vms Software

Compiler Future Directions

September 28 , 2015

vms Software

Compiler Future Directions

This information contains forward looking statements and is provided solely for your convenience. While the information herein is based on our current best estimates, such information is subject to change without notice.

Agenda

- What compilers are needed for the port?
- What do our compilers use today?
- Which code generator to use for x86-64?
- How to get from today to x86-64?
  - Leveraging GEM
  - Define the Calling Standard
  - What else is needed for x86-64?
- How does the port impact compilers?
  - C++ & RTLs
  - C & RTLs
  - Macro-32
  - Others

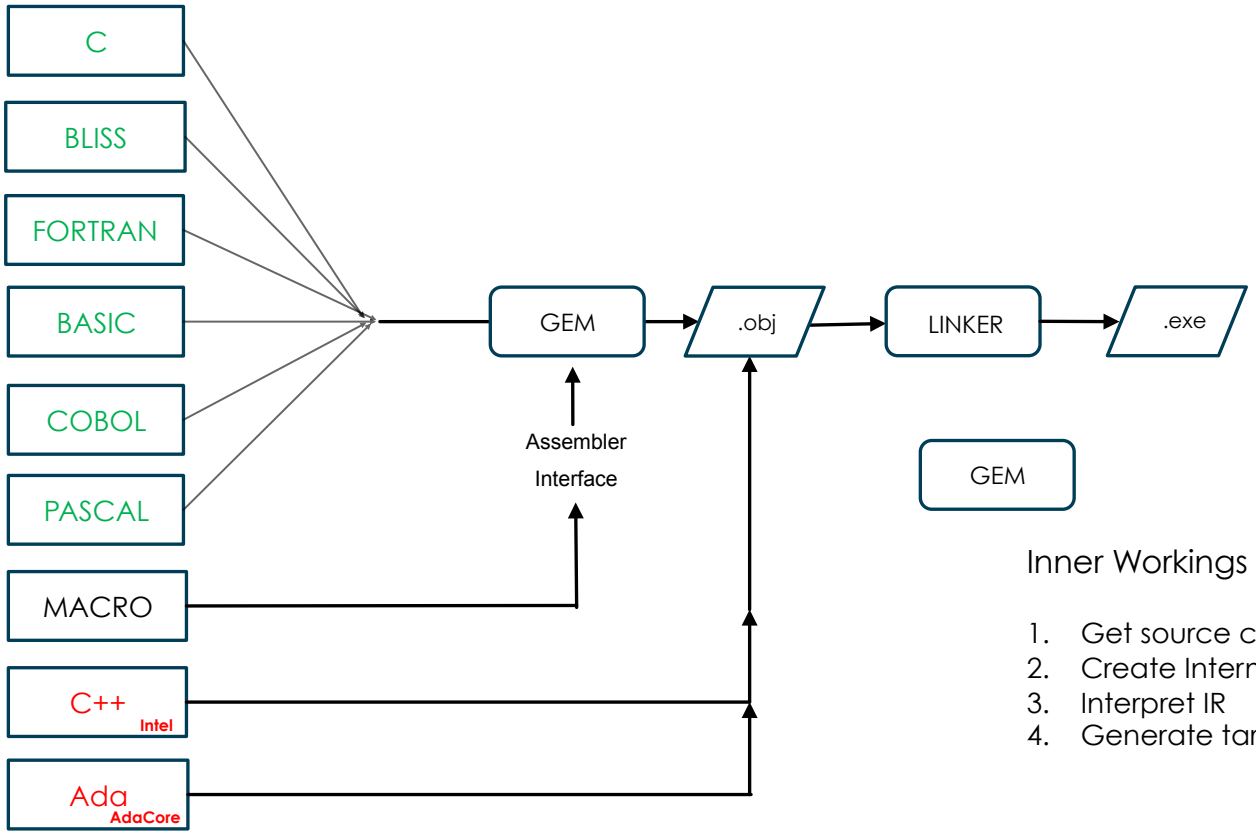
Compilers needed for port to x86-64

- BLISS, C, and Macro-32 are needed
- Choices for BLISS, C, and Macro-32 code
  - Convert code to C?
    - BLISS and Macro-32 can be very difficult to convert
    - Our C code also uses extensions so it would need work too
    - Has been studied in the past
    - Not just OpenVMS OS, but layered products too
    - Still need to provide compilers for customers and ISVs
  - Keep code and provide cross-compilers?
    - This is how the last two ports have been accomplished
    - The less code you touch, the less you break
    - A solution for BLISS, C, and Macro-32 can provide path for other compilers

What do our compilers use today?

- GEM is a multi-target, multi-platform code generator used for all Alpha compilers and most Itanium compilers.
  - BASIC, BLISS, C, COBOL, Fortran95, Pascal
  - Alpha Ada, Alpha C++
- High quality code generator and optimizer
- GEM has target-level instruction interface for the Macro-32 compiler to use

VMS Itanium Compilers and Image Building



Inner Workings of GEM

1. Get source code and command line directives
2. Create Intermediate representation (IR)
3. Interpret IR
4. Generate target object file

Which code generator to use for x86-64?

- Use and enhance GEM?
  - Only knows x86 32-bit ISA
  - Only knows older chips and instructions
  - x86 support was for Windows NT object format, calling conventions, and debug format
  - Would require several developers and would need to track future chip releases
  - Design model was for RISC chips. Would need new optimizations to use CISC addressing modes

Which code generator to use for x86-64?

- Other code generators available?
  - Intel
    - Expensive to license and use
  - gcc
    - Large user-base
    - GPL license impacts usage
    - Complicated and showing its age
  - Open64
    - Contains lots of code from many contributors
    - Great loop analysis and code quality on par with gcc
    - GPL license impacts usage
    - Poor documentation
    - Dwindling user base

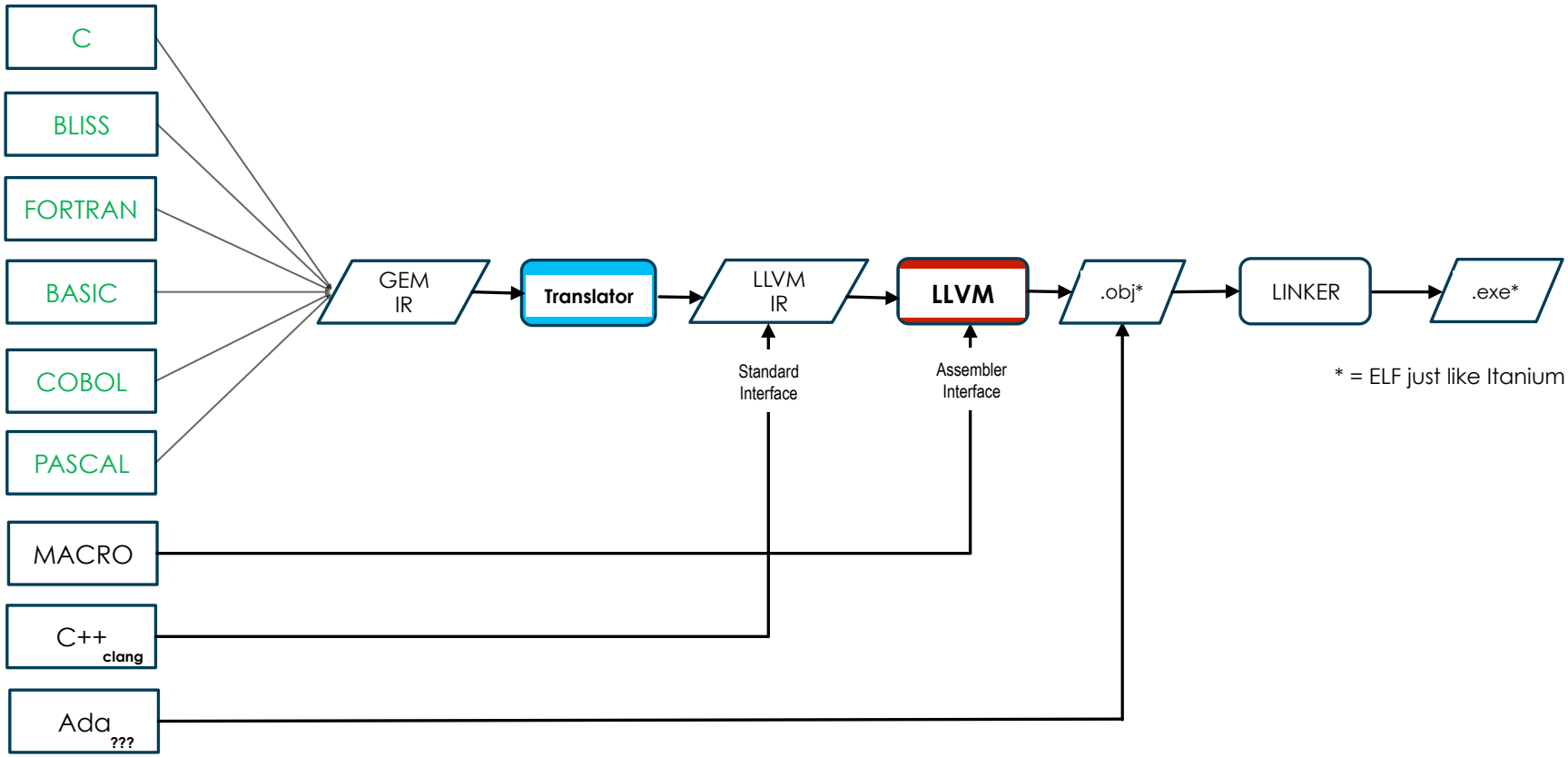
What code generator to use?

- LLVM
  - Modern
  - Multi-target
  - Excellent documentation
  - Massive user community
  - Very extensible
  - In wide-spread use by Apple, Intel, IBM, etc
  - Languages like Rust, Swift, Objective-C, Haskell
  - Great BSD-style license makes it easier to use
  - Recently surpassed gcc for code quality
  - Many libraries also included
  - Clang frontend conforms to C11, C++11, C++14
  - www.llvm.org

How to get from today to LLVM?

- Converting GEM IR to LLVM IR allows reuse of existing frontends
  - BASIC, BLISS, C, COBOL, Fortran95, Pascal
- Converter provides single place for dealing with VMSisms like dual-pointer sizes, VAX float, BASIC/COBOL datatypes, etc.
- LLVM has target-level instruction interface for the Macro-32 compiler to use
- You get all of LLVM's optimizations

Future VMS Compiler Strategy



- Continue with current GEM-based frontends
- Use open source LLVM for backend code generation
- *Create internal representation (IR) translator*
- LLVM targets x86, ARM, PowerPC, MIPS, SPARC, and more

Define the Calling Standard

- Based on the AMD ABI
  - www.x86-64.org/documentation/abi.pdf
- Add OpenVMS changes
  - Arg count (the ABI already has a form we can leverage)
  - Bound procedures (ABI already a form that is close)
  - LIB$X86 / ICB access routines
  - Translated image support
  - Condition handling / mechanism array
  - Floating point / IEEE modes
  - Exception handling / unwind information

What else is needed?

- LLVM additions
  - Add Calling Standard support as needed
    - Argument count
    - Translated image support/signatures
  - Additional DWARF support
  - Additional descriptor support for expressions
  - Exception handling enhancements
  - Machine code listing file support
  - And probably more
- Support library work
  - Compiler-rt
  - Exception handling

How does the port impact C++?

- Cannot leverage the Itanium C++ compiler
  - Intel licensed
  - Not GEM based
  - Only C++98 (and barely)
- Cannot leverage the Alpha C++ compiler
  - Old, not current, broken object model, etc

How does the port impact C++?

- Clang
  - Can compile C or C++
  - C11 and C++14 support
  - Well documented
  - However, will have to add VMSisms as needed
    - Itanium C++ already has fewer VMSisms than DEC C
    - Dual-pointer sizes and ILP models
    - DCL command line
    - #pragma warning disable
    - VAX floating
    - And others as deemed necessary
  - DEC C for legacy code; Clang for code coming to OpenVMS or for upgrading to newer language stds

How does the port impact C++ RTLs?

- Will have to merge headers and RTLs
  - C++14 headers and RTL provided by LLVM
  - The RTL on Itanium is based on RogueWave
  - Will need to merge/update headers and RTL for VMSisms
- Do we need to provide Boost also?
  - Several Boost libraries now part of C++11
  - Some headers are platform specific

How does the port impact C & RTL?

- Clang for C11 and beyond
- Enhancements to DEC C as needed
  - X86-64 intrinsics as needed
  - A real 64-bit ILP model?
  - #pragma linkage changes to match Calling Standard
- RTL & header changes
  - A real 64-bit ILP model?
  - Larger size_t? ptrdiff_t?
  - Merge in C11 features from Clang
  - RTL needs to work for both Clang and DEC C code
  - C99 changes
  - Performance fixes
  - Clean out old code
  - Chance to simplify the multitude of logicals

How does the port impact Macro-32?

- Macro has target instruction knowledge so the port is more complicated (just like from Alpha to Itanium)
- Switch to LLVM interface
- Need to define register mapping for x86-64
  - Reduced register set makes it harder for ‘recompile and go’
  - Some registers may not survive across calls (eg, store into R23 and use in called routine)
  - Getting temp registers might spill to stack (Alpha does that today)
- Additional x86_ intrinsics as needed
- Need to learn to use x86-64 addressing modes

How does the port impact other compilers?

- Fortran – still looking at more standard work
- BASIC, COBOL, Pascal
  - Additional ‘delighters’ if reasonable
  - BASIC unsigned integer for example
  - No, not object-oriented COBOL but some IBM or NonStop features to aid porting to OpenVMS perhaps
  - Better 64-bit descriptors for all perhaps
- BLISS – register linkage and global register changes to follow Macro-32 and Calling Std
- All compilers
  - DCL options for x86-64 specific items
  - New intrinsics as needed

Questions?

For more information, please contact us at:

RnD@vmssoftware.com

VMS Software, Inc. • 580 Main Street • Bolton MA 01740 • +1 978 451 0110