OpenVMS Technical Journal V5

# Porting the Macro-32 Compiler to OpenVMS I64

# Porting the Macro-32 Compiler to OpenVMS I64

John Reagan, Macro-32 Project Leader

## Overview

In June 2001, the OpenVMS Engineering organization began porting OpenVMS from the Alpha architecture to the Itanium architecture. One of the key components required was the Macro-32 compiler. Porting the compiler from OpenVMS Alpha to OpenVMS I64 presented several challenges and problems. This paper describes the porting of the Macro-32 compiler from OpenVMS Alpha to OpenVMS I64.

## History of the Macro-32 Compiler for OpenVMS Alpha

A large portion of OpenVMS is written in Macro-32. When OpenVMS was first ported from VAX to Alpha, a Macro-32 compiler was created that would accept Macro-32 source code and produce Alpha object files. This enabled much of the Macro-32 source code to be ported from VAX to Alpha with only mechanical changes to the source code. Usually, Macro-32 source code had to be enhanced to include new directives (such as ".call_entry" and ".jsb_entry") and to recode several VAX code constructs that could not be supported by the Macro-32 compiler on Alpha.

## Goals of the Macro-32 Compiler for OpenVMS I64

Unlike porting from OpenVMS VAX to OpenVMS Alpha which often required source code changes, our goal for porting Macro-32 code from OpenVMS Alpha to OpenVMS I64 was that modules would simply recompile with no source changes whatsoever. We hoped that the directives that were added to Macro-32 source files when ported from VAX to Alpha would be sufficient for compiling the code on I64. In the end, it turned out that we needed to add a few additional directives and a handful of source modules would need modification to use them.

## Organization of the Macro-32 Compiler for OpenVMS Alpha

The Macro-32 compiler has four main phases:

- Phase 1: Source Parser
  The source parser is the same parser that is used by the Macro-32 assembler on OpenVMS VAX. The parser, written in Macro-32, tokenizes the source lines into an intermediate tuple stream. This tuple stream includes instructions, register references, memory references, symbol assignments, conditional compilation information, and labels.
- Phase 2: Flow Analyzer
  The flow analyzer, written in C, analyzes the tuple stream to identify loops, register usage, condition code usage, and breaks up the instruction stream into a sequence of flow blocks where each flow block begins with a label and ends with a branch or call. The register analysis identifies which registers are input, output, read, or written in a flow block. This information is later used to identify registers that can be used by the code generator as short-term scratch registers. The condition code analysis optimizes the generated code by materializing the equivalent VAX condition codes only when they are used by subsequent instructions.
- Phase 3: Code Generator and Register Allocator
  The code generator and register allocator, written in BLISS, processes the flow blocks produced by the flow analyzer. For each VAX instruction in the flow blocks, the code generator produces Alpha instructions to produce the equivalent behavior. The output from the code generator is a list of code cells where each code cell might be an Alpha instruction with its operands or a label. The register allocator keeps tracks of temporary registers used by the code generator. Most of the temporary registers are used only during the code corresponding to a single VAX instruction. However, some of the temporary registers represent condition codes and may have to live for multiple instructions or even around a loop.
- Phase 4: Instruction Scheduler and Peephole Optimizer
  The instruction scheduler and peephole optimizer are provided by the GEM backend code

generator used by the other Alpha compilers. However, in those compilers, the corresponding language front ends produce a sequence of intermediate language tuples and GEM performs its own flow analysis and code generation. In the case of Macro-32, those components of GEM are not used. The Macro-32 compiler produces the list of code cells itself and passes them to the final phases of GEM. The instruction scheduler reorders instructions for better performance. The peephole optimizer removes or modifies related instructions for better performance. GEM also produces the debug information and writes the object file.

**What Changed and What Stayed the Same?**

For porting the Macro-32 compiler from Alpha to I64, each phase required a different set of changes with two phases requiring almost no work on our part.

The parser required only two source lines to be changed to identify a CALLS instruction to a routine that returned values in registers other than R0 or R1. Such a construct requires the use of a new directive for OpenVMS I64 (the directive is ignored by the compiler on Alpha). The resulting parser is common code between the Alpha and I64 compilers.

The instruction scheduler and peephole optimizer provided by GEM still accepted code cells as input but those code cells now represented Itanium instructions rather than Alpha instructions. There is a different GEM for each target. While the GEM team spent many person years developing the code, from the Macro-32 compiler's point of view, the interface stayed basically the same with the addition of a handful of new flags in the GEM symbol table.

**Flow Analyzer Changes**

The main tasks of the flow analyzer of grouping tuples into flow blocks and identifying registers used in each block remained essentially the same and required no changes.

However, the flow analyzer also is aware of how many routine arguments are passed in registers and which registers are preserved around routine calls. These values are different from Alpha to I64. For instance, on Alpha the first six arguments are passed in registers while on I64 the first eight arguments are passed in registers. Likewise, on Alpha a standard routine preserves registers R2-R15 by default while on I64 only registers R4-R7 are preserved by a standard routine.

Unfortunately, when the Alpha Macro-32 compiler was written, the information was coded with literal numbers such as "6" or as literal integer or hexadecimal masks. We had to hand examine the entire flow analyzer looking for literals like "5", "6", "7", etc. and change them into symbolic constants which expand to the correct value depending on the target of the compiler.

We added one significant feature to the flow analyzer to deal with a unique feature of the Itanium architecture. Itanium integer registers are actually 65 bits wide. They contain 64 bits of data and another bit called the NaT bit (Not a Thing). This bit identifies a register that has not been initialized and doesn't contain a value. When a register that contains a NaT is written to memory with the regular Itanium store instructions, the hardware raises a NaT Consumption Fault. To avoid this error, there are special Itanium instructions that must be used to spill all 65 bits of a register to memory.

In Macro-32 routines, we found a common practice of what we call a courtesy save. This is code, usually in JSB routines, that explicitly does a PUSHL of some register, then uses that register for some local purpose, and then does a POPL to restore that register. The save sequence is always executed, it does not matter if the caller was actually using that register. On VAX and Alpha, for callers that did not use that register, the save sequence simply saved and restored some random value. However, on Itanium, if that register happened to contain a NaT, the PUSHL would result in a NaT consumption fault. If that occurs in kernel mode, OpenVMS will crash. We found this out the hard way.

We could not use the special 65 bit register save instructions since Macro-32 code that is building a data structure on the stack or is pushing arguments for an upcoming routine call would only expect 32 bits to be written. So we decided that if we could identify registers written to memory that appear to be courtesy saves (writes to memory with no prior writes to that register), then we would generate additional code in the routine prologue to detect a NaT and write a negative one (-1) into that register. In addition, if the register was one of the Itanium preserved registers (R4-R7), we would have to restore the NaT at routine exit.

The flow analyzer was enhanced to find these courtesy saves and propagate the information between flow blocks. The result was a mask of registers that would need NaT Guarding in the routine prologue.

**Instruction Generation**

The code generator processes each instruction tuple in the flow blocks built by the flow analyzer and generates either Alpha or Itanium instructions. The code generator phase is the only place where we decided not to use common code but have separate modules for Alpha and I64. We cloned the eight Alpha-specific modules and then re-implemented every routine in each module. Some instructions were easy (for example, MOVL and ADDL) while others (DIV, FFS, INSV, EXTZV, EVAX_LDQ_L, and others) took several times to get correct. One subtle difference between the Alpha instructions and the I64 instruction is that literal operands in the Alpha instructions are zero-extended while literal operands in the I64 instructions tend to be sign-extended. We also found several cases where the compiler on Alpha had extended traditional VAX instructions to access part of the Alpha architecture. For example, allowing the BBC instruction to branch on a bit larger than 31 in a register. None of these were ever documented by the Alpha compiler and were only found after problem reports from OpenVMS I64 testing. After that experience, we were more careful about checking our results both against the VAX architecture and the Alpha compiler behavior.

The Alpha compiler has a large set of EVAX_ built-ins to provide access to Alpha instructions. In almost all cases, we support those built-ins by generating one or more Itanium instructions to do the same task. However, some of the EVAX_ built-ins are PAL calls on Alpha. Since there is no PAL code on I64, new system services were added to OpenVMS to provide the similar functionality. The PAL support was removed from the compiler (with the exception of the queue instructions) and new macros were provided in STARLET.MLB that expand to the appropriate system service. The end result is that Macro-32 source code believes that it is using Alpha EVAX_ built-ins to access PAL code when in fact it is calling newly written system services.

Given the relatively small number of available registers on Alpha, the Alpha compiler has extensive code to spill registers to the stack if the generated code requires more temporaries than are currently available. We were able to delete all of that code but had to extend our register allocation mechanisms to include predicate registers as well as distinguishing between the lower 32 static registers and the higher 96 stacked registers. Each routine on I64 can have up to 128 registers. In addition, because the I64 output registers vary based on the most recently executed 'alloc' instruction, we added analysis to ensure that routines that jumped between each other had compatible output registers numbers.

**Calling Standard Differences**

The Calling Standard for I64 defines registers R8 and R9 as the return value registers. However, all existing Macro-32 source code has assumed that it is dealing with either the VAX or Alpha calling standards and specifies R0 and R1 as the return value registers. Since our goal was to just recompile Macro-32 code from Alpha, we invented a register mapping table. With this register mapping, the compiler on I64 maps all references to R0 and R1 to R8 and R9, respectively. Once we moved R0 and R1, we had to move many of the other registers to make the puzzle all fit together. The end result is that Macro-32 source code believes that it is using Alpha-numbered registers and the compiler's register mapping silently adjusts to match the I64 calling standard. The complete mapping table is available in the HP OpenVMS Macro Compiler Porting and Users Guide in the OpenVMS documentation set.

Most existing Alpha Macro-32 code has been written with the Alpha calling standard in mind. Programmers have assumed that registers R2 through R15 will be preserved by calls to external routines, especially those written in another language like C or FORTRAN. However, on I64, only registers R4 through R7 are preserved by the calling standard. Since we did not want to make people change their source code between Alpha and I64, the compiler automatically preserves R2, R3, and R8 through R15 around each CALLS and CALLG instruction. In general, this behavior is correct. However, for routines that return values in registers other than R0 and R1, the register saves and restores might undo a non-standard return value. In order to support these routines, we had to invent several new directives to provide linkage information about the called routine. These new directives, named ".call_linkage", ".define_linkage", and ".use_linkage", tell the compiler about the non-standard output register usage of the target routine.

In a similar situation, it is possible on Alpha  to use a JSB instruction to call a routine that is not written in Macro-32.  For instance, if the target routine is written in C, the Alpha calling standard requires the C compiler to preserve registers R2 through R15.  Calling this routine with a JSB instruction works correctly.  However, on I64, the C compiler preserves only R4 through R7.  The Macro-32 program would suddenly find that registers R2, R3, and R8 through R15 would be  corrupted by JSB'ing to a C routine.  Again, the new directives have an option to indicate that the target routine is written in a language other than Macro-32.  In that case, the compiler will preserve R2, R3, and R8 and R15 even around the JSB instruction.

These new directives are also used by the system macros that expand to the system services that replace the Alpha PAL code.  For calls to those routines, the compiler must save many more registers to emulate the Alpha behavior of PAL calls not modifying any register, including those higher than R15.

### Floating and Packed Decimal Instruction Support

The Alpha compiler supports VAX floating and packed decimal instructions through a set of macros and helper routines written in Macro-64.  These routines were re-implemented in Itanium assembly code and the macros were modified to use a different calling sequence.

### AMACRO Utility Routines

All of the AMAC$ utility routines from Alpha had to be either rewritten or at least modified to deal with the parameter passing mechanism on I64.

### Summary

Porting the Macro-32 compiler from Alpha to I64 was a challenge since we were learning the Itanium architecture and also learning the internals of the Macro-32 compiler.  We came close to our original goal of being able to recompile Macro-32 code from Alpha.  The vast majority of the Macro-32 code in the OpenVMS source pool recompiled without modification.

### Acknowledgements

Peter Haynes and Karl Puder worked on porting the Macro-32 compiler along with the author.  Each contributed significant effort to the final compiler.  Greg Jordan helped with initial register mapping table.  Greg and Christian Moser suggested the Itanium code used to emulate the Alpha load-locked and store-conditional instructions.