OpenVMS Technical
Journal V13

OpenVMS I64 TIE Internals: Emulating Alpha Control Instructions

Tim E. Sneddon

Preface

This article is the first of what will hopefully become a series of articles that describe the internals of the OpenVMS I64 Translated Image Environment. In subsequent issues of the OpenVMS Technical Journal it is expected that there will be further articles covering topics such as:

- stack walking and unwinding;
- floating-point emulation;
- the Alpha instruction emulator;
- TIE initialization and setup; and
- image translation.

These are but a few of the topics that are likely to be covered in future articles. Suggestions are also always welcome. See the end of this document for details on contacting the author.

Conventions

Table 1 lists some of the conventions used in this article.

Convention	Meaning
Foreign	This term is used to describe any code that cannot be run natively on OpenVMS I64.
Native	This term describes any code that runs natively on OpenVMS I64
AEST	Alpha Environment Software Translator. The binary translation utility that generates OpenVMS I64 images from OpenVMS Alpha images.
VEST	VAX Environment Software Translator. The binary translation utility that generates OpenVMS Alpha images from OpenVMS VAX images.
TIE	Translate Image Environment. The name given to the execution environment that allows user-mode programs from different OpenVMS supported architectures to be executed.
Procedure Descriptor	This term refers to an OpenVMS Alpha procedure descriptor. This is described in the Starlet module $PDSCDEF.
Function Descriptor	This term refers to an OpenVMS I64 function descriptor. This is described in the Starlet module $FDSCDEF.

Table 1 Conventions Used in this Article

What Has The Translator Generated?

Using certain qualifiers on the AEST translation utility it is possible to observe what happens to the original OpenVMS Alpha image after it has been translated. Figure 1 demonstrates some of the output that can be seen. This example shows the correlation between the original Alpha instructions and the equivalent Itanium instructions.

```
0002006C   BIS             R31,#255,R17    | 0000000000040230h:      { 0:  120080fe840  M           add1    r33 = 0ffh, r0
00020070   BIS             R31,#10,R16     |                           1:  00008000000  F           nop.f   00h
00020074   MULQ            R16,R17,R17     |                           2:  12000014800  I           add1  r32 = 0ah, r0 ;;
00020078   ZAP             R17,#253,R17    |                         dispersal:   0d
0002007C   BIS             R31,#2,R25      |
00020080   LDA             R16,00B3(R2)    | 0000000000040240h:      { 0:  0c708040c80  M           setf.sig  f50 = r32
00020084   LDQ             R26,00D0(R2)    |                           1:  0c708042cc0  M           setf.sig  f51 = r33
00020088   LDQ             R27,00D8(R2)    |                           2:  00008000000  I           nop.i  00h ;;
0002008C   JSR             R26,(R26)       |                         dispersal:   09
                                            |
                                            | 0000000000040250h:      { 0:  00008000000  M           nop.m   00h
                                            |                           1:  1d19b200d00  F           xma.l  f52 = f50, f51, f0
                                            |                           2:  00008000000  I           nop.i  00h ;;
                                            |                         dispersal:   0d
                                            |
                                            | 0000000000040260h:      { 0:  08708068840  M           getf.sig  r33 = f52
                                            |                           1:  00000000000  L           movl  r92 = 0ff00h ;;
                                            |                           2:  0cff0001700  L
                                            |                         dispersal:   05
                                            |
```

Figure 1 Listing of Alpha Instructions and Corresponding Translated Itanium Instructions

This output was generated using the command:

```
$ AEST/VERBOSE/LIST/DUMP=IA64=ALL <image>
```

For more information on the qualifiers accepted by AEST and how to use them, refer to the online DCL help.

Observing Image Execution

With the help of some debugging features built into the TIE run-time library it is possible to observe the execution of the translated image. By defining the logical PRTCHK_PRT_ALL to "1" TIE$SHARE will begin producing a trace to SYS$OUTPUT. If it is inconvenient to have this output written to SYS$OUTPUT, it is possible to define the logical PRTCHK_FILE to an alternate output file. Not all routines generate output, but the important ones do. It is also possible to enable interpretation of all images, even those that had native code generated as part of the translation process. To do this, define the logical TIE$INTERPRET to "1". Figure 2 shows a portion of the output from a simple MACRO-64 program that executes a collection of ZAP and ZAPNOT instructions. It is annotated to give a brief description of the operations taking place.

```
   IAS control flow, magic : 6200 (1)
FUNC CALL: tie$xxxx_lookup, lookup_addr = 000000000002006C (2)
   Address is in TRANSLATED image : ZAP
FUNC RET: tie$xxxx_lookup, returns address : 000000000002006C, TIE$CODE_AXP, new GP 0000000000000000
FUNC CALL: tie$is_jump_to_native, AXP_R27 = 000000007B93C7D0 target_pc = 000000000002006C
FUNC RET: tie$is_jump_to_native, rc = 0
FUNC CALL: tie$get_AXP_FPCR_initial_value
   ieee_fpcr : 0000000000000000
FUNC RET: tie$get_AXP_FPCR_initial_value, fpcr 0000000000000000
FUNC CALL: tie_alpha_emulate, asb->pc = 000000000002006C (3)
   000000000002006C  47FFF411  BIS             R31,#255,R17
   0000000000020070  47E15410  BIS             R31,#10,R16
   0000000000020074  4E110411  MULQ            R16,R17,R17
   0000000000020078  4A3FB611  ZAP             R17,#253,R17
   000000000002007C  47E05419  BIS             R31,#2,R25
   0000000000020080  220200B3  LDA             R16,00B3(R2)
   0000000000020084  A74200D0  LDQ             R26,00D0(R2)
   0000000000020088  A76200D8  LDQ             R27,00D8(R2)
   000000000002008C  6B5A4000  JSR             R26,(R26) ;hint=0000, likely PC=0000000000020090
    R27 is pointing to 000000007B93C7D0 contents FFFFFFFF849986C0
    R26 is pointing to 000000007B93C7D8 contents 000000007BB1A000
   Bingo we have a match all is well - r27==asb_pc (4)
FUNC RET: tie_alpha_emulate, TIE_K_JUMP/BRANCH , jump to : FFFFFFFF849986C0
   IAS control flow, magic : 8100 (5)
```

Figure 2 TIE Run-Time Library Trace Output

1. Execution has just transferred to the TIE$AXP_JUMP_TO glue routine.
2. A lookup is being performed on the address 000000000002006C to determine what kind of code is to be executed.
3. It was determined the code was native Alpha and no equivalent translated code was found. Therefore the Alpha instruction emulator is called and used to execute the code.
4. The last instruction to execute was a JSR. The emulator is now determining how to proceed with execution.
5. The TIE is now looking up the return address specified in the JSR instruction to determine how to procede.

NOTE: AEST also responds to the PRTCHK_PRT_ALL logical and generates quite a bit of output. Before translating an image it is usually a good idea to deassign this logical.

Alpha Register Mapping

To make it easier to understand some of the Itanium code samples, the following table shows the static register mapping used by the TIE to maintain the Alpha register file.

Alpha Register	Usage	Itanium Register
R0	Function value register	R101
R1	Conventional scratch register	R102
R2-R15	Conventional saved registers	R54-R67
R16-R21	Argument registers	R32-R37
R22-R24	Conventional scratch registers	R103-R105
R25	Argument information (AI) register	R106
R26	Return address (RA) register	R68
R27	Procedure value (PV) register	R69
R28	Volatile scratch register	R107
R29	Frame pointer (FP) register	R70
R30	Stack pointer (SP) register	R12
R31	ReadAsZero/Sink (RZ) register	R0*
F0-F1	Floating-point function value register	F32-F33
F2-F9	Conventional saved registers	F16-F23
F10-F15	Conventional scratch registers	F34-F39
F16-F21	Argument registers	F8-F13
F22-F30	Conventional scratch registers	F40-F48
F31	ReadAsZero/Sink register	F0*
MBPR	Mailbox pointer register	R72
FPCR	Floating-point control register	R73
PS	Processor status register	R74
PC	Program counter	R75
	Internal TIE scratch register	R3, R21-R24, R26-R31
	Internal TIE local registers	R76-R80
	Internal TIE translator flag register	R82
	Internal TIE output registers	R108-R115

Table 2 Alpha Register Mapping

If some of these mappings don't appear to follow a logical order it is because many of register mappings were changed late in the design of TIE. Originally it was intended that Alpha registers sharing similar functions as Itanium registers would be mapped together (as was done with VAX registers on the Alpha platform). However, this was eventually changed and so now all Alpha registers (with the exception of the stack pointer) exist in the register stack frame.

The mapping of VAX registers can be found in *OpenVMS Alpha Internals and Data Structures: Scheduling and Process Control*. The mappings have been retained, so all VAX registers map to their original Alpha registers, which in turn map to the equivalent Itanium register.

* Read-only. Writing to Itanium register R0 or F0 results in an Illegal Operation fault.

Introduction

The Translated Image Environment (TIE) is the support environment which executes user mode images compiled and linked on OpenVMS VAX and OpenVMS Alpha that have been subsequently translated for execution on the OpenVMS I64 platform. The translation is achieved using the Alpha Environment Software Translator (AEST) and, in the case of OpenVMS VAX images, the VAX Environment Software Translator (VEST) binary translation tools. While the VEST translator is only available on OpenVMS Alpha, it is supported to translate a VEST'd image using the AEST translator.

These translation utilities generate native images that work with the TIE run-time library to emulate a native OpenVMS Alpha or VAX environment. It is the responsibility of the translator to present the original image in such a fashion that the TIE run-time library can then execute the foreign code. In most cases these utilities are able to generate equivalent native code from the foreign code. It is the branching inside and between these environments that is the focus of this article.

In some areas VAX support is touched on. However, the main focus of this article is the support of the Alpha control instructions. The *OpenVMS Alpha Internals & Data Structures* manual is still a relevant reference as the TIE, present on OpenVMS Alpha for the support of the OpenVMS VAX environment, has been ported to OpenVMS I64 with minimal changes.

Taking a JuMP…

The Alpha architecture presents a collection of closely related control instructions. In the Alpha Architecture Handbook, these are divided as 'Conditional Branch', 'Unconditional Branch' and 'Jumps'. While this division is important to the TIE, a more relevant way to divide them is 'Local' and 'Non-Local'.

Although the two types of branch are handled differently in how they obtain their addresses, all branches within the Alpha environment use the regular conditional branch, 'br', instruction. This is because the emulated environment is contained within a regular OpenVMS I64 frame. The conditional procedure call, or 'br.call', instruction is only used for native calls.

This does make the process of stack walking and unwinding non-trivial. However, by avoiding a true procedure call the emulated Alpha registers continue to be available across Alpha calls. There is no need to consider the consequences of the 'alloc' instruction and how to maintain context. The only time this needs to be considered is when switching environments and that is handled by jacketing procedures, discussed later in this article[†].

Local Branches

Local branches are described as those that are capable of branching forward or backwards a PC relative distance of +/-1M instructions. It also happens that this encompasses all conditional branch instructions. Table 3 summarizes the local control instructions.

Mnemonic	Operation
BEQ	Branch if Register Equal to Zero
BGE	Branch if Register Greater Than or Equal to Zero
BGT	Branch if Register Greater Than Zero
BLBC	Branch if Register Low Bit is Clear
BLBS	Branch if Register Low Bit is Set
BLE	Branch if Register Less Than or Equal to Zero
BLT	Branch if Register Less Than Zero
BNE	Branch if Register Not Equal to Zero
BR	Unconditional Branch
BSR	Branch to Subroutine

Table 3 Local Alpha Control Instructions

These instructions are usually used within a single object module to handle local branching. As such when it comes to translating these instructions to a native instruction sequence there is almost always no execution time lost in the resulting image. This is due to the fact that the branch target is known at the time of translation. However, if the destination of these branch instructions cannot be located within the image being translated then the instruction will be emulated by TIE run-time library via the Alpha instruction emulator.

For all non-emulated branches a small native instruction sequence is generated. Figure 3 shows the sequence generated by AEST when translating an Alpha BNE instruction, it is annotated below.

[†] See the section 'Switching Environments'.

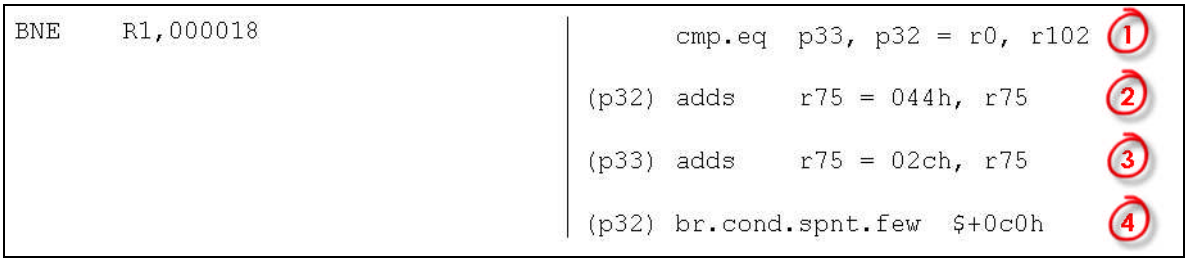

Figure 3 Itanium Code Generated for Alpha BNE Instruction

1. Here the compare part of the BNE instruction is performed.
2. In the event that the comparison yields a positive result, the Alpha PC is set to the address of the branch target.
3. If the comparison is not true, the Alpha PC is updated to point to the next instruction, following the BNE instruction.
4. Lastly, the branch is taken if the result was positive.

All other conditional branches are of the same format, simply substituting the relevant compare instruction relation as necessary.

For the unconditional branches it is even simpler. Figure 4 shows the sequence generated by AEST for a BSR instruction and is annotated below. The only difference between it and a BR instruction is that BSR sets up a return address.

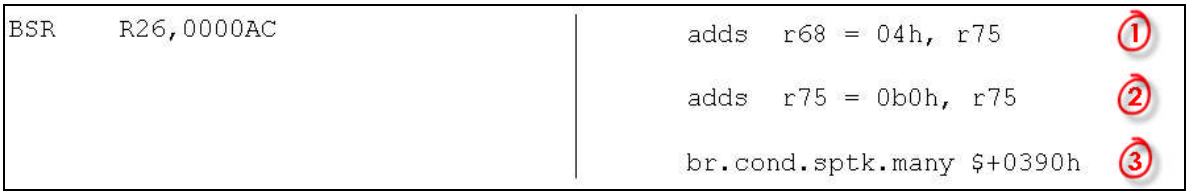

Figure 4 Itanium Code Generated for Alpha BSR Instruction

1. Here the emulated Alpha R26 register is configured with the return emulated PC.
2. The emulated Alpha PC is then updated to point to the first instruction of the destination.
3. Lastly, the branch is taken.

Non-Local Branches

Non-local branches are described as those that take a register argument containing an absolute address, rather than a PC offset. Table 4 summarises the non-local control instructions.

Mnemonic	Operation
JMP	Jump
JSR	Jump to Subroutine
RET	Return from Subroutine
JSR_COROUTINE	Jump to Subroutine Return

Table 4 Non-Local Alpha Control Instructions

The emulation of these instructions does incur quite a bit of overhead as the destination is assumed to be unknown at translation time. This means that these instructions cannot be handled with a small instruction sequence like local branches. It requires support from the TIE run-time library. This comes in the form of the routine TIE$AXP_JUMP_TO. This routine is the first stop on all non-local branches with the exception of JSR. All JSR instructions first branch to TIE$AXP_JSR_TO before falling through to the routine, TIE$AXP_JUMP_TO.

JSR instructions require special pre-processing by TIE$AXP_JSR_TO because in some cases the destination PC for input to the JSR instruction is fetched from a procedure descriptor (see Figure 5b) and not a linkage pair (see Figure 5a). This causes problems when the destination is a native routine as the procedure value register then points to a native function descriptor (see Figure 5c). As can be seen from the illustrations there is no problem in the instance of a function descriptor being mistaken for a linkage pair as the offset to the address of the procedure entry point is the same. However, in the case of the function descriptor being mistaken for a procedure descriptor the global data pointer (GP) for the native image is then loaded and used as the destination PC. Figure 6 compares the two instruction sequences.

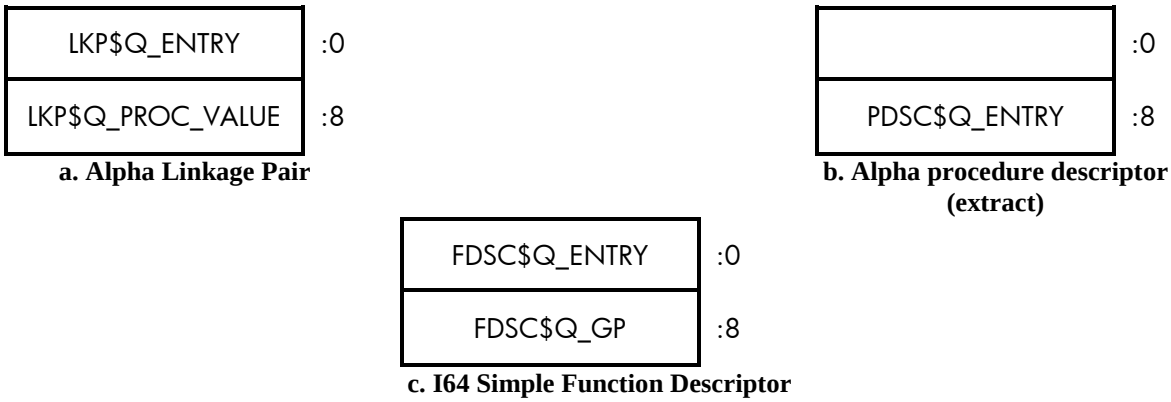

Figure 5 Alpha and I64 Procedure Descriptors

To remedy this when the TIE walks the list of activated images (IAC$GL_IMAGE_LIST) during initialisation gathering details for its own internal list of native images, it also gathers up the GPs and caches them in another list (TIE$CACHED_GPs). When TIE$AXP_JSR_TO is called it attempts to match the destination PC with an entry in this list. If a match is found, then the destination PC is altered by fetching the real entry point address from the function descriptor pointed at by the Alpha register, R27 (procedure value register). At this point TIE$AXP_JSR_TO then falls through to TIE$AXP_JUMP_TO and continues as normal.

```
LDQ    R27, 20(R27)  ; Fetch procedure descriptor
LDQ    R26, 08(R27)  ; Fetch procedure entry point
JSR    R26, (R26)    ; Call procedure
```

Figure 6a. Fetching Entry Point From Descriptor

```
LDQ    R26, 20(R4)    ; Fetch procedure entry point
LDQ    R27, 28(R4)    ; Fetch procedure descriptor
JSR    R26, (R26)     ; Call procedure
```

b. Fetching Entry Point From Linkage Pair

Once the address lookup has been performed, TIE$AXP_JUMP_TO then transfers control appropriate to the type of address being branched to. This may mean starting the Alpha instruction emulator, calling a native routine, branching to another translated Alpha routine or to a pseudo image.

Finding A Place To Go

Jumping to an address may be one thing. However, locating that address to determine how to jump is entirely something else. For the Alpha environment, all call address lookups go through the routine TIE$XXXX_LOOKUP. For the VAX environment the equivalent routine is TIE$VESTED_LOOKUP. It is these routines that have the job of determining what kind of code is at the specified address and locating any translated code that may be associated with it.

For the rest of this section, TIE$XXXX_LOOKUP refers to both the Alpha environment TIE$XXXX_LOOKUP and the VAX environment TIE$VESTED_LOOKUP, unless otherwise specified.

Performing The Lookup

An address lookup begins with TIE$XXXX_LOOKUP first checking that the address being looked up is not the special return address used when returning from native code. In this case the lookup terminates with a return status of TIE$CODE_N2T_RETURN and the caller (usually TIE$ALPHA_TO_IPF or TIE$VAX_TO_IPF) will then begin the process of translating arguments back to their respective environments.

The lookup continues by first checking that this address has not previously been requested. This is done by checking the lookup cache (TIE$GR_LOOKUP_CACHE). To speed up the process of looking up a call address, all successful address lookups and their results are stored in a 4096 entry hash table. This table is managed using the FNV-1a[‡] hash algorithm. In the event that no match is found the lookup routine begins walking internal TIE data structures.

The process starts by first looking up the list of pseudo images loaded by the TIE during initialization. A pseudo image, as the name implies, is not a real image. It exists only as a collection of data structures in memory. Its purpose is to allow the native TIE to intercept lookups by translated or emulated code to TIE routines from previous TIE environments, such as the TIE$SHARE image used to support the VAX environment under OpenVMS Alpha. An example of this is the substitution of the OpenVMS Alpha routine OTS$CALL_PROC with the internal TIE service, TIE$$AXP_OTS$CALL_PROC. The original OTS$CALL_PROC from OpenVMS Alpha has no relevance on OpenVMS I64, so the TIE$$AXP_OTS$CALL_PROC service has been written to act as a substitute. It is simply a wrapper that jumps to TIE$AXP_JUMP_TO.

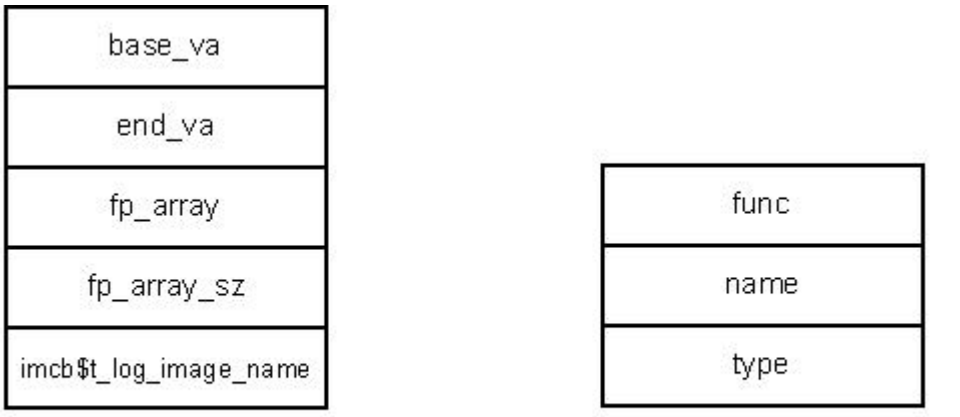

Figure 7 TIE Pseudo Image Data Structures

[‡] See the section 'For more information' for details of the Fowler Noll Vo hash algorithm.

Like almost all internal image data blocks the list of pseudo images is stored in a B-tree. The root of this tree is TIE$PAXP_IMG_DESC_ROOT. Each node in the tree contains a pointer to a TIE pseudo image header (see Figure 7a). These structures are queried using the routine TIE$FIND_PSEUDO_IMG. In the event that the call address being looked up exists in the address space described by the pseudo image, then the routine TIE$FIND_PSEUDO_FP attempts to locate a matching function

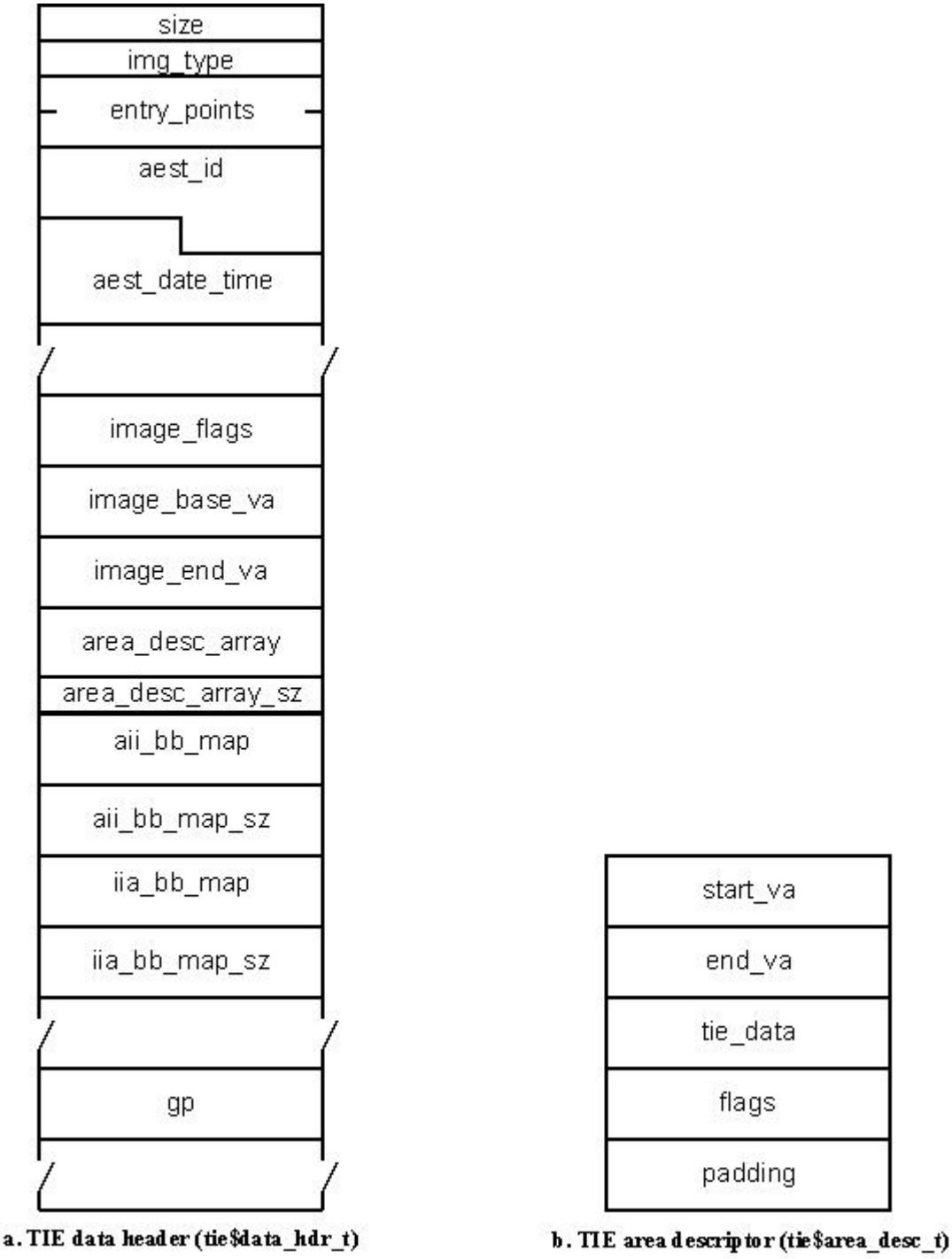

Figure 8 TIE Internal Image Descriptors

The next step is to try and find the call address in the list of loaded translated images. This too is a B-tree list. The root of this tree Is TIE$IMG_DESC_ROOT and is queried using the routine TIE$FIND_IMG. If the call address exists in the address space of this image, then TIE$FIND_IMG

returns a pointer to the TIE data header (see Figure 8a). TIE$FIND_IMG_AREA then uses this structure to locate, using a binary search, the specific area containing the call address and return the corresponding TIE area header (see Figure 8b). The type of code pointed at by the call address can then be determined based on the flags field of the TIE area header. Table 5 shows possible values for the flags field.

Value	Symbolic Name	Meaning
1	tie$vax_code_area_fl	Section contains VAX instructions
2	tie$axp_code_area_fl	Section contains Alpha instructions
4	tie$ipf_code_area_fl	Section contains Itanium instructions
8	tie$axp_nonshraddr_area_fl	Section contains Alpha non-shareable address data

Table 5 Tie Area Descriptor Type Flags

In the case that the call address is found to exist in an area containing foreign code (the flag tie$axp_code_area_fl is set) TIE$XXXX_LOOKUP calls TIE$FIND_AII_BB. This routine performs a binomial search on the list of basic blocks pointed to by the field aii_bb_map. This field points to an array of octawords containing the mapping between Alpha basic blocks and their corresponding native translated blocks. The first quadword contains the address of the Alpha block and the second contains the address of the corresponding native block. It is also possible to perform a reverse mapping using the array pointed at by the iia_bb_map field.

Return Status	Meaning
TIE$CODE_IPF_TRANSLATED TIE$CODE_VAX_TRANSLATED[#]	The lookup address either points to translated native code, or a foreign basic block that had a corresponding translated block and TIE$INTERPRET is not active. The address returned points to translated native code.
TIE$CODE_VESTED_AXP[#]	The lookup address points to VAX code that has corresponding translated Alpha code, but no valid native translated code. The return address points to the Alpha code.
TIE$CODE_AXP TIE$CODE_VAX	The lookup address points to foreign code that either has no corresponding translated code, or TIE$INTERPRET is in effect. The return address matches the lookup address.
TIE$CODE_OUTSIDE	The lookup address points to code outside of any translated image. The return address matches the lookup address.
TIE$CODE_PSEUDO_AXP	When the lookup address is in a pseudo image.
TIE$CODE_PSEUDO_VAX[#] TIE$CODE_VAX_SYS_SERV[#]	When the lookup address points to a VAX system service entry point. The returned address is the corresponding native system service entry point.
TIE$CODE_AXP_TIE[#]	The lookup address points to an Alpha TIE image.
TIE$CODE_N2T_RETURN	The lookup address points to the return address used by TIE$NATIVE_TO_TRANSLATED

[#]These status codes are returned only by TIE$VESTED_LOOKUP.

Table 6 TIE$XXXX_LOOKUP and TIE$VESTED_LOOKUP Return Codes

For the VAX environment lookups are performed using the routine TIE$FIND_VAX_BB. This routine performs a binomial search of the basic block mapping list pointed at by viai_bb_map in the TIE data header (not shown in Figure 8a).

In the event that no match is found at all TIE$XXXX_LOOKUP assumes that the address is native and returns the status TIE$CODE_OUTSIDE. Table 6 shows possible values returned by TIE$XXXX_LOOKUP and TIE$VESTED_LOOKUP.

Switching Environments

The final stage in transferring control is the actual branch. As stated in previous sections, transferring from one foreign procedure to another (in the same processor environment) is nothing special. It is handled with a simple condition branch instruction. The real complexity starts when transferring control to another environment and trying to hide that from translated software.

Translated to Native

If a call address is determined to exist in a native image TIE$AXP_JUMP_TO transfers control by branching to TIE$ALPHA_TO_IPF. This routine is the final stepping stone for all translated code transferring control to the native OpenVMS I64 environment. The process begins by extracting the call signature information[§] from the callee's function descriptor. Figure 9 demonstrates the program logic that determines the final argument information that is used to process the argument list.

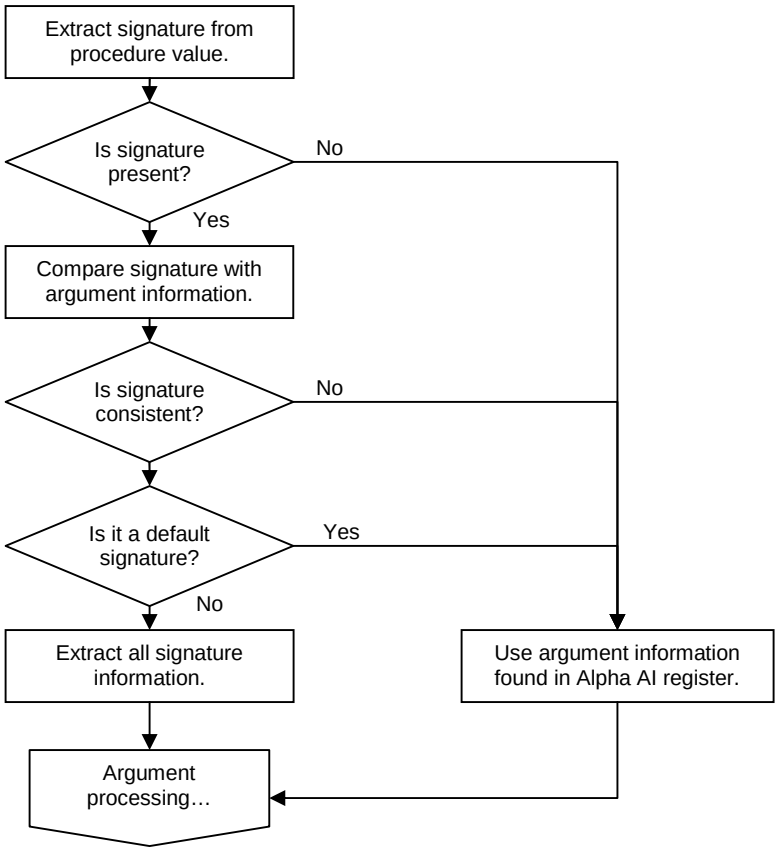

Figure 9 Logic to Determine Argument Information

As Figure 9 shows, in the event that the signature is deemed unusable the fail over is to use argument information from the Alpha AI register (R25). The difference between the call signature and the argument information register is that a signature provides complete details of all arguments and return values. It is present purposefully for the translated environment and is usually only available in images that contain code that has been compiled with the /TIE qualifier. Argument information from the AI register, on the other hand, is available in all calls. However, it only details the format of the arguments present in registers.

[§] See the *HP OpenVMS Calling Standard,* referenced in 'For more information' for further details.

The convention of using the argument information register as the default signature actually defies the calling standard. It states that when transferring control from the Alpha TIE to a native I64 procedure the default signature specifies that it should be assumed that all register arguments are RASE$K_RA_I32 and all memory arguments are MASE$K_MA_I32 (32 bit integers, sign extended to 64 bits). By using the argument information register, the default argument type is actually assumed to be AI$K_AR_I64 (64 bit integer). The default function result is also assumed to be PSIG$K_FR_I64 (64 bit integer).

Once all argument information has been extracted it is now necessary to prepare the arguments for the call to a native procedure. While both architectures employ similar methods in argument passing, the differences are still far too great. Figure 10 shows the logic used to convert Alpha register arguments to their native I64 counterparts.

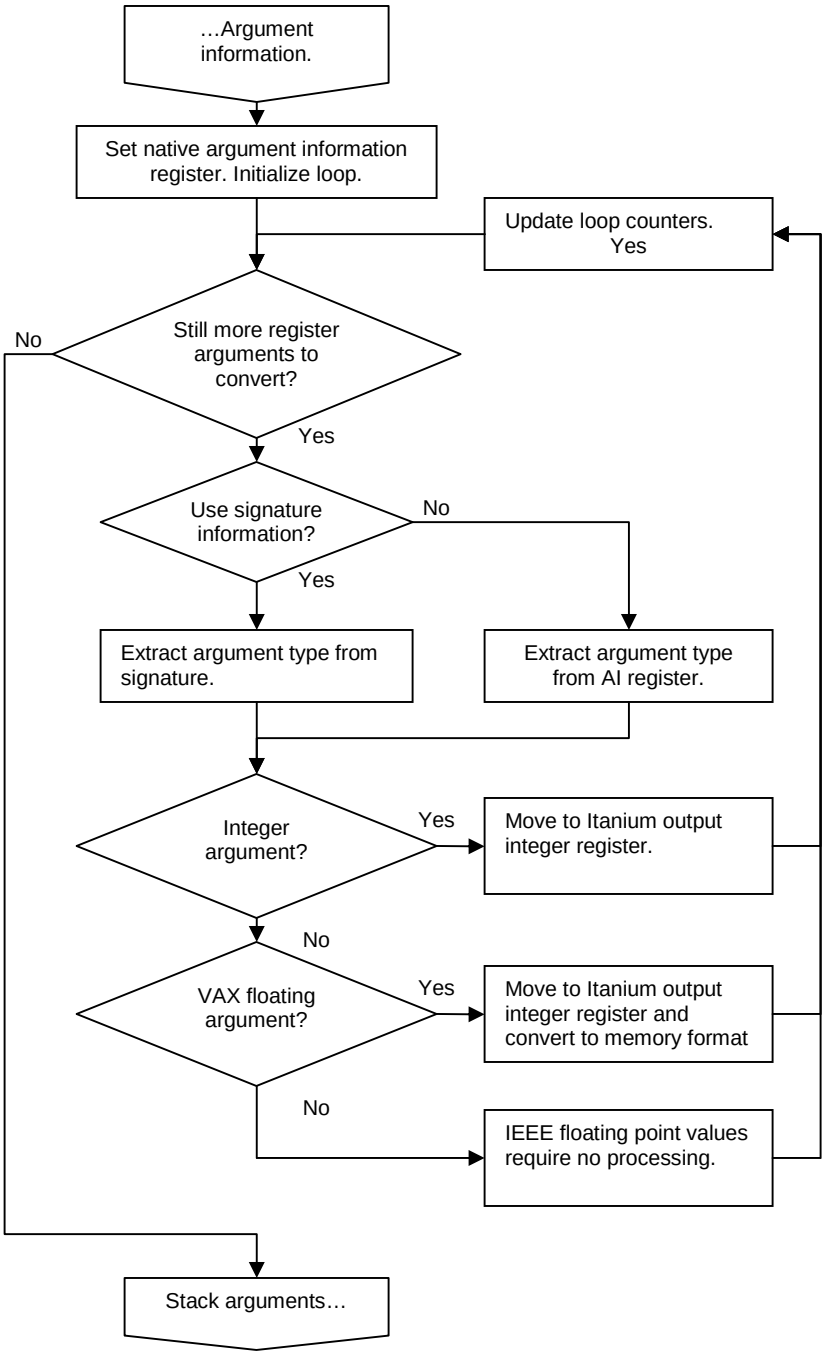

Figure 10 Register Argument Conversion Logic

Floating point arguments on the Alpha, like the Itanium, are passed in the floating point registers. However, the OpenVMS I64 calling standard requires that VAX floating point numbers (F_FLOAT, G_FLOAT and D_FLOAT) must be passed in general registers in memory format. TIE translated code stores all floating point numbers in register format in native floating point registers. Therefore all any arguments must be moved to general registers and converted.

It is also a requirement of the Itanium architecture that a rotating region of output registers be used to pass arguments to the callee. Therefore it is necessary to copy all arguments in the Alpha registers R16-R21 (at least as many as the AI register dictates) into these output registers.

Lastly, there are less argument registers on the Alpha than there are on the Itanium. How this is handled is not shown in Figure 10. However, the process is quite simple. The arguments in slots 7 and 8 (stacked on the Alpha) are fetched and stored in their corresponding output registers. Again, it is necessary to consider the type of the argument to ensure it is stored in the correct register in the correct format. Moving these two arguments into register does have an advantage. The I64 calling standard requires that the caller leave a 16 byte scratch area before stacking any extra arguments passed to the callee. The stack locations formerly held by arguments 7 and 8 can then be used for this region preventing the need to re-shuffle the remaining stacked arguments. For calls with less than 8 arguments this scratch region is allocated by adjusting the stack pointer.

Once the arguments have been prepared the Alpha procedure descriptor is examined to determine the exception mode. Depending on the value in PDSC$V_EXCEPTION_MODE the call may pass through a wrapper that configures the floating point status application register (as.fpsr).

The final step before making the actual call is to move the contents of the Alpha registers R0 and R1 into their corresponding Itanium register, R8 and R9. Both of these registers can be used by compilers to pass information between calls. R1 (R9 on the Itanium) is often used to pass a pointer to the callers automatic storage to support uplevel references.

On returning from the native routine it is necessary to go through the process of converting the return value to an acceptable format. For return values the Itanium register R8 is copied into the Alpha register R0. For return values specified as PSIG$K_FR_D64, this also means that the contents of Itanium register R9 are moved to Alpha register R9. For all return values specified as VAX floating point they are converted back to register format and moved from the Itanium register F8 to Alpha register F0. For all complex values, the contents of Itanium register F9 is also moved in to Alpha register R1. For IEEE values there is no conversion. The contents of the Itanium registers F8 and F9 are copied to their Alpha equivalents. The register F9 is only copied in the case of a complex result.

At this point TIE$ALPHA_TO_IPF has completed the native call. The only thing left to do is to branch to the Alpha return address. This is done by branching to TIE$AXP_JUMP_TO.

Native to Translated

Although branching from native code into TIE might not strictly be emulation of a JSR instruction, it certainly needs to appear that way to the TIE. All transfers from native to translated code should be done through the routine OTS$CALL_PROC/TIE$NATIVE_TO_TRANSLATED**. While this routine is largely a wrapper, it is the interface used by all OpenVMS compilers. To call a native routine without using a compiler requires the call to be coded in Itanium assembly.

To begin, OTS$CALL_PROC calls TIE$NATIVE_TO_TRANSLATED which determines the environment it needs to transfer control to using the routine TIE$PROC_KIND**. For calls destined for the Alpha

** See the section 'Routine Reference'.

environment this means a branch to TIE$IPF_TO_ALPHA (for the VAX environment it is TIE$IPF_TO_VAX). This routine is responsible for the translation of all arguments and return values when going from native to translated code and back again. It can be thought of as the reverse of TIE$ALPHA_TO_IPF.

TIE$IPF_TO_ALPHA begins in much the same way as TIE$ALPHA_TO_IPF. It fetches the procedure signature and verifies it. If the signature is not found or invalid the argument information is retrieved from the Itanium argument information register (R25). The next step is to translate all the arguments as required. In the case of IEEE floating point and integer values there is no change. VAX floating point numbers are converted to register format and stored in the relevant floating point register. The 16 byte scratch region is used again. This time it holds any values found in the last two general register argument slots.

Once all arguments and the stack are prepared some Alpha registers are configured. The Alpha frame pointer is set and the Alpha return address is set to a special address that begins the process of converting return values. The contents of the Itanium registers R8 and R9 are also copied into the Alpha registers R0 and R1. The execution of the Alpha routine is then started by branching to TIE$AXP_JUMP_TO.

On return from the Alpha routine the return values are converted back to meet the requirements of the native OpenVMS I64 environment. This is done in much the same way as TIE$ALPHA_TO_IPF. The final action of TIE$IPF_TO_ALPHA is to branch back to the caller.

Routine Reference

The following are some quick notes on the calling details of the publicly defined TIE services mentioned in this article.

TIE$PROC_KIND

This routine determines the 'kind' of a procedure address. This routine is used by OTS$CALL_PROC/TIE$NATIVE_TO_TRANSLATED to determine if it should transfer control to the TIE or not.

Parameters:

ofd.rq.v	Address of an official function descriptor. For native images this argument is the address of a function descriptor. In the case of translated Alpha images this is the address of a procedure descriptor. Lastly, for translated VAX images this is a pointer to the original VAX code.

Completion Codes:

The following table describes possible return values of this function. The symbolic names are found in [IA64_TIE]TIE_DEFS.H and not publicly available.

Value	Symbolic Name	Description
0	TIE$K_PROC_NATIVE	Address points to a native function descriptor.
1	TIE$K_PROC_ALPHA	Address points to an Alpha procedure descriptor.
2	TIE$K_PROC_VAX	Address points to a VAX procedure.

In the event that the function descriptor is not readable this routine will signal TIE$_WRONG_PV.

OTS$CALL_PROC/TIE$NATIVE_TO_TRANSLATED

The routine OTS$CALL_PROC is used by all OpenVMS compilers when calling external routines and the /TIE qualifier is active. In the event that the call is user mode and the TIE is active (CTL$GQ_TIE_SYMVECT) is non-zero) then OTS$CALL_PROC transfers control to TIE$NATIVE_TO_TRANSLATED (also known as TIE$CALL_PROC).

Parameters:

p1, …pn	All arguments are passed as if directly calling the routine in question. Floating point arguments are all passed in f16-f23. All other arguments are passed in r32-r39.
sig.rr.r	Address of the call argument signature block. This provides details of the arguments being passed. This is passed in register r17.
ofd.rr.r	The destination function descriptor. This is passed in register r18.
ai.rq.v	Argument information. This is passed in through register r25.
ra.ra.v	Return address. This is passed in the register b0.

In the case of calling a translated routine, registers r8 and r9 (up level environment value) are copied into R0 and R1 of the translated environment. They are then copied back on return.

Completion Codes:

Only the completion codes of the routine being indirectly called.

Relevant Sources

The following list comprises the source modules from the OpenVMS I64 source tree that are discussed in this article.

- [IA64_TIE]AUX_CALLING_STANDARD.S
- [IA64_TIE]TIEDATA.H
- [IA64_TIE]TIE_CONT.C
- [IA64_TIE]TIE_CONT.H
- [IA64_TIE]TIE_DEFS.H
- [IA64_TIE]TIE_IMGSUP.C
- [IA64_TIE]TIE_JACKETS.S
- [IA64_TIE]TIE_JUMPS.S
- [LIBOTS]OTS$CALL_PROC_IA64.IAS
- [STARLET]FDSCDEF.SDL
- [STARLET]PDSCDEF.SDL
- [STARLET]PSIGDEF.SDL

For more information

Tim Sneddon can be contacted via email at tim.sneddon@bigpond.com.

For further information regarding the OpenVMS Alpha TIE and the VEST binary translation tools, please see the following:

- Goldenburg, R., Saravanan, S., Duma, D. *OpenVMS Alpha Internals and Data Structures: Scheduling and Process Control: V7.0* (1997) Digital Press ISBN: 978-1555581565
- Sites, R. L., Chernoff, A., Kirk, M. B., Marks, M. P., Robinson, S. G. *Binary Translation, Digital Technical Journal Vol. 4 No.4* (1992)
    - http://www.dtjcd.vmsresource.org.uk/pdfs/dtj_v04-04_1992.pdf
- *HP OpenVMS Migration Software for VAX to Alpha Systems (OMSVA) Documentation*
    - http://h71000.www7.hp.com/openvms/products/omsva/omsva.html
- For those readers with access to the OpenVMS Alpha source listings (or the source itself), the source for the TIE$SHARE.EXE run-time library can be found in the [TIE] facility.

For further information regarding the VAX architecture and instruction set, please see the following:

- *VAX Architecture Reference Manual* (1991) Digital Press ISBN: 978-1555580575

For further information regarding the OpenVMS I64 TIE and the AEST binary translation tool, please see the following:

- *HP OpenVMS Calling Standard*
    - http://h71000.www7.hp.com/doc/os83_index.html
- *HP OpenVMS Migration Software for HP AlphaServer Systems to HP Integrity Servers (MSAIS) Documentation*
    - http://h71000.www7.hp.com/openvms/products/omsva/omsais.html
- For those readers with access to the OpenVMS I64 source listings (or the source itself), the source for the TIE$SHARE.EXE run-time library can be found in the [IA64_TIE] facility.

For further information regarding the Alpha architecture, please see the following:

- Alpha Architecture Committee, Witek, R. T., *Alpha Architecture Reference Manual (Third Edition)* (1998) Digital Press ISBN: 978-1555582029

For further information regarding the Itanium, please see the following:

- Evans, J. S., Trimper, G. L. *Itanium Architecture for Programmers: Understanding 64-bit Processors and EPIC Principles* (2003) Hewlett-Packard Books ISBN: 0-13-101372-6
- *Intel® Itanium® Architecture Software Developer's Manual* (2006) Intel Corporation
    - http://www.intel.com/design/itanium/manuals/iiasdmanual.htm

For further information on the Fowler Noll Vo hashing algorithm, please see the following:

- Wikipedia, *Fowler Noll Vo hash*
    - http://en.wikipedia.org/wiki/Fowler_Noll_Vo_hash
- Noll, L. C., *FNV hash*
    - http://isthe.com/chongo/tech/comp/fnv/