cleared. It is the task of the porter to define appropriate expansions for SetBound and ClearBound.

Clearing the boundary is easily accomplished by boundary = 0 and this is predefined as the value of ClearBound in params.h.

It is usually necessary to resort to assembly language to set the boundary. On the VAX, this can be accomplished by

```
#define SetBound {asm(" movl fp,_boundary");}
```

The asm statement causes its operand to be placed directly in the assembly code; the braces are necessary to avoid a bug in some C compilers that causes incorrect placement of the assembly language text.

If the C compiler in use does not support the asm statement, a subroutine can be used instead. For example,

```
#define SetBound setbound()
```

and setbound itself would just move the fp value saved in the frame for setbound into _boundary. Using an asm is preferable because it is much faster, but a subroutine call works as well.

### 7.4.3 interp.s

**Overview**

interp.s is the main loop for the interpreter. As the interpreter executes an Icon program, it fetches instructions and accompanying operands out of the instruction stream of the interpretable file. Operands for interpreter instructions are pushed on the stack and results accumulate on the stack as operands for other instructions. In addition to simple incremental and decremental stack changes, the expression evaluation mechanism may cause portions of the stack to be duplicated and may also cause the top portion of the stack to be removed.

**Generic Operation**

An Icon program is executed by interpreting the interpretable file produced by the linker. The interpretation process itself is fairly simple. ipc points at the next icode instruction to be executed. (Recall that the interpretable file is loaded into memory.) The opcode of the instruction is fetched and the corresponding word in a jump table is taken as the address of a sequence of instructions that perform the desired operations. A branch is taken to the referenced location and the operation is performed. The operation may require operands; if so, they appear in the instruction stream following the opcode. The segment of code that performs a particular operation is responsible for fetching the appropriate operands out of the stream. When the operation is complete, a jump is taken to the top of the interpreter loop and the process continues.

Interpreter operations are of two types. Operations of the first type call a routine to perform a task. Operations of the second type are executed entirely by the interpreter; no subroutine call is necessary.

Operations that require a call to be made call routines in the ops or lib directories. The routine being called may require one or more arguments. If arguments are required, they appear on the stack. When the routine returns, it removes any arguments that it was called with from the stack and leaves its result on the top of the stack.

To facilitate the calling of routines, the table optab parallels the jump table. An opcode of *n* references the *n*th word of the jump table. If the operation designated by the opcode requires a call, the *n*th word of optab contains the address of the routine that should be called.

The interpreter saves space in its instruction stream by encoding operand information in some opcodes. For example, the line instruction has one operand that is used to set the value of _line, the current source line number. The linex instruction is an alternate form of line which encodes the line number as the low order bits of the opcode. For example, the opcodes from 192 to 256 are linex opcodes and opcode 195 is equivalent to a line opcode with an operand of 3. Other such instructions are: global, local, int, static, arg, and invoke.

**Implementing the Interpreter Loop**

interp.s stands alone among the assembly language files as one that is well suited to coding in a macro fashion. Most of the interpreter loop is written in terms of C preprocessor macros and thus porting it is largely a matter of writing the macros for the target machine. The porter should copy vax/interp.s to sys/interp.s and work on it, changing VAX-specific sections to code appropriate for the target machine.

The following #defines must be made.

Op

The operand register. Any general purpose register will do. The value of the register need not be preserved between instructions; its lifetime is only from the time that an operand is fetched until the next opcode is fetched or a routine is called.

**GetOp**

This must expand into code that fetches the next operand out of the instruction stream and places it in the register **Op**. Recall that operand size is determined by the **#define** for **OpndSize** in **params.h**. On the VAX, **GetOp** is merely

```
movl    (ipc)+,Op
```

This is because operands are one word long and can begin on any byte boundary. If the VAX did not support word fetching from arbitrary boundaries, it would be necessary to get the bytes from the instruction stream one at a time and make a word out of them using boolean operations. If such were the case, a possible alternative would be to make opcodes one word in size and thus all instruction stream objects (opcodes, operands, and words), would be of the same size and lie on word boundaries.

A second alternative is to have a more elaborate **GetOp**. A subroutine could be used to fetch the next operand, but the interpreter loop is a busy piece of code and incurring the overhead of a subroutine call for each operand is not a good idea. The solution of course is to have **GetOp** expand into the required instructions.

For example, on the Sun, the **ipc** is checked and if it is even, a simple word fetch into the **Op** register is performed, but if it is odd, the operand is fetched in pieces and assembled in the **Op** register using bit-manipulation operations.

**PushOp**

Push the **Op** register on the stack. The VAX uses

```
pushl   Op
```

**Push_R(x), Push_S(x), Push_K(x)**

Push the value of **x** on the stack. To accommodate machines with non-orthogonal instruction sets, **Push_R** is used to push a value contained in a register, and **Push_S** is used to push the contents of a storage location. **Push_K** is used to push a constant value. The VAX uses

```
pushl   x
```

for both **Push_R(x)** and **Push_S(x)**, while

```
pushl   $x
```

is used for **Push_K(x)**.

**PushOpSum_R(x)** and **PushOpSum_S(x)**

**PushOpSum_R(x)** adds the value of the register **x** to **Op** and pushes the result on the stack. **PushOpSum_S(x)** is similar, adding the value in the memory location **x** to **Op** and pushing the result. On the VAX,

```
addl3   Op,x,-(sp)
```

is used for both.

**NextInst**

Branch to the top of the interpreter loop. The VAX uses

```
jmp     _interp
```

**BitClear(m)**

The constant value **m** designates bits in the **Op** register to leave on. All other bits in **Op** should be turned off. That is, the complement of **m** is ANDed with the contents of **Op** and the result is placed in **Op**. This is used to decode opcodes with encoded operands. The VAX uses

```
bicl2   $0!m,Op
```

Call(n)

This calls the routine corresponding to the current opcode with n arguments. On the VAX, the opcode fetching segment loads r6 with a byte offset into the jump table. This same byte offset references the word in optab that contains the address of the routine corresponding to the current opcode. On the VAX, this expands to:

```
CallPush(n,*optab(r6))
CallPop
```

CallCtl

This macro is used to call the routines esusp, lsusp, and psusp. These routines do not require an istate block and the nargs word and CallCtl merely calls the appropriate routine. As with Call, the routine to call is implicitly named by an optab offset in r6. On the VAX, this is

```
calls    $2,*optab(r6)
```

The descriptor for the value being suspended is on the top of the stack.

Arg_desc, Arg_dword, and Arg_vword

These macros reference the argument descriptor in routines that are called by CallCtl. Arg_desc and Arg_dword reference the first word of the descriptor and on the VAX are (1*WordSize)(ap). Arg_vword references the second word of the descriptor with (2*WordSize)(ap) on the VAX.

DerefArg(n, lab)

This examines the *n*th descriptor from the top of the stack and calls _deref if the descriptor is a variable of some sort. Such descriptors always have the F_Nqual and F_Var bits set. On the VAX (and other 32-bit machines), a string would have to be over two billion bytes in length in order to have its length overflow into the F_Var and thus, the presence of the F_Var is considered to be characteristic of a variable descriptor. On the VAX, this code is used:

```
          bitl      $F_Var,(n*8)(sp)
          beql      lab
          pushal    (n*8)(sp)
          calls     $1,_deref
lab:
```

The bitl instruction tests a set of bits. The two operand values are ANDed together and the condition codes are set according to the value of the result. The result itself is discarded.

On a 16-bit machine such as the PDP-11, it is necessary to check for both F_Nqual and F_Var to identify a variable descriptor.

Jump(lab)

Branch to the label lab. The destination label is close to the jump, so a short jump of some type may be used. The VAX uses

```
jbr      lab
```

LongJump(lab)

LongJump is like Jump with the exception that lab may be quite distant. The VAX uses

```
jmp      lab
```

Label(lab)

Generate a label declaration for lab. The VAX uses

```
lab:
```

## VAX Specific Sections of interp

Several sections of interp are machine specific and must be coded on a per-machine basis. The sections in question are explained on an individual basis:

_interp
: The next opcode is fetched and loaded into r6 with a movzbl which moves a byte and zero-extends it to a word value. Because a byte was fetched, ipc is incremented by 1. The opcode is copied to Op in case it contains an encoded operand. r6 is multiplied by 4 to turn it into a byte offset. A jump is made to the address indexed by r6 in jumptab to perform the desired operation. Eventually, a jump returns control to the label _interp to fetch and execute the next instruction.

op_bscan
: A descriptor for _k_subject is pushed on the stack. Then the value of _k_pos is pushed, followed by the constant D_Integer. The routine corresponding to op_bscan, _bscan, is called with 0 arguments. (This causes the descriptors for _k_subject and the value of _k_pos to be left on the stack.) When _bscan returns, a branch is made to _interp.

op_ccase
: A null descriptor is pushed on the stack. The word immediately below the current expression frame is then pushed on the stack.

op_chfail
: The operand of chfail is fetched into Op. Op and ipc are added together and the result replaces the failure address in the current expression frame.

op_dup
: A null descriptor is pushed on the stack. The value that was on top of the stack is now at 8(sp), and it is pushed on the stack using a movq.

op_eret
: eret saves the value on top of the stack, removes the current expression frame and puts the previous top of stack value back on the top of the stack. First of all,

```
movq    (sp)+,r0
```

moves the descriptor on the top of the stack into the r0-r1 register pair and increments the stack pointer by 8. The gfp is loaded with the gfp value stored in the expression frame marker. sp is loaded from efp, bringing the expression marker to the top of the stack. The old efp value from the marker is loaded into efp. Finally, the value stored in the r0-r1 pair is pushed on the stack.

op_file
: The operand of file is loaded into Op. Op and the value of _ident are added and the result in placed in _file.

op_goto
: The operand is loaded into Op and then added to ipc.

op_init
: This one is tricky. The init instruction arises from the initial expression in Icon and is used to effect one-time execution of a segment of code. The operand of init is the address of the first instruction after the segment that is to be executed once. The instruction

```
movb    $59,-(ipc)
```

decrements ipc by 1 and then stores the constant 59 in the byte that ipc references, which is the init opcode. The magic number 59 is the opcode for goto, so in effect, the init has been made into a goto that skips a section of code. By adding 5 to ipc, it leaves ipc pointing at the first instruction of the initial code. The constant 5 is derived from the width of the opcode and associated operand, i.e., OpSize+OpndSize.

op_int
: This instruction has two entry points: op_int gets control if int has an operand, and op_intx gets control if the operand is encoded in the opcode. If an operand is specified, it is fetched into Op. If the operand is encoded, BitClear(15) is used to isolate the operand in Op. Control converges at intjmp. The Op value

is pushed on the stack and is followed by a D_Integer word, forming an integer descriptor.

op_line
: Like op_int, op_line has a secondary entry point. The operand value is obtained and then moved into _line.

op_mark
: The operand is fetched into Op and ipc is added to it. efp is pushed on the stack and the new sp value is put in efp. gfp is pushed on the stack and cleared. Op is pushed on the stack.

op_mark0
: Like op_mark, with an implicit operand value of zero.

op_pop
: Two tstl instructions serve to add 8 to sp which removes the top value from the stack.

op_sdup
: The descriptor on the top of the stack is pushed on the stack, duplicating it.

op_unmark
: The operand, the number of expression frames to remove from the stack, is fetched into Op. efp is restored from the current expression frame. The instruction

```
sobgtr   Op,unmkjmp
```

decrements Op and then branches to dounmark if Op is not zero. This chains through the number of expression frames specified by the operand. gfp is restored from the current expression marker. efp is loaded into sp to move the expression marker to the top of the stack. Finally, efp is restored from the marker and sp is incremented to remove the last word of the marker.

op_unmk1-7
: Similar to unmark, but uses a series of

```
movl     (efp),efp
```

instructions rather than a loop.

op_global
: Dual entry points are used to deal with possible operand encoding. The operand, which is a number of a variable in the global region, is multiplied by 8 to provide a byte offset from the start of the global region. The sum of Op and the value of globals is pushed on the stack to provide a descriptor address. The constant D_Var is pushed on the stack to complete the descriptor for the global variable.

op_static
: Identical to op_global except that the array statics is used instead of the array globals.

op_local
: The operand value is the number of a local variable for which a variable descriptor is to be pushed on the stack. Recall that the local variables lie below the procedure frame and, on the VAX, the descriptor for the first one is at -16(fp). Op is negated. The instruction

```
pushaq  -16(fp)[Op]
```

performs the calculation

```
-16+fp+(Op*DescSize)
```

which computes the address of the descriptor of the desired variable and pushes it on the stack. The variable descriptor is completed by pushing D_Var on the stack.

op_arg
: Like op_local, but it uses Argn_loc as the base for the address calculation and the operand value is not negated.

quit

Performs the call _c_exit(0). Push a 0 on the stack and call the routine _c_exit to terminate execution of the Icon program.

err

err should never be encountered during normal execution. Reaching it indicates that an invalid opcode was encountered. It need not do anything more than abort execution. On the VAX, it calls sprintf to create a string containing the invalid opcode and the ipc where it was encountered and then calls syserr (in iconx/init.c) with the string as an argument.

op_invoke

One of the more complex tasks required of the interpreter is called *invocation* and it arises from a source expression of the form:

$arg_0(arg_1, \ldots, arg_n)$

There are four distinct outcomes from the execution of this expression:

call a built-in function or operator
call an Icon procedure
create a record
perform mutual evaluation

The evaluation of an invocation expression includes the evaluation of each $arg_i$ (in a strict left-to-right order) and accumulation of the resulting values on the stack. After every $arg_i$ has been evaluated, the code implementing invocation takes control, examines $arg_0$ and performs the appropriate actions.

**Generic Operation**

The code that implements invocation is clearly complex, and principles of software modularity suggest that this code be implemented in the form of a subroutine. However, an often-performed task during invocation is the adjustment of argument lists for built-in functions and Icon procedures. Doing this while in the frame context of the interpreter loop (as opposed to in a subroutine) facilitates a much simpler and efficient implementation. The net result is that the majority of the invocation code is found at the op_invoke case of interp, and a short subroutine named invoke assists in the final step.

When the op_invoke case is reached, the stack is:

sp → value from $arg_n$
...
value from $arg_i$
...
value from $arg_1$
value from $arg_0$

(1) $arg_0$ is dereferenced and checked to see if it is a procedure. If so, execution continues with (4).

(2) Since $arg_0$ is not procedure-valued, an attempt is made to convert $arg_0$ to an integer. If this succeeds, a mutual-evaluation is to be done and the appropriate $arg_i$ replaces $arg_0$. The other arguments are popped, leaving the selected value on top of the stack. For example, given

```
2(1,5,9)
```

$arg_0$ is 2 and $arg_2$, the value 5, is the result of the expression.

Execution continues at the top of the interpreter loop.

(3) $arg_0$ is neither a procedure or an integer. An attempt is made to convert $arg_0$ to a string. If this succeeds, strprc is called with the address of $arg_0$ and the number of arguments to see if the string names a procedure-valued object. If this is the case, $arg_0$ is converted to a descriptor for the named object and execution continues with (4).

If the conversion to string and/or the subsequent conversion to procedure is unsuccessful, it is noted as run-time error 106.

(4) At this point it is known that $arg_0$ is procedure-valued. Each $arg_i$ in turn is dereferenced.

(5) If the procedure being invoked has a fixed number of arguments, the argument list is adjusted as necessary. If too few arguments were supplied, null values are pushed on the stack. (Note that the shortage is always at the right end of the argument list, which corresponds to the top of the stack.) Conversely, if too many arguments were supplied, excess arguments are popped.

(6) The routine invoke is called to create the frame for the procedure being invoked and execution proceeds therein.

(7) If a built-in procedure or operator is being invoked, that is, if the invocation will cause execution to continue in C code, the boundary is set to the current frame pointer value and the appropriate routine is entered by a branch. Otherwise, an Icon procedure is being invoked and further actions are required.

(8) If _k_trace has a non-zero value, the function ctrace is called with appropriate arguments. ctrace produces output that includes the name of the procedure being called and the arguments that are being passed to it.

(9) The remainder of the procedure frame (partially constructed by the call to invoke) is built. This includes pushing values for _file and _line on the stack. _file is a pointer to a string that names the source file from which the code currently being executed came. _line is the number of the source line that is currently being executed. A null-valued descriptor is pushed on the stack for each dynamic local identifier of the procedure.

(10) The generator frame pointer is cleared (because a new expression context is being entered). ipc is loaded with the entry point of the procedure being called. Control is then passed back to the interpreter using a jump.

When the invoked routine returns, control returns to the point in the interpreter loop where invoke was called. The boundary is cleared and a branch is taken to the beginning of the interpreter loop.

Record creation occurs when an object whose value is a record constructor is invoked. The data block associated with this object is essentially a procedure block, but the routine associated with it is mkrec, which handles the actual record creation in a machine-independent way.

### invoke on the VAX

As with several other icode instructions, the operand of invoke is encoded in the opcode if possible. The label invk_start begins the actual code for invoke.

When control reaches invk_start, the interpreter's Op register contains invoke's operand, the number of arguments for the invocation. This value is transferred to r6.

invoke makes frequent use of $arg_0$, but its address is not a fixed distance from any known point. Rather, the address of $arg_0$ must be calculated using the address of the last argument and the number of arguments. The VAX movaq instruction makes this calculation easy. The desired calculation is

$$\&arg_0 = \text{sp+(number of arguments * DescSize)}$$

and is performed by

```
movaq   (sp)[r6], r7
```

$arg_0$ may be a variable and if so, it needs to be dereferenced. r7, which contains the address of $arg_0$ is pushed on the stack and deref is called. The dereferencing is done "in place"; the previous value of $arg_0$ is replaced with the dereferenced value. The dereferenced value is a descriptor whose first word contains type information and whose second word (in some cases) contains the address of a data block which holds the actual value of the object. Note that r7 points to the first word of this descriptor.

Recall that the first task of invoke is to determine what $arg_0$ is and to act accordingly. The simplest case is when $arg_0$ is a procedure. That is checked for by comparing 0(r7) with D_Proc. If $arg_0$ is a procedure, a forward jump is made to doderef.

It is more interesting if $arg_0$ is not a procedure. The first alternative investigated is mutual evaluation. Mutual evaluation is similar to a procedure call, but rather than $arg_0$ being a procedure, it is an integer that selects one of the $arg_i$. The selected $arg_i$ is the outcome of the mutual evaluation. The routine cvint is used to try to convert $arg_0$ to an integer. If $arg_0$ cannot be converted to an integer, a forward branch is taken to trystr to explore another possibility. For mutual evaluation, a non-positive value of $arg_0$ is acceptable and is converted to a positive value using the cvpos routine. (Expressions in the argument list are indexed the same way that characters in a string are indexed.) If the returned by cvpos is zero or is greater than the number of expressions in the list, that is, if the reference is out of range, the mutual evaluation fails by branching to efail. If the position is in range, the selected $arg_i$ must be produced as the result of the invocation (and the result of the mutual evaluation). The $arg_i$ to return is selected by multiplying the position by the size of a descriptor, producing the displacement of the desired $arg_i$ from $arg_0$, which is then added to r7 with the result being placed in r0. Thus, r0 points at $arg_i$, r7 points at $arg_0$ and

```
movq   (r0),(r7)
```

moves the desired value into place. The result must be on the top of the stack; moving r7 into the stack pointer accomplishes this. With the result on the top of the stack, a branch is taken to interp to execute the next icode instruction.

If $arg_0$ is not convertible to an integer, conversion to a procedure is attempted. (Note that this is an extension to standard Icon.) $arg_0$ is first converted to a string using cvstr. If the conversion is successful, the routine strprc is called to see if the string "names" a procedure. The conversion performed by strprc is "in place", i.e., $arg_0$ becomes a descriptor for a procedure. If either the conversion in cvstr or strprc fails, $arg_0$ is uninvocable and this is noted by run-time Error 106.

At this point (the label doderef), $arg_0$ is a descriptor for a procedure to be invoked and r7 points to $arg_0$. The next step is to dereference the arguments. For procedures with no parameters, e.g., f(), the associated icode does push a null value on the stack, as if the the expression had been f(&null). Thus there is always at least one argument.

A copy of r7 is made in r8, which is used to point to each argument in turn. In a loop, r8 is decremented by DescSize, and if the F_Var bit is set in the referenced descriptor, the value of r8 is pushed on the stack and deref is called to dereference the descriptor. Since r8 starts at $arg_0$ and moves towards the top of the stack, the loop continues until r8 is less than sp.

The next operation is to make the number of arguments supplied conform to the number of arguments that the procedure is expecting. The number of arguments that a procedure expects is b_proc.nparam, the fourth word of its procedure block. r8 is pointed at the block for the procedure being invoked and the expected argument count is loaded into r1. If the value is negative, the number of arguments that the procedure expects is variable, and no argument adjustment is needed. If this is the case, the supplied argument count in r6, is moved into r1 and a branch is taken to argsdone.

It is now known that the procedure requires a fixed number of arguments and the discrepancy is calculated by subtracting r1, the expected argument count, from r6, the supplied argument count. If the two are equal, no adjustment is required and a branch is taken to argsdone. Otherwise, if r6 is positive, too many arguments were supplied and these can be deleted by pointing the sp to $arg_i$, where $i$ is the expected number of arguments. The instruction

```
movaq     (sp)[r6],sp
```

performs the required calculation. A branch is taken to argsdone.

If the discrepancy value calculated in r6 is negative, too few arguments were supplied and a null value must be provided for each missing argument. r6 is negated with a mnegl instruction and is used as the loop counter for a sobgtr instruction, and the required number of null descriptors are pushed on the stack using successive PushNull operations.

At this point (argsdone) the arguments have been dereferenced and the correct number of arguments are present. The frame for the procedure being invoked is created by entering the invoke routine using

CallName_R(r1,_invoke)

The computational context is now that of the procedure being invoked.

The fifth word of the procedure block (b_proc.ndynam) contains the number of dynamic locals the procedure has. For built-in procedures or operators, this value is negative. If this is the case, control is transferred to builtin, where _boundary is set using the value of fp. All that remains is to enter the C routine itself. The second word of the procedure block (b_proc.entryp.ccode) contains the address of the routine. As described in the section on frame layout, execution of the routine must begin with the first instruction after the prolog that would normally be used to establish the frame for the routine. The displacement of this instruction from the address of the routine is represented by the constant Ep_off, and this value is added to the routine address and a branch is taken to the resulting location. At this point, the C routine is active.

If an Icon procedure is being invoked, more actions are required before the procedure can be activated.

If tracing is on, (indicated by a non-zero value for _k_trace), a trace message must be produced at this point. The routine ctrace does all the work. It needs to be called with the appropriate arguments. ctrace requires three arguments: procedure block address, number of arguments, and the address of the first argument. These are pushed on the stack and ctrace is called.

The portion of the stack from $arg_0$ on down constitutes a partial Icon procedure frame and it must be completed. _line and _file are pushed on the stack. To complete the frame, the local variables must be pushed on the stack. Local variables have an initial null value, and as above, a sobgtr loop of PushNulls are used to push the locals on the stack.

Because an Icon procedure is being invoked, the boundary is cleared and _k_level (the value of &level keyword) is incremented. The entry point for the procedure (b_proc.entryp.icode) is loaded into ipc. Since a new expression context is being entered, both gfp and efp are cleared using a clrq instruction.

Control is passed back to the main loop of the interpreter by jumping to interp. At this point, the Icon procedure is active.

When the invoked routine returns, control is transferred to the instruction following CallName_R(r1, _invoke) in interp.s. _boundary is cleared and NextInst transfers control to _interp.

### 7.4.4 efail.s

**Overview**

efail handles the failure of an expression. When Icon evaluates an expression, it tries to produce a result from it. If at some point in the evaluation of an expression the expression fails, Icon resumes inactive generators in the expression in an attempt to make the expression succeed. efail is at the heart of this activity. efail has three distinct outcomes:

(1) Resumption of the newest inactive generator in the current expression frame.

(2) Failure of the current expression with execution continuing at the failure address contained in the expression marker.

(3) Failure of the current expression with propagation of failure to the enclosing expression frame. This is similar to (2), but occurs when the failure address is 0. After the current expression fails, control loops back to efail, serving to produce failure in the now-current expression frame.

efail is branched to rather than being called. This is because it serves as a "back-end" for several failure actions that may occur during the course of execution:

(1) When a built-in procedure fails, it calls the routine fail, which in turn branches to efail.

(2) When an Icon procedure fails via the pfail routine, pfail terminates by branching to efail.

(3) When the efail opcode is executed by the interpreter, efail is branched to.

(4) The generator frames built by esusp and lsusp use efail as a return address. This is explained in detail later.

**Generic Operation**

efail is essentially a simple routine. There are two separate paths of execution that efail may take. The first is to resume an inactive generator. The second is to cause failure of the expression in lieu of an inactive generator.

If there is an inactive generator in the current expression frame, it must be resumed. If the generator is an Icon procedure and tracing is on, atrace is called with appropriate arguments. _k_level, _line, and _file are restored from the generator frame. A return is performed and the net result is that the stack is restored to the state that it was in before the suspension that created the generator.

If there are no inactive generators that can be resumed, the expression being evaluated must fail. This is done by popping the stack back to the current expression frame and resuming execution at the point indicated by the failure address in the expression marker. This is a two-step process. The first is to pop the frame and the second is to resume execution. The failure address in the expression marker is saved before the frame is popped. If this address is not zero, execution is continued by branching to the address. If the address is zero, the failure is propagated to the enclosing expression by branching to efail.

Zero failure addresses are generated by the ucode instruction

```
mark   L0
```

Thus, whenever efail pops an expression whose marker has a zero failure address, efail causes failure in the enclosing expression.

**efail on the VAX**

The first action is to determine if there is an inactive generator that can be reactivated. If the generator frame pointer is non-zero, it points to the inactive generator to activate. Note that whenever a new expression frame is created, the generator frame pointer is zeroed. Thus, if gfp is non-zero, it points to a generator frame contained in the current expression frame.

If there is an inactive generator, it must be reactivated. First, _boundary is restored from the generator frame. The stack is popped back to the generator frame by loading fp from gfp. But, before fp is loaded, its value is saved in r0 for later use. fp now points at word 0 of the generator frame, but that is a word below the actual stack frame that it should be pointing at, so fp is incremented by 4 using a tstl.

There are three types of generators that may be encountered by efail:

(1) An Icon procedure that did a suspend. In such cases, the routine psusp handled the suspension.

(2) A built-in procedure or operator that called the C function suspend().

(3) A generator created by an esusp or lsusp instruction. Such generators arise from source code constructs like $expr_1$ | $expr_2$, |*expr*, and $expr_1$ \ $expr_2$, which are referred to as *control regimes*.

All three types of suspensions create generator frames with identical formats, so the frames may be handled identically as far as resumption is concerned. However, if an Icon procedure is being resumed, a tracing message must be generated if _k_trace is not 0.

If the value of _boundary is not the same as fp, the generator is a built-in procedure or operator and tracing is not done. If the fp saved in the current frame is the same as the fp was upon entry to efail (the value was saved in r0), the generator was made by an esusp or an lsusp and tracing is not done.

Otherwise, the generator is an Icon procedure, and atrace must be called. atrace takes one argument, the address of the procedure block for the procedure being resumed. Recall that $arg_0$ on the stack is a descriptor for the procedure block. The address of $arg_0$ is calculated using

&$arg_0$ = ap+Argn_off+(nargs*DescSize)

The resulting address is used as the single argument for atrace. Note that the ap and nargs values used in the calculation are from the generator frame, i.e., from the context of the suspended Icon procedure.

The generator is now ready to be resumed. _k_level, _line, and _file are restored by popping them from the generator frame. If the generator is a built-in procedure, _boundary is cleared. A return is performed to activate the generator. The return has different effects depending on the type of generator being resumed.

If the generator is a built-in procedure or operator, the return restores the stack to the state it was in before suspend was called, and execution proceeds at the point just after suspend(). In this case the pc value being returned to references the instruction following the call to suspend in the built-in function or operator.

If the generator is an Icon procedure, the stack is restored to the state it was in before the psusp icode instruction was executed. The pc value being returned to references the instruction in the interpreter loop that follows the call to psusp.

If the generator is a control regime, the stack is restored to the state it was in before the esusp or lsusp that created the generator was performed. The return pc points to efail itself. Thus, when the return is done, the stack is cleared, and an efail is performed. This has the effect of transferring control to the failure label in the expression marker of the enclosing expression frame.

If there is no generator to reactivate, the expression must fail. This is handled at the label nogen. efp points to the expression frame marker. ipc is loaded from -8(efp) which contains the address to go to in the event that the current expression fails (as it has). gfp is restored from the expression marker. efp is restored from the marker and the marker is popped off the stack.

If the failure address in ipc is non-zero, control is passed back to the interpreter via a branch and execution of the icode resumes at the failure address. If ipc is zero, the expression failure is transmitted to the surrounding expression frame by branching to efail. (Recall that a zero failure address comes from a mark L0 instruction and that a failure that reaches a mark L0 marker must be propagated to the next expression marker.)

### 7.4.5 pfail.s

**Overview**

pfail handles the failure of an Icon procedure. pfail is entered via a branch when the interpreter encounters the pfail instruction.

**Generic Operation**

The task of pfail is to signal failure in the expression instance that contains the procedure call being evaluated. This is done by removing the Icon procedure frame from the stack, restoring appropriate registers and values, and branching to efail. All pfail needs to do is to remove the procedure frame from the stack; from then on things can be handled just like expression failure. Thus, efail does most of the work.

pfail calls ftrace to produce a trace message if tracing is on. pfail also decrements _k_level because a procedure is being exited.

Note that the procedure frame on the stack is a frame that was created by invoke.

**pfail on the VAX**

After _k_level is decremented, _k_trace is checked to see if a trace message should be produced. If tracing is on, ftrace must be called. ftrace takes one argument, the address of the procedure block for the failing procedure. $arg_0$ is the descriptor for the procedure block, and the address of $arg_0$ is calculated using

$$\&arg_0 = \text{Argn\_loc+(nargs*DescSize)}$$

The resulting address is pushed on the stack and ftrace is called. Note that the context of the calculation is that of the failing Icon procedure.

Execution continues at dofail to remove the procedure frame from the stack. The frame cannot be merely popped because it contains pertinent state information. Values for _line, _file, ipc, gfp, efp, ap, and fp, are restored from the frame. When fp is restored, it serves to remove the procedure frame (made by invoke) from the stack. At this point, the stack is in the same state it was in before the interpreter performed the invoke instruction. A branch is made to efail to cause failure in the enclosing expression.

## 7.5 Testing

Change to v5 and

```
make Test-basis
```

This runs a number of simple programs and compares the results to correct output.

Although mstart, interp, invoke, efail, and pfail are required for these tests, there are several unexercised paths in these routines, and in particular, many interpreter opcodes are not encountered. Further testing of the run-time system exercises all the execution paths, but the improper operation of a newly-coded may be due to an error in a routine that has already checked-out.

There similar entries in v5/Makefile for testing each module that the porter needs to write. The entries correspond directly to the module name, e.g.,

```
make Test-arith
```

tests arith.s. The other testing entries are:

| | |
|---|---|
| Test-fail | fail.s |
| Test-esusp | esusp.s |
| Test-lsusp | lsusp.s |
| Test-psusp | psusp.s |
| Test-suspend | suspend.s |
| Test-display | display.c |
| Test-gc | gcollect.s and sweep.c |

After completing each module in turn, the porter should test it by *make*ing the appropriate entry in v5/Makefile.

## 7.6 Porting the Rest of the Run-Time System

### 7.6.1 arith.s

#### Overview

arith.s contains code for routines that add, subtract, and multiply long integers and check for overflow. If overflow occurs, run-time error 203 is produced. These operations are performed by subroutines rather than doing them in-line because C does not check for overflow.

The arguments to ckadd, cksub, and ckmul are two C long integers on which to operate. For example, if ckadd were written in C, it would be declared

```
long ckadd(a,b)
long a,b;
{
...
}
```

The routines return the result of the operation using standard C return conventions.

#### arith on the VAX

The two arguments appear on the stack; a is at 4(ap) and b is at 8(ap). The appropriate 3-operand VAX instruction is used to perform the operation and the result is placed in r0 in accordance with C return conventions. If overflow occurs during the operation, the overflow bit in the program status word is set.

After the operation is performed, the overflow bit is checked. If it is on, indicating that an overflow occurred, a branch is taken to oflow, where runerr(203,0) is called. If overflow did not occur, the routine returns and the value in r0 is the value returned to the calling expression.

arith.s is trivial on the VAX because the hardware supports operations on C long integers. This may not be the case on the target machine. If so, arith.s will be considerably more complicated. However, it usually is not difficult to locate routines that perform these functions. It may be helpful to look at the code the C compiler generates for the various arithmetic operations on long integers.

### 7.6.2 fail.s

#### Overview

fail handles the failure of built-in procedures and operators. Built-in procedures and operators are implemented by C routines and they signal failure by calling fail(). When a failure of this type occurs, the failure must be transmitted to the Icon expression whose evaluation is in progress and that requires the services of an assembly-language routine. In some cases, a subsidiary routine used by the function or operator may call fail(); this is handled as if the top-level routine had failed.

#### Generic Operation

fail itself does very little, the real work is done by efail. fail restores the computational context at the time of call to the top-level C routine and then branches to efail to make the enclosing expression fail.

fail is akin to pfail in that it pops the stack back to a state that it was in when an expression was being evaluated and then causes failure of the expression. The differences in the two rises from the slightly different formats of the two types of frames.

#### fail on the VAX

_boundary points to the procedure frame for the top-level C routine that was called from Icon. fp is loaded from _boundary and this puts the stack back to the state that it was in when the top-level C routine was entered. For a built-in procedure, the procedure frame now on the top of the stack (after loading fp from _boundary) is the frame constructed in invoke. For an operator, the frame on the stack is the one constructed when the interpreter loop called the C routine for the operator.

The task at hand is to remove the procedure frame and restore the istate registers. Because the only information directly available about the frame of the failing top-level routine is its fp (just restored from _boundary), the location of the argument list (and thus, the istate block) is unknown. The variability of the location of the arguments is caused by the presence of a variable number of saved registers in the frame for the routine.

The most expedient way to find the istate block is to pop the saved registers off the stack. The mask/psw word of the frame is manipulated so that the mask portion of the word resides in bits 0:11 of r0 and the remaining bits of r0 are 0.

The saved registers start at 20(fp) and sp is loaded with this address. Then popr r0 restores the registers that are saved in the frame. Note that the manipulations of the mask/psw are necessary because it is not known *a priori* which registers were saved. In particular, popr $0x0fff would be disastrous.

When the saved registers have been restored, the nwords word is on the top of the stack and this is popped, leaving the istate block on top. The istate registers are then restored with Pop_isb.

After the registers have been restored, ap and fp are restored from the saved ap and fp values in the frame.

At this point, the stack is as it was before the frame for the built-in procedure or operator was created. All that remains is to signal failure in the expression being evaluated and this is done by branching to efail.

### 7.6.3 pret.s

#### Overview

pret handles the return of a value from an Icon procedure. pret is entered by a branch from the interpreter loop. The descriptor on the top of the stack is the value being returned. The value is dereferenced if necessary. If tracing is on, a trace message is produced. The return value is copied over $arg_0$ in the frame of the procedure that is returning a value. pret does a return through the frame of the returning procedure and this is manifested as a return from invoke, with execution continuing in the interpreter loop. The return leaves $arg_0$ on the top of the stack as the result of the call.

**Generic Operation**

(1) _k_level is decremented because a procedure is being exited.

(2) The stack address where the return value is to be placed is calculated. Recall that when a procedure is invoked, the return value (if any) ultimately replaces $arg_0$, the descriptor for the procedure returning the value.

(3) The value being returned must be dereferenced if it is a local variable or an argument. This is because local variables and arguments are on the stack and the portion of the stack associated with a procedure "goes away" when a procedure returns. If the return value is a variable (its type word has the F_Var bit set) and its address is between the base of the current expression stack* and the stack pointer, it is dereferenced. If it is a substring trapped variable (is of type T_Tvar and points to a block of type D_Tvsubs), and the address of the variable containing the substring is between the base of the current expression stack and the stack pointer, it is dereferenced.

(4) If _k_trace is non-zero, rtrace is called with the address of the block for the returning procedure and the address of the return value descriptor.

(5) fp, _line, and _file are restored from the frame of the returning procedure.

(6) pret returns from the Icon procedure by executing a return instruction. Because the current fp points to the procedure frame for the Icon procedure, and the frame was built by invoke, the return is effectively a return from invoke and the net result is that the return value is left on the stack.

**pret on the VAX**

_k_level is decremented because a procedure is being exited.

The address of $arg_0$ is calculated via

$\&arg_0$ = Argn_loc+(nargs*DescSize)

and stored in r11 for later use.

As described, the value being returned needs to be dereferenced in certain cases. The return value is a descriptor and is on the top of the stack. The first word of this descriptor lies at 0(sp) and contains type and flag information. This word is placed in r1 for further examination.

The instruction

```
bitl    $F_Nqual,r1
```

ANDs the type and flags word with the F_Nqual mask. The F_Nqual bit is set if a descriptor is *not* a string qualifier. If the F_Nqual bit is not on, the result of the AND is a 0. The test is followed by

```
beql    chktrace
```

Thus, if the return value *is* a qualifier, dereferencing is not required and a branch is taken to chktrace.

If the return value does have the F_Nqual attribute, it is checked to see if it is a variable. The F_Var bit is tested. If it is not on, the return value is not a variable and does not have to be dereferenced. A branch is made to chktrace if this is the case.

If a variable is in hand, the F_Tvar bit is checked to see if it is a trapped variable. If it is not a trapped variable, the address field of the return value's descriptor is moved into r1 for further testing and a branch is taken to chkloc.

If the return value is a substring trapped variable, it may reference a local variable or an argument. The type bits of the descriptor are isolated by ORing it with TypeMask. If the type is not T_Tvsubs, no dereferencing is needed and a branch is taken to chktrace. If it is a substring trapped variable, the address of the

*For purposes of uniformity, the system stack is treated as if it were a co-expression stack The global variable _k_current is a pointer to the descriptor for the co-expression stack block for the current co-expression Co-expressions need not be implemented, it is only important that _k_current and the descriptor that it points to be initialized correctly This is done in iconx/init.c variable containing the substring is obtained from the trapped variable's data block and is loaded into r1.

At this point (chkloc), r1 points to a descriptor that is directly or indirectly referenced by the return value. If the descriptor is in the current expression stack, the return value must be dereferenced. r1 is first compared to sp. If it is less than sp, the descriptor is not in the stack and a branch is made to chktrace. Otherwise, r1 is compared to the base address of the current expression stack. If r1 is greater than the base of stack, the descriptor is not in the frame of the current procedure and a branch is made to chktrace.

If control has not branched to chktrace, it is now certain that the return value must be dereferenced, lest it "disappear" when the portion of the stack it is in is re-used. The address of the return value is pushed on the stack and deref is called. Note that deref completely handles dereferencing of substring trapped variables and thus no special provisions need to be made.

At chktrace, the return value has been dereferenced if necessary and it is time to produce a tracing message if _k_trace is non-zero. rtrace does the work and it requires two arguments: the address of the block for the returning procedure, and the address of the return value. Earlier, the address of descriptor for the procedure block ($arg_0$ of the now-returning procedure) was calculated and left in r11. The address of the return value and the address of the block for the returning procedure are pushed on the stack as arguments for rtrace and it is called.

pret "returns" the designated value by overwriting the procedure's descriptor with the descriptor of the return value. r11 points at the descriptor for the procedure and the return value is still on the top of the stack, so

```
movq    (sp),(r11)
```

does the trick.

_line and _file are restored from the Icon procedure frame. A ret is executed. The return goes through the procedure frame built by invoke. Thus, control is returned to the point just after the call to invoke and it appears as if invoke itself had just returned.

### 7.6.4 esusp.s

**Overview**

esusp suspends a value from an expression. esusp is called from the interpreter loop and the value to suspend appears as an argument. A generator frame hiding the current expression is created. The surrounding expression frame is duplicated. esusp leaves the value being suspended on the top of the stack.

The esusp operation arises from the alternation ($expr_1$ | $expr_2$) control structure. For example

```
p(5 | 10)
```

indicates that the call p(5) should be made and if it fails, then p(10) should be called.

The function of esusp is best explained using an example. The following ucode is generated for p(5 | 10)

```
        mark     L1
        var      0        (the variable p)
        mark     L2
        int      0        (constant 5)
        esusp
        goto     L3
lab L2
        int      1        (constant 10)
lab L3
        invoke   1
        unmark   1
lab L1
```

When execution reaches esusp, the stack looks like

| | |
|---|---|
| sp → | descriptor for constant 5 |
| efp → | expression marker with L2 as failure address |
| | descriptor for variable p |
| | expression marker with L1 as failure address |

gfp is zero at this point. After the esusp is performed, the stack is

| | | |
|---|---|---|
| sp → | descriptor for constant 5 | |
| | descriptor for variable p | } duplicated region |
| gfp → | generator frame built by esusp | |
| | descriptor for constant 5 | |
| | expression marker with L2 as failure address | |
| | descriptor for variable p | |
| efp → | expression marker with L1 as failure address | |

A branch is taken to L3, where invoke 1 is performed. This invokes p with one argument, the constant 5 on the stack. If p(5) succeeds, the unmark 1 is performed and the stack is popped back through the L1 expression frame, the current location of efp.

Suppose that instead of succeeding, p(5) fails. p fails by calling pfail, which removes the procedure frame from the stack and then calls efail. The previous stack diagram shows what the stack looks like after the procedure frame has been removed. efail finds that gfp is not null and restores certain values that are saved in the generator frame. The frame, which was created by esusp, contains a return address that points to efail. Thus, when efail removes the frame by returning through it, control goes back to the start of efail and the stack is

| | |
|---|---|
| sp → | descriptor for constant 5 |
| efp → | expression marker with L2 as failure address |
| | descriptor for variable p |
| | expression marker with L1 as failure address |

This time around, gfp is zero, so efail must remove the current expression frame and branch to the failure address in the frame's marker. When the expression frame is removed, the stack looks like

| | |
|---|---|
| sp → | descriptor for variable p |
| | expression marker with L1 as failure address |

The failure address in the expression frame was L2, so control is transferred to label L2 in the ucode. (Note how much went on as the result of the invoke being executed.) The instruction int 1 is executed and a descriptor for the constant 10 is pushed on the stack giving:

| | |
|---|---|
| sp → | descriptor for constant 10 |
| | descriptor for variable p |
| | expression marker with L1 as failure address |

invoke 1 is performed again, which does p(10).

If p(10) succeeds, the unmark 1 is executed, which removes the L1 marker and transfers control to L1. If p(10) fails, the same thing happens, but efail does the work rather than unmark.

**Generic Operation**

(1) The frame created by the call to esusp partially forms the generator frame. The frame is completed by pushing _boundary, _k_level, _line, and _file. The generator frame pointer is set to point at the word of the frame which contains the boundary.

(2) The bounds of the expression frame to be duplicated are determined. The upper bound is the stack word below the current expression frame marker. The lower bound is dependent on efp and gfp values saved in the current expression marker. If the saved gfp is non-zero, the lower bound is the first word above the generator frame marker. If the saved gfp is zero, the lower bound is the first word above the expression frame marker referenced by the saved efp. In the example, this region only contains the descriptor for the variable p. The region is copied to the top of the stack.

(3) The value being suspended is pushed on the stack.

(4) The return address in the new generator frame is replaced by the address of efail so that when efail removes the frame by returning through it, efail regains control. The old return address is momentarily retained. The procedure frame pointer is restored. _boundary is cleared because control is returning to Icon code.

(5) efp in the current expression marker replaces the expression frame pointer. Thus, if an unmark is performed, the entire expression frame is removed. In the example, this happens if p(5) or p(10) succeeds.

(6) The return pc value that was saved earlier is jumped to. This is in effect a return from esusp, but the stack is untouched.

**esusp on the VAX**

esusp is entered from the interpreter loop by a CallCtl and this partially constructs the generator frame. The entry mask directs ipc, gfp, and efp to be saved in the frame. _boundary is set to the current fp value and is pushed on the stack. The generator frame pointer is pointed at the word containing the boundary. The frame is completed by pushing _k_level, _line, and _file on the stack.

The upper bound of the region to copy is the first word below the current expression frame marker. Recall that an expression frame looks like

| | | |
|---|---|---|
| | -8 | failure address |
| | -4 | old generator frame pointer |
| efp → | 0 | old expression frame pointer |

Thus,

```
addl3   $4,efp,r0
```

points r0 at the upper end of the region to copy.

The lower bound of the region to copy is the high word of the marker for the enclosing generator or expression frame. If gfp is non-zero the generator frame marker is used. Otherwise, the expression frame marker is used. Recall that a generator frame looks like

| | | |
|---|---|---|
| | -12 | saved _file |
| | -8 | saved _line |
| | -4 | saved _k_level |
| gfp → | 0 | boundary |
| | 4 | 0 |
| | 8 | psw and register mask |
| | 12 | saved ap |
| | 16 | saved fp |
| | 20 | reactivation address (saved pc) |
| | | saved registers |

So, if the saved gfp is non-zero, the lower bound of the region to copy is

saved gfp - 12

Otherwise, it is

saved efp - 8

The appropriate calculation is performed and r2 pointed at the bounding word.

At this point, the stack looks something like

| | | | |
|---|---|---|---|
| sp → | -12 | _file | generator marker |
| | -8 | _line | |
| | -4 | _k_level | |
| | 0 | boundary (fp at entry to esusp) | |
| | 4 | condition handler address | |
| | 8 | psw and register mask | |
| | 12 | saved ap | |
| | 16 | saved fp | |
| | 20 | reactivation address (saved pc) | |
| | 24 | saved r9 (ipc) | |
| | | saved r10 (gfp) | |
| | -4 | saved r11 (efp) | |
| ap → | 0 | nwords (2) | |
| | 4 | descriptor for value to suspend | |
| | -8 | failure label | expression marker |
| | -4 | saved generator frame pointer | |
| efp → | 0 | saved expression frame pointer | |
| r0 → | 4 | first word of region to copy | |
| | | ... | |
| | | last word of region to copy | |
| r2 → | | high word of expression or generator frame marker | |

The region starting at r0 and extending to r2 is to be copied to the top of the stack. The length of the region in bytes is calculated in r2. The value of r2 is subtracted from sp, moving sp up to accommodate the region. The region is then copied using

```
movc3   r2,(r0),(sp)
```

which moves r2 bytes starting at 0(r0) to 0(sp).

The descriptor for the value to suspend is at Arg_desc and it is pushed on the stack using

```
movq   Arg_desc,-(sp)
```

The stack now looks like

| | |
|---|---|
| sp → | descriptor for value to suspend |
| | first word of copied region |
| | ... |
| | last word of copied region |
| gfp → | generator frame marker |
| | ... |
| | descriptor for value to suspend |
| efp → | expression frame marker |
| r0 → | first word of region to copy |
| | ... |
| | last word of region to copy |
| r2 → | high word of expression or generator frame marker |

The return address that is saved in the generator frame is moved into r1 for later use. It is then replaced by the address of efail so that when the frame is returned through, control will go to efail.

fp and ap are restored from the generator frame. _boundary is cleared because control is returning to Icon code.

efp is pointed at the previous expression frame. That is, efp is moved back one link in the expression frame chain.

Control is returned to the interpreter loop by branching to 0(r1), the reactivation address originally saved in the generator frame.

### 7.6.5 lsusp.s

**Overview**

lsusp suspends a value from a limited expression. A limited expression arises from a source code expression of the form

$expr_1 \setminus expr_2$

This limits $expr_1$ to at most $expr_2$ results ($expr_2$ must have a non-negative integer value).

lsusp is just like esusp except that it has provisions for checking and decrementing the limit counter and taking the appropriate action when the counter reaches zero. As a simple example, consider

```
p(x \ 2)
```

which generates the ucode

```
mark      L1
var       0         (variable p)
int       0         (constant 2)
limit
mark      L0
var       1         (variable x)
lsusp
invoke    1
unmark    1
...
```

When control reaches lsusp, the stack looks like

```
sp  →   descriptor for variable x
efp →   expression marker with L0 as failure label
        descriptor for integer with value of 2
        descriptor for variable p
        expression marker with L1 as failure label
```

The limit instruction insures that the value on the top of the stack (its argument) is a non-negative integer, converting it if necessary. After lsusp, the stack is

```
sp  →   descriptor for variable x
        descriptor for variable p          } duplicated region
gfp →   generator frame built by lsusp
        descriptor for variable x
        expression marker with L0 as failure label
        descriptor for integer with value of 2
        descriptor for variable p
efp →   expression marker with L1 as failure label
```

This is the same thing that esusp would do, with the exception that the limit counter, the integer descriptor, is not part of the duplicated region.

**Generic Operation**

(1) The procedure frame created by the call to esusp partially forms the generator frame.

(2) The limit counter is decremented. If it is zero, no suspension is performed. Instead, the current expression frame is removed and the limit counter is replaced by the value that would have been suspended had the limitation not been in effect. lsusp returns, leaving the value on the top of the stack.

(3) If the limit counter is not zero, execution proceeds exactly as it does for esusp with the exception that the determination of the region to copy takes the limit counter into consideration and does not include it in the region that is copied.

**lsusp on the VAX**

As with esusp, lsusp is entered from the interpreter loop by a CallCtl and this partially constructs the generator frame.

The expression frame and associated limit counter have the following layout:

| | | | |
| --- | --- | --- | --- |
| | −8 | failure label | |
| | −4 | old generator frame pointer | expression frame |
| efp → | 0 | old expression frame pointer | |
| | 4 | D_Integer | |
| | 8 | number of results left | limit counter |

The limit counter is decremented and if it is not zero, control passes to the label dosusp and from then on execution proceeds exactly as it does in esusp. Specifically, the code beginning at dosusp is an exact duplicate of that in esusp with the exception of the instruction that determines the upper bound of the region to be duplicated. esusp uses

```
addl3 $4,efp,r0
```

which points r0 at the word immediately below the expression frame. lsusp uses

```
addl3 $12,efp,r0
```

which points r0 at the word below the limit counter that is in turn directly below the expression frame marker.

If the limit counter is zero, the counter is to be replaced with the value which was to be suspended. The value appears as an argument to lsusp. This is accomplished with

```
movq    Arg_desc,4(efp)
```

The value of gfp that is stored in the expression frame is restored.

The saved pc in lsusp's frame is moved into r0 for later use.

The expression frame is removed by moving efp into sp, which leaves the expression frame marker word that contains the old efp on the top of the stack. This word is popped off the stack and moved into efp, restoring efp and leaving the return value on the top of the stack.

ap and fp are restored from the procedure frame made upon entry to lsusp.

lsusp "returns" by jumping to 0(r0), the return point that was saved in the frame. The value that was to be suspended, but instead was returned because of the limitation, is left on the top of the stack.

### 7.6.6 psusp.s

**Overview**

psusp suspends a result from an Icon procedure. psusp is called from the interpreter loop and the value to suspend appears as an argument. A generator frame is created and the generator or expression frame immediately containing the frame for the suspending procedure is duplicated on top of the stack. psusp simulates a return from the suspending Icon procedure by restoring appropriate registers and values. The net effect is that a generator frame is left on the stack and it appears that the suspending Icon procedure has returned, i.e., the call to invoke seems to have returned.

The psusp operation arises from the

suspend *expr*

expression.

psusp is conceptually similar to esusp, the difference being that a procedure frame is part of the expression frame being duplicated and that requires some extra work. To get a feel for what psusp does, consider a simple example:

```
procedure main()
        f(p(3))
end

procedure p(a)
        suspend a
end
```

The generated ucode for main is

```
            ...
            mark      L1
            var       0            (the variable f)
            var       1            (the variable p)
            int       0            (the constant 3)
            invoke    1
            invoke    1
            ...
lab         L1
```

and the generated code for p is

```
            mark      L2
            mark      L0
            var       0            (the argument a)
            psusp
            ...
lab         L2
```

When control reaches the invoke instruction, the stack resembles

```
sp  →   descriptor for constant 3
        descriptor for variable p
        descriptor for variable f
efp →   expression marker with L1 as failure address
```

After p has been invoked, just before the psusp is executed the stack is

```
sp  →   descriptor for argument a
efp →   expression marker with L0 as failure address
        expression marker with L2 as failure address
        procedure frame for p (created by invoke)
        descriptor for constant 3 (becomes argument a)
        descriptor for variable p
        descriptor for variable f
        expression marker with L1 as failure address
```

Just before control returns from psusp, the stack is

```
sp  →    descriptor for variable f                } duplicated region
gfp →    generator frame built by psusp
         descriptor for constant 3 (argument a after dereferencing)
         expression marker with L0 as failure address
         expression marker with L2 as failure address
         procedure frame for p
         descriptor for constant 3
         descriptor for variable p
         descriptor for variable f
         expression marker with L1 as failure address
```

After psusp returns, the situation is

```
sp  →    descriptor for constant 3 (the suspended value)
         descriptor for variable f
gfp →    generator frame built by psusp
         descriptor for constant 3 (originally argument a)
         expression marker with L0 as failure address
         expression marker with L2 as failure address
         procedure frame for p
         descriptor for constant 3
         descriptor for variable p
         descriptor for variable f
efp →    expression marker with L1 as failure address
```

The return from psusp goes to the second invoke, which calls f with one argument, the constant 3 that was suspended. If f(3) fails, the procedure frame for f is removed. efail takes control and returns through the generator frame built by psusp. This leaves the descriptor for a on top of the stack. Execution continues by p failing, and then main failing.

**Generic Operation**

(1) The procedure frame created by the call to psusp partially forms the generator frame. _boundary is set as the current location of fp and it is added to the generator frame.

(2) As in pret, the value being suspended must be dereferenced in certain cases. For example, if the value is a local variable or an argument, it is dereferenced. The same code that handles dereferencing in pret appears in psusp as well. Note that while suspension leaves the local variables and arguments of a procedure intact, if the enclosing expression frame were to be removed by an unmark, the procedure frame would be destroyed, leaving undeferenced values pointing at meaningless data.

(3) The generator frame is completed by pointing gfp at the boundary value already in the frame and by adding _k_level, _line, and _file.

(4) The bounds of the expression frame to be duplicated are determined. The upper bound is the word below $arg_0$ of the suspending procedure and the lower bound is the marker for the expression frame or generator frame that is just prior to the procedure frame. As in esusp, if gfp is non-zero, the marker it points to is used. Otherwise, the marker referenced by efp is used. The gfp and efp values used are those found in the procedure frame of the suspending procedure. The region is copied to the top of the stack. In the example, the duplicated region contains only the descriptor for the variable f.

(5) If _k_trace is non-zero, strace is called to produce a trace message noting that the procedure is suspending a value. strace requires the address of the block for the suspending procedure and the address of the descriptor for the value being suspended.

(6) _line and _file are restored from the frame of the suspending procedure. This is done because when psusp is finished, it is as if the Icon procedure had returned. Thus, the line number and file name need to be what they were before the procedure was called.

(7) The duplicated region is now on the top of the stack and the value being suspended, Arg_desc, is pushed on the stack. When psusp is done, this descriptor is left on the top of the stack.

(8) _boundary is cleared because control is returning to Icon code.

(9) A return from psusp is simulated by restoring ipc, efp, and other pertinent information from the frame of the suspending procedure. The result is that it appears as if the invoke that originally called the suspending procedure has returned.

A more straightforward but less efficient approach is to include, as the upper end of the duplicated region, the procedure frame for the suspending procedure. psusp then returns through this frame, leaving the value to be suspended on the top of the stack. This is not recommended, but is mentioned because it is used in some implementations.

### psusp on the VAX

psusp is entered from the interpreter loop by a CallCtl and this partially constructs the generator frame. _boundary is set to the current value of fp and this value is pushed on the stack as part of the generator frame.

The value being suspended is dereferenced if it is a local variable or an argument. This operation is the same as is done in pret; consult the section on it for details of the actions taken.

The generator frame is completed by pointing gfp at the frame word containing the boundary value and by adding _k_level, _line, and _file to the frame.

The region to be duplicated is determined. The high word to be copied is the word below $arg_0$ of the suspending procedure.

The low word to be copied is dependent upon the expression and generator environment present at the call of the now-suspending procedure. If the gfp in the suspender's environment is not zero, the word just above the generator frame marker is the lowest word to be copied. If gfp is zero, the word just above the expression marker pointed at by efp in the suspender's environment is the lowest word to be copied.

The istate block of the suspending procedure contains the efp and gfp values of interest. As in esusp, if the saved gfp is non-zero,

saved gfp - 12

is used for the lower bound, otherwise

saved efp - 8

is the lower bound. r4 is pointed at the appropriate word on the lower end. As in esusp, sp is moved up to accommodate the region to be duplicated and the region is copied to the top of the stack using a movc3.

After _k_level is decremented, _k_trace is checked to see if a trace message should be produced. If so, strace is called with pointers to the descriptors for the suspending procedure and the value being suspended. The address of the value being suspended is named by Arg_desc and the address of the descriptor for the procedure is determined using the standard

$\&arg_0$ = ap+Argn_off+(nargs*DescSize)

calculation.

The values of _line and _file are restored from the suspender's frame.

The descriptor for the value being suspended is pushed on the stack with

```
movq    Arg_desc,-(sp)
```

_boundary is cleared because control is going back into Icon code.

ap and fp are restored from the frame of psusp and then the ipc, efp, ap, and fp values are restored from the suspender's frame, serving to mimic a return from the suspending procedure. The ipc now references the icode instruction following the psusp instruction just executed. A branch to interp resumes execution of the program with the suspended value on top of the stack.

Note that since the appropriate registers have been restored and the result is on the top of the stack, it is not correct to branch to the instruction after the call to psusp as that code assumes that registers need to be restored.

### 7.6.7 suspend.s

**Overview**

suspend suspends a value from a built-in procedure or operator. suspend is similar to psusp and amounts to little more than a simplified version of it. Recall that built-in procedures and operators are implemented by C functions; thus, suspend is directly called from C.

A generator frame is created and the generator or expression frame immediately containing the frame for the suspending procedure is duplicated on the top of the stack. As in psusp, a return is simulated, and this appears to be a return from the original call to the C routine.

For built-in procedures, the procedure frame is built by invoke, while for operators, the procedure frame is built directly by the call to the appropriate function from the interpreter loop. The value being suspended by the C function is represented by the arg0 descriptor in the argument list. When suspend is called, the value to suspend is in place in arg0. Note that suspend is only called from top-level routines.

**Generic Operation**

suspend can be considered as a "subset" of psusp. The actions of psusp that are *not* taken by suspend are:

(1) The value being suspended is not dereferenced because the suspending routine created the value and no further action is required.

(2) No tracing message is produced because tracing is only done for Icon procedures.

(3) The value being suspended does not need to be moved into the duplicated region because it is already in place as arg0 of the suspending routine and this value is part of the duplicated region. In psusp, $arg_0$ is not part of the duplicated region and instead is pushed on the stack.

(4) _k_level is not decremented because it keeps track of Icon procedure calls and suspend is returning from a C routine. _line, and _file are not restored because they are not part of the procedure frame of the C routine.

The operations that are performed by suspend are:

(1) The procedure frame created by the call to suspend partially forms the generator frame. _boundary is set as the current location of fp and it is added to the generator frame.

(2) The bounds of the expression frame to be duplicated are determined. $arg_0$ of the suspender's argument list lies at the upper end of the region to duplicate. The lower bound is the marker for the expression or generator frame that is just prior to the procedure frame. As in the other suspension routines, if gfp is non-zero, the marker it points to is used. Otherwise, the marker referenced by efp is used. The gfp and efp values used are those found in the frame of the suspending routine. The region is copied to the top of the stack.

(3) _boundary is cleared because control is returning to Icon code.

(4) A return from suspend is simulated by restoring ipc, efp, and other pertinent information from the frame of the suspending routine. The result is that it appears as if the original call to the suspending routine has returned.

**suspend on the VAX**

When suspend is entered, the generator frame is partially constructed as a result of the call. _boundary is set to the current value of fp and this value is pushed on the stack as part of the generator frame. The generator frame is completed by pointing gfp at the frame word containing the boundary value and by adding _k_level, _line, and _file to the frame.

The region to be duplicated is determined. The high word to be copied is the first word of $arg_0$ of the suspending routine.

As in the other suspension routines, the low word to be copied is dependent upon the expression and generator environment present at the call of the now-suspending procedure. If the gfp in the suspender's environment is not zero, the word just above the generator frame marker is the lowest word to be copied. If gfp is zero, the word just above the expression marker pointed at by efp in the suspender's environment is the lowest word to be copied.

The istate block of the suspending procedure contains the efp and gfp values of interest. As in esusp, if the saved gfp is non-zero,

saved gfp - 12

is used for the lower bound, otherwise

saved efp - 8

is the lower bound. r4 is pointed at the appropriate word on the lower end. As in esusp, sp is moved up to accommodate the region to be duplicated and the region is copied to the top of the stack using a movc3.

_boundary is cleared because control is going back into Icon code.

ap and fp are restored from the frame of suspend and then the ipc, efp, ap, and fp values are restored from the suspender's frame, serving to mimic a return from the suspending routine. The ipc now references the icode instruction following the icode instruction that initiated the call of the now-suspending routine. A branch to interp resumes execution of the program with the suspended value on top of the stack.

### 7.6.8 display.c

#### Overview

display.c implements the Icon function display(). display traces back through Icon procedure frames printing various sorts of information.

#### Generic Operation

display makes one calculation that is machine dependent. The calculation is to take a frame whose address is contained in the variable fp and calculate the address of the procedure descriptor in the frame that is pointed at by the frame pointer value saved in the frame that fp references. display "walks" the arguments and local variables, but code is conditionally compiled to handle the case of up-growing stacks.

#### display on the VAX

ap and fp are restored from the frame referenced by fp. The number of arguments to the procedure is contained in ap[4]. This is loaded into the variable n. The address of the procedure descriptor ($arg_0$) is calculated using:

```
dp = ap+5+(2*n)
```

Note that this is the same computation that is made at several points in the assembly language routines. Because the calculations are being made using int * variables and thus the constants represent word counts instead of byte counts as they do in the assembly language routines.

### 7.6.9 gcollect.s

#### Overview

gcollect is a simple routine that insures that garbage collections are done using the stack for the main co-expression. This done by saving certain values in the co-expression block of the current co-expression, restoring values from the co-expression block for _k_main, calling the garbage collector, and then restoring the original values. gcollect takes a single argument that is passed directly to collect.

### gcollect on the VAX

r0 is pointed at the heap block for the current co-expression. sp, ap, and _boundary are saved in the appropriate words of the block.

r0 is pointed at the heap block for _k_main, the co-expression that is initially active. sp is restored from the block for _k_main. The argument to collect, at 4(ap), is pushed on the stack. Then, ap and _boundary are restored. Note that the argument must be pushed after sp has been restored, but before ap is restored.

collect is called with one argument, which is the argument passed to gcollect.

r0 is pointed at the heap block for the current co-expression and the sp, ap, and _boundary values saved at the start of the routine are restored.

gcollect returns.

## 7.6.10 sweep.c

### Overview

sweep is used during garbage collection to sweep a stack, marking all the descriptors in the stack. sweep begins at the top of a stack and moves down through the stack, looking for descriptors and marking them. A stack is composed of four kinds of objects: descriptors, and markers for procedure, generator, and expression frames. sweep uses knowledge of frame marker formats to skip over markers and to process the intervening descriptors.

Although sweep is written in C, the knowledge of frame formats that it employs requires that it be written on a per-machine basis.

### Generic Operation

There are three places that descriptors can appear on the stack: above an expression marker, in an argument list, and below an argument list. This can be considered as only two places because descriptors below the argument list can be considered as part of the argument list.

sweep is called with a single argument that is the frame pointer value for the frame at the boundary. For purposes of discussion assume that sp references the stack word of current interest. sweep has a loop and each time through the loop, one of four actions is taken based on the word that sp is pointing at:

(1) If sp is pointing at the high word of a procedure frame marker, sp is moved to point at the low word of the argument list of the procedure. efp, gfp, and fp are restored from the procedure frame. The number of arguments to the procedure is placed in nargs.

(2) If sp is pointing at the high word of a generator frame marker, the boundary value in the generator frame is examined to determine if the generator frame was made by suspend or if it was made by one of esusp, lsusp, or psusp (i.e., an Icon generator). If the former, fp is restored from the boundary word of the generator frame, and sp is pointed at the high word of the frame referenced by fp. This skips the C portion of the stack contained in the generator frame and the remainder of the frame can be processed as a procedure frame. The fp value assigned causes the next iteration of the loop to select the procedure frame case.

    If the frame is that of an Icon generator, efp, gfp, and fp are restored from the frame and sp is pointed at the argument descriptor (i.e., Arg_desc) for the routine in question.

(3) If sp is pointing at the low word of an expression frame marker, gfp and efp are restored from the marker and sp is pointed at the word above the marker.

(4) If none of the preceding conditions are true, the word that sp points at is assumed to be the low word of a descriptor and that descriptor is marked. sp is incremented to move past the descriptor. If nargs is not zero, it is decremented.

This process continues as long as fp and nargs are not both zero. nargs is used so that the arguments in the very last frame are processed; the fp at that point is 0.

**sweep on the VAX**

The routine getap is used by sweep. getap takes the address of a frame and returns the address of 0(ap) in that frame. That is, it returns the address of the start of the argument list for the frame.

Note that the C code uses *int* * variables for the various calculations that are performed. Thus, a calculation such as x+2 is actually performing x+8. Similarly, x[-1] would be the address x-4.

sweep is called with a single parameter, fp. fp holds the address of the frame with which to start the marking process. This address is a _boundary value, and thus it points to the condition handler address word of a procedure frame.

sp is set to fp-PFMarkerHigh, so that the first time throughout the loop, the procedure frame on the top of the stack is processed. This gets the ball rolling, so to speak.

sweep loops while fp and nargs are not both zero. It should be noted that the variables used in sweep have no connection to actual registers other than having the same name.

If sp is equal to fp-PFMarkerHigh, it indicates that sp is pointing at a procedure frame marker.

When a procedure frame marker is encountered, efp and gfp values are restored using negative displacements from ap. ap points at the nwords word of the frame, and sp is set to ap+2 so that it points at the descriptor for the first argument. nargs is loaded from the argument list. ap and fp are restored from the frame

A generator frame is indicated by sp being equal to gfp-GFMarkerHigh. fp is restored from the frame. A new ap value is calculated from fp using getap. If fp is equal to gfp+1, a C generator (created by suspend) is at hand, and sp is set to fp-PFMarkerHigh to cause recognition of a procedure frame the next time around.

Otherwise, efp and gfp are restored from ap[-1] and ap[-2] respectively. Then sp is pointed at Arg_desc, the argument descriptor that lies at the word below the nwords word of the frame. ap and fp values are restored from the frame and nargs is set to 1.

An expression frame marker is indicated by sp being equal to efp-EFMarkerHigh. efp and gfp are restored from the marker. sp is incremented by 3 which leaves it pointing at the word above the marker, which may be a descriptor.

If sp suits none of the preceding criteria, it is assumed to point at a descriptor. mark is called with the value of sp as its argument. sp is incremented by 2 to move past the descriptor just marked. If nargs is non-zero, it is decremented.

## 8. Wrapping Up the Port

When the port is believed to be complete, the complete battery of tests can be run by

```
make Samples Testtest
```

in v5. See [3] for information on interpreting the results of these tests.

The porter may also wish to bring up the Icon program library [4] and personalized interpreters [5]. See [3] for instructions for installing and testing these components of the Icon system.

To make the port part of the Icon system in the same manner that other implementations are included, it is necessary to create a directory for it in src, since sys is overwritten in the normal course of installing Icon. To do this, in v5

make Back-in SYS=*name*

where *name* is the host name used in the original Setup-port (see Section 3, *Porting Overview*).

At this point, the new port has the same status as the other implementations. A set-up entry for it can be added to v5/Makefile, using existing entries as guidelines. Note that the -host option, which to this point has been *name*, may need to be changed to something more appropriate for general use [3].

Once this is done, the new system can be set up, installed, and tested like any other implementation of Icon.

## Acknowledgements

Ralph Griswold patiently suffered through a number of drafts on the original version of this document and made innumerable suggestions about grammar, form, and content. Ralph Griswold also served as editor for this version of the document.

Steve Wampler graciously answered a number of questions about the internal workings of Icon and provided a number of comments on the original version of this document.

A number of persons involved with various Icon porting projects have contributed to knowledge about porting this version of Icon that has in turn been incorporated into this document. Thanks go to Rick Fonorow, Phil Kaslo, Mark Langley, Rob McConeghy, and Janalee O'Bagy for contributing in this way.

Special thanks go to Rick Fonorow, Rob McConeghy, and Janalee O'Bagy for using previous versions of this document to successfully port Icon, showing the author that writing this document was worthwhile.


## References


1. R. E. Griswold and W. H. Mitchell, *A Tour Through the C Implementation of Icon; Version 5.10*, Technical Report 85-19, Department of Computer Science, The University of Arizona, August 1985.
2. *VAX Architecture Handbook*, Digital Equipment Corporation, Maynard, Massachusetts, 1982.
3. R. E. Griswold and W. H. Mitchell, *Installation and Maintenance Guide for Version 5.10 of Icon*, Technical Report 85-15, Department of Computer Science, The University of Arizona, August 1985.
4. R. E. Griswold, *The Icon Program Library; Version 5.10*. Technical Report TR 85-18, Department of Computer Science, The University of Arizona. August 1985.
5. R. E. Griswold and W. H. Mitchell, *Personalized Interpreters for Icon; Version 5.10*. Technical Report TR 85-17, Department of Computer Science, The University of Arizona. August 1985.

## Appendix — The Icon Hierarchy

- v5 — root of the Icon system (location is site-dependent)
  - /src — source code for the Icon system
    - /tran — source code for the Icon translator
    - /link — source code for the Icon linker
    - /h — header files for the Icon system
    - /fncs — source code for built-in functions
    - /ops — source code for operators
    - /rt — source code for run-time support routines
    - /lib — source code for routines called by the Icon interpreter
    - /iconx — source code for the Icon interpreter
    - /icont — source code for the Icon command processor
    - /sys — source code for target machine
    - /proto — source code for prototype implementation
    - /att3b — source code for AT&T 3B implementation
    - /pcix — source code for IBM PC/IX implementation
    - /pdp11 — source code for PDP-11 implementation
    - /ridge — source code for Ridge 32 implementation
    - /mc68000 — source code for Sun Workstation implementation
    - /unixpc — source code for AT&T UNIX-PC implementation
    - /vax — source code for VAX implementation
    - /pifncs — source code for Icon library C functions
  - /docs — Icon documentation
  - /book — source code for procedures from the Icon book
  - /bin — executable binaries for Icon
  - /library — Icon program library
    - /src — source code for Icon library programs
      - /cmd — source code for programs
      - /lib — source code for procedure libraries
    - /ibin — executable binaries for programs
    - /ilib — linkable code for procedure libraries
    - /libtest — Icon library test programs
    - /man — manual
      - /man0 — front matter
      - /man1 — application programs
      - /man2 — procedures
      - /man3 — C functions
      - ... — ...
      - /man7 — miscellaneous
      - /man8 — library maintenance
      - /cat0 — formatted front matter for manual
      - /cat1 — formatted pages for application programs
      - ... — ...
  - /rtlib — code for building personalized interpreters
  - /pidem — sample personalized interpreter
  - /samples — Icon installation test programs
  - /test — Icon test suite
  - /port — Icon porting test suite