vms Software

Porting OpenVMS to x86-64
Update

October 16 , 2015

vms Software

Porting OpenVMS to x86-64
Update

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.

Update Topics

- Review “The Plan”
- Executable Images
- Architecture-Specific Needs
- Virtual Machines

Porting Play Book (The Plan)

Chapter 1 – Executable Images

- **Definition**: Register Mapping, Calling Standard extensions
- **Creation:** Compilers, Assembler
- **Action:** LIBRARIAN, LINKER, INSTALL, Image Activator
- **Analysis:** SDA, DEBUG/XDELTA, ANALYZE IMAGE, ANALYZE OBJECT

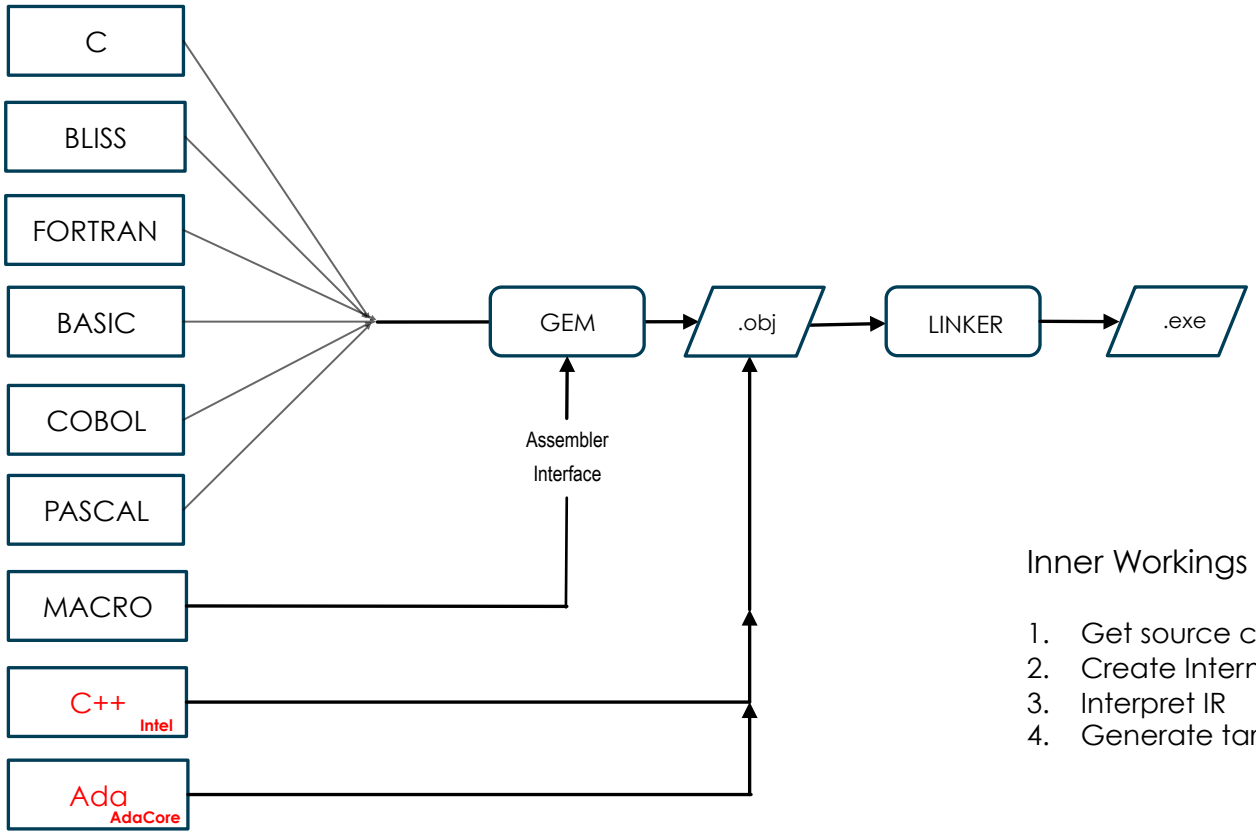
Chapter 2 – Architecture-Specific Needs (a.k.a. “The 5%”)

- Booting
- Interrupts, Exceptions
- Memory Management: protection types, access modes, address space, etc.
- Atomic Instructions
- Floating Point
- Special needs for code in assembler (e.g. VAX QUEUE instruction emulation)

Chapter 3 – Compiling and Linking Everything Else (a.k.a. “The 95%”)

- Large task but mostly mechanical
- Flush out any remaining ‘inter-routine linkage’ problems

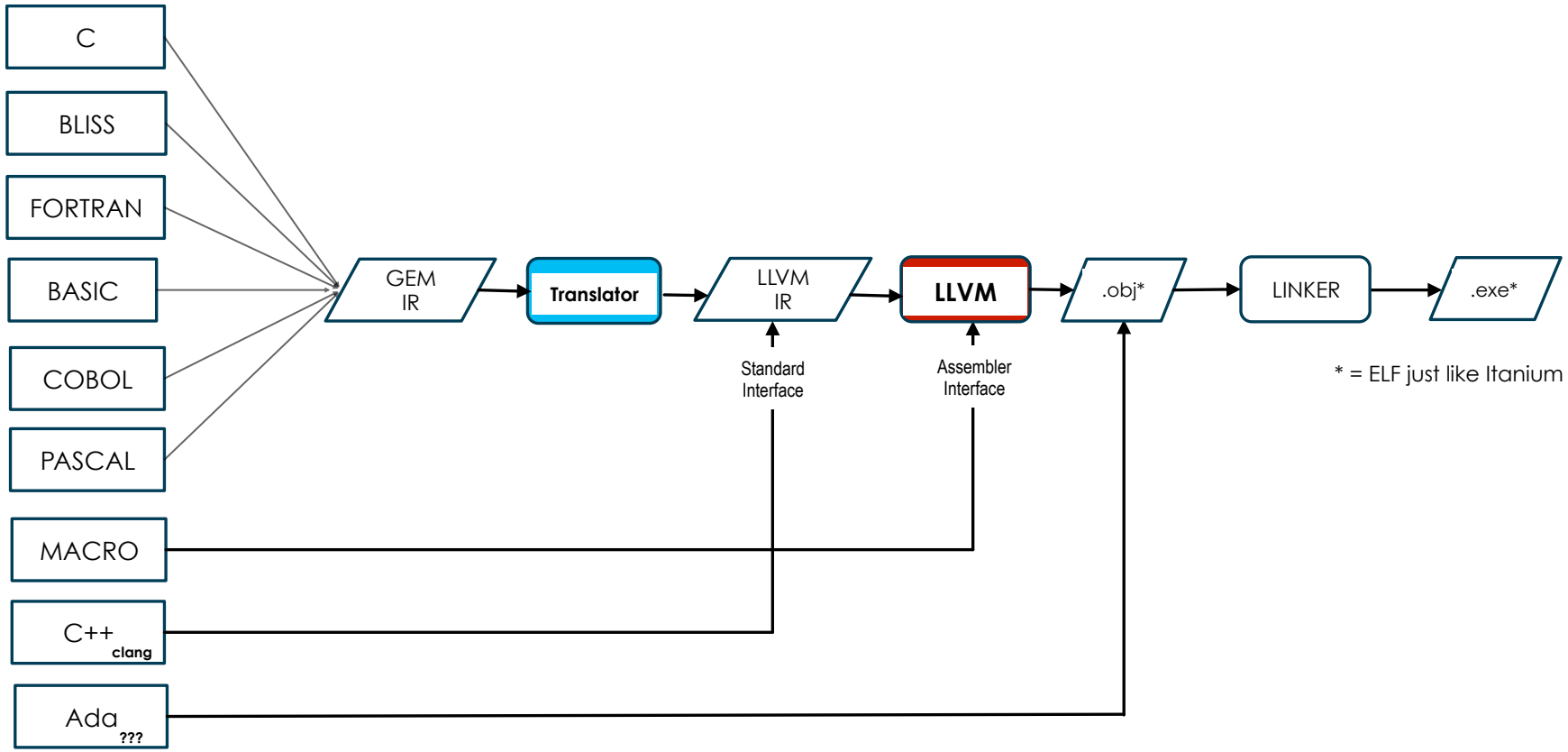
VMS Itanium Compilers and Image Building



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

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

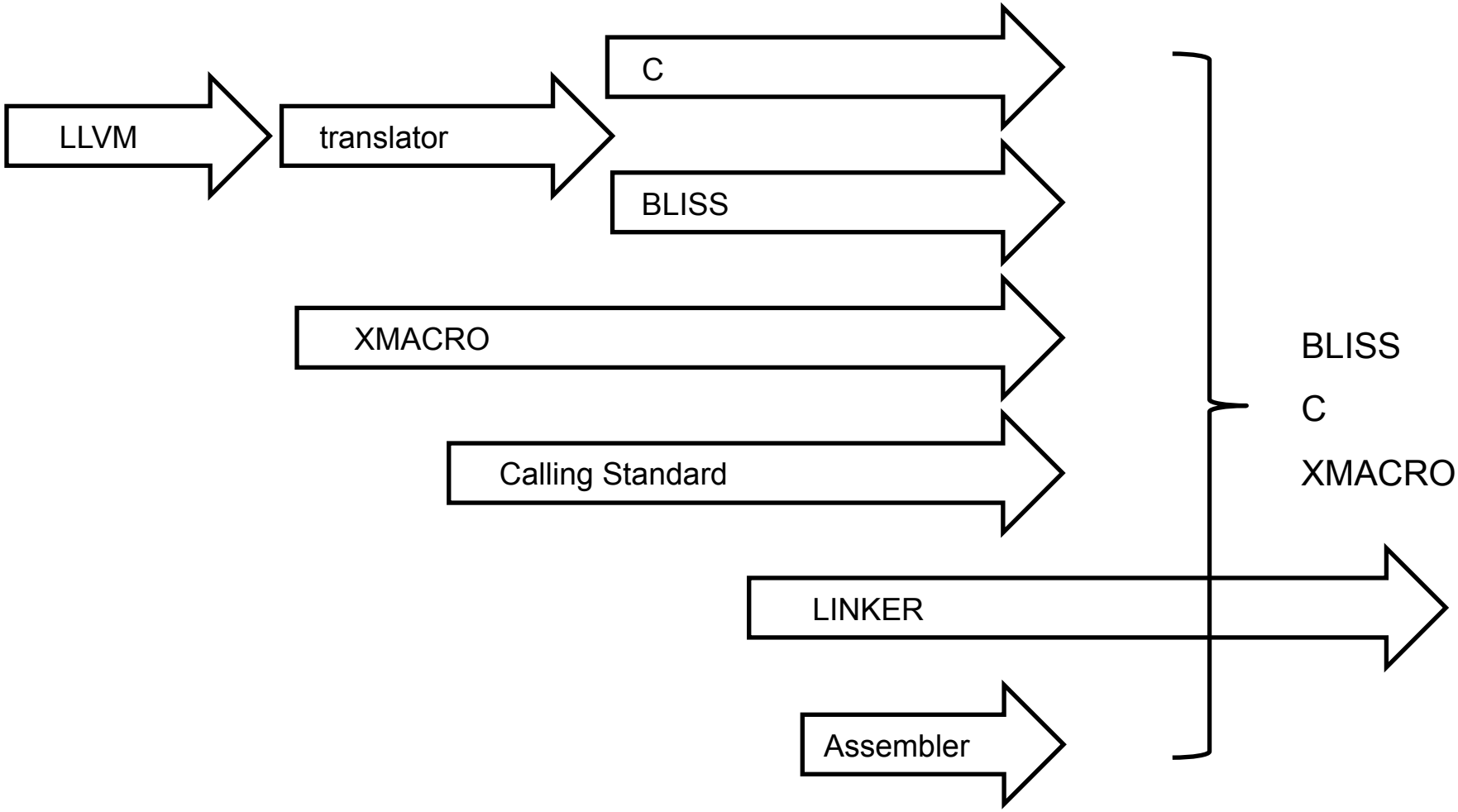
Executable Images

- Compilers
  - LLVM
    - Compiled on VMS – 99% complete, good enough for now
    - GEM IR to LLVM IR translator design mostly complete
    - Implementation of Wave 1 tuples has started
  - XMACRO is about to get started
  - Register mapping
    - Initial definition in place
    - Subject to change as XMACRO work unfolds
  - Rewriting Ada code – evaluating now
    - ACME in C
    - Security_Server in C++
- Calling Standard – work starting soon
  - Based on AMD64 Application Binary Interface
  - Like Itanium - ELF, DWARF, unwind tables

Executable Images (continued)

- LINKER – initial outline of work
  - Relocations and fixups – need thorough investigation but minimal work expected
  - Calculate virtual address space – some work
  - Pass 2 – most of the change is here
  - Writing executable/shareable images and symbol table files – small change
- LIBRARIAN - minimal work
- INSTALL, Loader, Image Activator
  - Relocations/fixups – small change
  - Share most crucial pieces of code

“First Boot” (with Cross Tools) Images



Analysis Tools: Laying the Foundation

- x86 Instruction Set Decoder – complete
  - 640 opcodes in total
  - Test ‘byte streams’ created for each instruction; developed/verified on linux
  - Used by SDA, DELTA/XDELTA, DEBUG, SCD, ANALYZE/OBJECT

- Evaluate LIB$IPF_CALLING_STANDARD routines; used by “stack walkers” and others
  - Invocation Context (current, previous)
  - Registers
  - Unwind data

Architecture-Specific Needs

- Boot Path
- Memory Management
- Use of Assembler

Boot Path

- VMS_LOADER.EFI
  - Loader is built with Visual Studio on a PC
  - Using UEFI 2.3 toolkit – Itanium and x86
    - Itanium VMS implements the original Intel EFI (circa 2002)
    - Debugging some “newness” issues on Itanium
    - Fix memory disk booting
  - Make better use of UEFI device drivers
  - Replace HP-specific interfaces to ACPI with direct ACPI calls
  - Replace use of Itanium’s SAL and PAL services
  - Create
    - memory descriptors
    - Interrupt Vector Table (IVT)
    - Machine Check frame
    - CPU enumeration list
    - x86-specific structures
  - Implement cleaner socket/core/thread initialization
- XPB (descendent of VMB ⟹ APB ⟶ IPB) – little change

Memory Management

- Initial investigation complete
  - Tasks identified and sized (S, M, L)
  - Continue diving into details
- Currently working on boot path conversion of memory descriptors and the related ACPI location info into VMS data structures
- Factoids:
  - Four levels of page tables
  - VMS will run in two hardware processor modes
  - Page protection requires page tables per mode
  - No PROBE instruction; look it up in page tables
  - Page sizes – 4KB, 2MB, 1GB

Running in Two Processor Modes

- x86 has four modes (rings) 0, 1, 2, 3.
- They do not provide the strict hierarchy of memory access protection expected by VMS.
- Example: No way to allow kernel write and prevent exec write.

- VMS will run in two hardware processor modes – privileged (0) and unprivileged (3).
- Supervisor (1) and Exec (2) memory protections will be implemented in software.
- With a small (we think) amount of change we can test two-mode operation on Itanium now.

Leveraging Previous Porting Work

- Many related architecture-specific details in one place: Software Interrupt Services (SWIS), Exception, AST Delivery…
- Conceptually architecture independent – the code is specific for each platform but it performs the same logical functions
- The largest single piece of concentrated assembler code apart from IMATHRTL
- Hides the details from the rest of VMS
  - Many aspects of the Calling Standard
  - Entering a more privileged mode
  - Interrupt handling
    - Software Interrupts
    - ASTs
    - External Interrupts
  - Saved state
  - Exception frames
  - Context switching
  - System service calling

Use of Assembler

- Evaluate Itanium assembler code
  1. Eliminate what is not needed
  2. Replace with C equivalents if possible
  3. Convert to x86 assembler
- Priority: Follow the boot path
- If better performance is needed then use assembler – difficult to predict, just let it happen and react

Virtual Machines

- Using CentOS-7 / kvm on Proliant DL380 Gen8 as development platform
- Create / delete / clone domains as needed
- Used so far for
  - Debugging parts of VMS_LOADER.EFI
  - Examining memory descriptors
  - Getting info from ACPI tables
  - Verifying transition to XPB
  - Investigating domain “tickless” timing
- Starting to work with xen and Virtual Box
- Investigating and writing design specs for
  - paravirtualized (virtio) drivers for kvm and VB
  - xen’s I/O interfaces

For more information, please contact us at:

RnD@vmssoftware.com

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