

Adventures in Porting

An ISV's Continuing Journey From AMODE 31 to AMODE 64 in Assembler

M. Ray Mullins
Phoenix Software International
10 August 2017

Session 21312

Legalese

- These techniques are presented as suggestions
- Every situation is different
- Some of these techniques may not be applicable to your situation
- YMMV
- Use at your own risk

Legalese

- z/OS, z/VSE, z/VM, z/TPF, and IBM are registered trademarks of International Business Machines, Inc.
- (E)JES is a registered trademark of Phoenix Software International
- All other trademarks are properties of their respective owners

Background

- Phoenix Software International
 - (E)JES
- We are converting as much code as possible to AMODE 64
- Why are we doing this? One reason...
 - “Phoenix is recognized as a leader when it comes to leveraging the latest hardware and software technology. The company invests countless development man hours to exploit these technologies for the sole purpose of helping our customers increase performance and reduce costs.”
 - https://www.phoenixsoftware.com/company/development_philosophy.htm

General notes

- This presentation is about converting assembler code, but also references interlanguage issues
- It is z/OS-centric
- Many of these techniques are valid for z/VSE, z/VM, and Linux on Z, and probably for z/TPF as well

Notes on my examples

- Glossary
 - HOB = high-order bit
 - LOB = low-order bit
 - ROT = rule of thumb
- I use relative branches; base-displacement branches work just as well
- I personally use SLGR for zeroing a register. You can use whatever you like: SGR, XGR, LGHI, BRCTG Rn, *, BRCTH Rn, */SRAG Rn, Rn, 32

64-bit machine instruction notes

- Majority of opcodes with a “G” are “Grande” instructions
- Target is 64 bits (whether register or storage)
- Examples
 - AGR (Add Grande Register) is a 64-bit version of AR
 - LGHI is a 64-bit version of LHI
 - STG stores 64 bits at the RXY address
- 64-bit storage opcodes are RXY or RSY
 - $Y = \pm 2^{20}$ displacement ($-524,288 \leftrightarrow +524,287$ / $X'80000' \leftrightarrow X'7FFFF'$)
 - There are 16- and 32-bit analogous Y instructions (e.g., AHY)

64-bit machine instruction notes

- GF instructions: 64-bit target, 32-bit source
 - AGF[R] – Add Grande From Fullword [Register]
- LL– prefixed instructions: Load Logical (unsigned)
 - LLGF[R]: Load Logical Grande From Fullword [Register]: 32 bits in lower half, upper half set to zeros
 - LLGT[R]: Load Logical Grande Thirty-One Bits [Register]: 31 bits in lower half, bit 32 and upper half set to zeros
- 64-bit storage opcodes are RXY or RSY
 - $Y = \pm 2^{20}$ displacement ($-524,288 \leftrightarrow +524,287$ or $X'80000' \leftrightarrow X'7FFFF'$)
 - There are 16– and 32–bit analagous Y instructions (e.g., AHY)

64-bit machine instruction notes

- LA[Y] operates on the lower half only in 24- and 31-bit AMODE
- B[AS]SM: to enter 64-bit mode, LOB must be on (odd address)
- LLGF[R]: Load Logical Grande From Fullword [Register]: 32 bits in lower half, upper half set to zeros
 - LLGT[R]: Load Logical Grande Thirty-One Bits [Register]: 31 bits in lower half, bit 32 and upper half set to zeros
- 64-bit storage opcodes are RXY or RSY
 - $Y = \pm 2^{20}$ displacement ($-524,288 \leftrightarrow +524,287$ or $X'80000' \leftrightarrow X'7FFFF'$)
 - There are 16- and 32-bit analagous Y instructions (e.g., AHY)

64-bit conversion strategy

1. Try not to change any storage areas, widen pointers, etc. unless absolutely required
2. Expand all internal stacking services to save/restore 64-bit registers
3. Implement BASSM/BSM for all inter-program linkage, calling services, etc.
4. Update abend recovery to handle programs running in AMODE 64
5. Change all XR/SR/SLR Rx, Rx in all programs to XGR/SGR/SLGR
- ~~6. Change all L in all programs to either LGF, LLGF, or LLGT as appropriate (majority were LLGF)~~
7. Clone executable macros to create 31-bit counterparts; change all programs to use clones; change back during conversion
8. Update 64-bit version of complex executable macros to work in AMODE 64
9. One by one, change outer routines to AMODE 64
10. One by one, change internal subroutines and control blocks to AMODE 64
11. **GOAL:** Implement virtual storage constraint relief by moving larger storage areas above the bar

PSI's 64-bit conversion strategy

- Small and large scope changes
- One step at a time
- Fully rebuild and regression test after steps that involve a global change (steps 1-7)
- Full regression tests after steps involving changing one program at a time (steps 8-11)

Hindsight is bliss or ignorance is 20/20

- Sometimes you find flaws and you have to go back and reassess or correct
 - 6. *Change all L in all programs to either LGF, LLGF, or LLGT as appropriate*
 - For non-address loads, LLGF was used the most, implying unsigned usage
 - This creates problems if use or test of value is arithmetic in nature, or HOB is a flag, and LTR+J (N)M/J(N)P were used previously.
 - This caused 95% of reported bugs directly attributable to conversion
 - Deeper analysis of following code required
 - In retrospect, we used LLGF too often, and should not have been a separate step, but rather be a parts of steps 8 and 9.

Steps 1, 2, 4–Addresses in control blocks

- We modified only if absolutely necessary
 - Internal stacking architecture
 - Internalabend recovery control blocks
- Internal-use-only control blocks that required changes
 - Convert fullwords to doublewords
 - Watch for usage of HOB of addresses (look for J(N)M/J(N)P instructions)
 - Watch for alignment and slack bytes, especially if addresses have ALETs (doubleword/fullword combination)
 - May require some reorganization to reduce slack bytes
 - Watch out for interlanguage requirements (especially pointer data types)

Step 1—Supporting AMODE 64 API calls

- External-facing control blocks for APIs
 - AMODE 24/31 API
 - Keep fullword addresses (don't change)
 - AMODE 64 or AMODE-agnostic API
 - Redefine fullword addresses as AD, where a 64-bit caller stores in the doubleword at the label, & a 31-bit caller stores in the fullword at the label
 - Agnostic requires testing AMODE in the entry logic
 - Recommend using at least a 144-byte save area, mapping to the proper format as required (read Chapter 2 of the *z/OS MVS Assembler Services Guide*, subtitled “Everything You Ever Wanted To Know About Save Areas, But Were Afraid To Ask”)

Step 1—Modifying APIs

- API routine handling of different AMODEs
 - Two separate modules (maintenance headache)
 - One module with separate pathing (smaller headache)
 - AMODE-specific APIs assembled from one source
 - Use a modified CSECT name as the differentiator, e.g. -4 suffix
 - SETC variables under conditional assembly (&SYSAMODE and &SYSASCE), also good for AR mode
 - Based on CSECT name, OPSYN to G or GF
 - 31-bit, L → LGF
 - 64-bit, L → LG
 - Create macros for fullword instructions to substitute grande instructions

Step 3—BASSM/BSM

- Preparatory for AMODE 64
- For staying in AMODE 31, this requires HOB to be turned on in either ACON or VCON (binder option HOBSET), or dynamically set, otherwise code swaps to AMODE 24
- Update any macro used to wrap internal subroutine calls as well
 - This is a good initiation into the required instruction changes
- Perform full regression tests
 - Lingering AMODE 24 code might be affected

Step 3—RMODE 24 diversion

- Exploit RMODE(SPLIT) if you have 24-bit code, using a glue routine

- A DCB exit example (24 to 31)

DCBABXIT RSECT ,	Address stored in DCBE
DCBABXIT AMODE 24	Control in AMODE 24
DCBABXIT RMODE 24	Staying below the line
LARL R15,ABXIT31	Point to actual exit
SAM31 ,	Flip to AMODE 31
BR R15	Transfer in AMODE 31
ABXIT31 RSECT ,	
ABXIT31 AMODE 31	Execute in AMODE 31
ABXIT31 RMODE ANY	It can be anywhere*

- The binder creates an RICON for the LARL, and CSV properly sets its value upon loading
- LARL does not set either AMODE indicator

Step 4–Recovery

- Must deal with AMODE 64 abends
- Must deal with registers with information in the high half
- Testing
 - Have a way to trigger an abend (undocumented command, for example)
 - Add code to this method to abend in AMODE 64

Step 5-Converting zeroing registers

- This should be straight forward
- A low-execution-time-impact step
- Clearing of high halves should not affect your code, depending on use
 - Unless you do your own version of the compilers' HGPR option
- Perform full regression tests!
 - Code not under your control could be affected if it does not properly manage the high halves

Step 6 ~~Changing LOAD (FULLWORD)~~

- This was done as a global scope change
- In retrospect, this was not the best idea
- Better done as part of steps 8 and 9
- Discussed later

Step 7—Create 31-bit versions of macros

- Allows you to retain macro names for AMODE 64
- Copy macros to a new name to reflect AMODE 31
 - EJESS31, for example
- Refactor source
- Build
- Deal with the ones that somehow got missed
- Build
- Test

Steps 8, 9, 10–AMODE 64 conversion

- Now the real fun begins!
- Analyze, analyze, analyze
- Work your way inward, heading towards common subroutines

First and foremost!

- AMODE 64 for the C/RSECT, binder parms & statements
 - Tells the binder to do things
 - CSV does things based on what the binder does
 - Otherwise you have to do it yourself
- SYSSTATE AMODE64=YES
 - For the macros. Think of the poor macros!

Temporary testing tip

- In entry code of outermost routine, add a LMH R0,R15,=16F'-1'
- Alternatively, for system-wide settings, use undocumented PARMLIB(DIAGxx) parameter TRAPS NAME(IeaInitRegsTask, IeaInitArSRB) (initializes both access and general-purpose-register high halves with X'FFFFFFFF' for TCBs and SRBs)
- Any subsequent AMODE 64 address reference that has not insured the purity of the high half willabend
- Arithmetic references – “unpredictable results may will occur”

Converting lower levels and calling

- Callers must use BASSM into 64-bit mode (but we did that already)
- LOB of address must be set on
- Fun things ensue if both low half HOB and LOB are on
 - You are now executing in 64-bit mode in the 2GB-4GB address range.
 - In z/OS, you are in the JRE work area with expected abends
 - In other IBM Z operating systems, you are in usable storage, although it may not be allocated

Modifying address constants

ADCONs

- Internal usage
 - $A(\text{FOO}) \rightarrow \text{AD}(\text{FOO})$
 - $31 \rightarrow 64$:
 $A(\text{FOO} + \text{X}'800000000')$ \rightarrow $\text{AD}(\text{FOO} + \text{X}'1')$
 - $24|31 \rightarrow \text{Just } A \rightarrow \text{AD}$
- External service call
 - Switch to equivalent AMODE 64 IBM services when provided

VCONs

- Almost no worries! The Binder does it all for you (if option HOBSET is on, for AMODE 31 addresses)

AMODE 64 calling AMODE 31 caveat

- If calling system or other services via macros that do not support BASSM (e.g., SVC, BASR) that cannot be invoked in AMODE 64, bracket call with SAM31/SAM64
- CALL has a LINKINST= keyword, specify LINKINST=BASSM
- CALL parameter list generation is contingent on SYSSTATE settings
 - Not 64-bit or PLIST4=YES: 4 bytes, VL honored
 - 64-bit or PLIST8=YES: 8 bytes, VL not honored
 - ASC=AR, before 2.2, ALETs always follow addresses
 - 2.2 makes a change...
 - Not 64-bit or PLIST4=YES, include ALETs unless PLISTARALETs=NO
 - 64-bit or PLIST8=YES, suppress ALETs unless PLIST8ARALETs=YES

IBM (Assembler) Services

- Only very few authorized assembler services support AMODE 64 invocation
- No DFSMS macro support
- No Callable Services for HLL support
- Some services have 64-bit equivalents (usually with a "4")
 - UNIX ® System Services especially (BPX–)

System services returned values

- IBM explicitly documents that all system services that return information return in the lower 32 bits only, unless otherwise documented (e.g. IARV64 addresses)
- Important for return codes! Use LTGFR
- One important documented exception: dynamic 24– and 31–bit storage allocation services return clean 64–bit addresses
 - There are a couple of others, like the R0 value from LOAD
- ROT: Use the same standard in internal routines as well as APIs

Loading addresses

- LLGT is usually your friend for dealing with fullword 31-bit addresses, but if the high-order-bit (HOB) could be on...
 - If it is merely an end-of-parameter list flag, LLGT
 - If it is an AMODE indicator, use LLGF, **not LGF! (Fun ensues!)**
 - If it is a flag, use LLGF, test with TMLH, then NILH X'7FFF' as soon as possible
- Sometimes LLGTR or LLGFR might be a better replacement for LR than LGR
- Context and following code are important!

Return code branch tables

- After BASSM/BSM, the return address is odd!
- Subtract 1 from the displacement in the B instruction

```
— BASR    R14,R15  
  B       4(R15,R14)  
  J       BAD  
  WTO     'It worked!'
```

becomes

```
BASSM R14,R15  
B      4-1(R15,R14)  
J      BAD  
WTO    'It worked!'
```

- Also implications for a vectored return off of R14
- *Personally, I am not a fan of either technique.*

Converting instructions

- General ROTs
 - Binary numbers, straight switch to grande, e.g., L → LG
 - Binary data where propagation of the sign bit would be an issue, use LOAD LOGICAL, e.g., SLR/ICM B '0011' → LLGH
 - Pure 31-bit address: LLGT
 - Register pair arithmetic and shifts → grande and single instructions, e.g., D → DSG(F)
 - No more games with testing condition codes after logical arithmetic

Some instruction optimizations

- Reduce number of instructions
 - Copy a register's value and immediately shift: replace with SLLG/K, which have source and target registers
- Replace storage references with immediate instructions
 - IF CH,R0,GE,=H'16' → IF CGHI R0,GE,16

- ICM[Y]
 - With B'1111'
 - If testing signed binary arithmetic value for (N)Z or (N)M, replace with LTGF
 - If testing unsigned binary arithmetic value for (N)Z, replace with LLGF/LTGR
 - If testing 31-bit address for (N)Z
 - If HOB might be on, LLGF/LTGFR
 - If HOB will never be on, LTGF
 - Unknown or don't care, LLGT/LTGFR
 - Unsigned B'0001' into a zeroed register, LLGC/LTGR
 - Signed B'1000'+SRA 24, LB/LTGR

Instructions

- ICM[Y] (continued)
 - Unsigned B'0011' into a zeroed register, LLGH/LTGR
 - B'0111' into a zeroed register
 - Don't change
 - B'1000' (MVCL padding value, sometimes X'80' into the HOB)
 - 0ILH/IILH
- IC[Y] into a zeroed register
 - LLGC

Instructions

- Add/Subtract instructions with constants for address manipulation
 - Consider converting to LA/LAY
 - Makes register usage more intuitive/self-documenting

Instructions—possible “gotchas”

- L (LOAD FULLWORD)
 - Context is important!
 - LGF, LLGF, LLGT are most frequent replacements
 - LGF if value is known to be a signed binary arithmetic value
 - LLGF if value is known to be unsigned and HOB is part of the value, or is a flag and needs to be tested first
 - LLGT if 31-bit address and HOB is irrelevant
 - Also applies to RR flavors of instructions, although you probably have already handled them

Instructions—possible “gotchas”

- Halfword arithmetic
 - IBM does not provide ?GH instructions outside of LGH/LLGH
 - If you have control of the target field, consider expansion to fullword or doubleword
 - Remember step 1 and control blocks caveat
 - Otherwise, find an unused register, L(L)GH/?GR
- LM
 - IBM does not provide LMGF
 - If all are signed binary, LMH/SRAG 32
 - If all are unsigned, LM + LMH, or LMD, using =nF'0'
 - If mixed...split up; this is also valid for the first two

Instructions—possible “gotchas”

- BCTG and its relatives
 - 64-bit unsigned value
 - Exposure to horrendous CPU loop if high half polluted
 - Generally avoid changing unless you can be absolutely sure the value will be within an acceptable range
 - HLASMTK DO SPM construct does not use it

Virtual Storage Constraint Relief-the goal!

- Storage can now be above the bar
 - Affects any control block that has pointers
- Modify control blocks first
 - a few related pointers at a time
 - Fully test
 - Lather, rinse, repeat
- Once done making pointer changes, begin changing dynamic storage allocation to 64-bit
 - Again, modify only one or related storage areas at a time
 - Test
 - Lather, rinse, repeat

Architecture level set issues

- As an ISV, we support versions of z/OS that run on older hardware, back to z8/900
- This limits (E)JES to
 - HLASM: OPTABLE(YOP)/MACHINE(ZSERIES-2)
 - z/OS XL C and METAL C, and other compilers: ARCH(5)
- Some useful instructions we can't use
 - LT instructions that reference storage ☹
 - Fullword immediate, like NILF
 - K instructions ($R_2 \rightarrow R_1$, result placed in R_3 , R_1 unchanged)
- A workaround is to create macros to generate equivalent instructions
 - Test O' and define the macro if the opcode is not defined

z/VM, z/VSE, and z/TPF

- z/VM supports AMODE 64 very nicely
- z/VSE has support
 - Code can run AMODE 64 and obtain storage above the bar using IARV64 memory objects, but...
 - z/VSE system services do not support being called in AMODE 64
 - HLASM, compilers, LNKEDT do not support AMODE 64
 - Böblingen, fangt Sie zu entwickeln an!
- z/TPF supports AMODE 64 very nicely

Interlanguage issues

- z/OS XL C/C++
 - Compiling under ILP32, all pointers are 4 bytes
 - Compiling under LP64, all pointers are 8 bytes, unless declared with `__ptr32`
 - If dealing with 8-byte pointers in a control block under ILP32, create a typedef that uses either a struct for allocating 4 bytes slack followed by a 4 byte pointer, or a union with a long long and a pointer (depending how you set up the field).
 - Watch out for pollution from the HGPR(NOPRESERVE) option, which allows the compiler to use the high halves of the registers

Interlanguage issues

- Enterprise COBOL for z/OS
 - Still only understands 31-bit pointers, even with Enterprise COBOL V6.1
 - 8-byte pointers can be split into a group level with a POINTER FILLER and the actual POINTER
 - V5+: watch out for pollution from the HGPR(NOPRESERVE) option, which allows the compiler to use the high halves of the registers

Interlanguage issues

- Enterprise PL/I for z/OS
 - Supports AMODE 64 with version 5.1 and LP(64) option
 - POINTER(64), with POINTER(32) for 4-byte pointers
 - See *Programming Guide* for advice
 - For version 4.5
 - POINTER is 4 bytes
 - If dealing with 8-byte pointers in a control block, create a TYPE that uses a STRUCTURE for allocating 4 bytes slack and a 4 byte pointer
 - Watch out for pollution from the HGPR(NOPRESERVE) option, which allows the compiler to use the high halves of the registers

Interlanguage issues

- REXX (interpreter and compiler), CLIST (ugh)
 - Still living in AMODE 31
 - STORAGE() does understand 64-bit addresses
- VS FORTRAN—Really? YES! But...
 - Only lives in AMODE 31
 - Don't be confused by POINTER*8
 - First fullword is ALET (used in EMODE)
 - Second fullword is address
- APL2, PASCAL/VS, BASIC/MVS
 - Just checking if you're still awake

Interlanguage issues

- z/OS Language Environment (including z/VM)
 - Does support AMODE 64
 - Provides AMODE 64 services
 - Well-documented
 - Some 64-bit services are being revised
- z/VSE languages, including REXX (not assembler)
 - Still AMODE 31

Wrapping up

- Changing AMODEs is not a short project, by any means
- Test, test, test!
- After completion, (almost) no more limits
- It could be a competitive advantage
- Did I mention test, test, test?

Questions?

Think of something after you leave?

- E-mail: m.ray.mullins@gmail.com
- LinkedIn: <https://www.linkedin.com/in/raymullins>
- Facebook: <https://www.facebook.com/zarchasmpgmr>
- Twitter: @zarchasmpgmr
- XING: https://www.xing.com/profile/Ray_Mullins

FIN

Thank You for Attending!
**Please remember to complete your evaluation of
this session in the SHARE mobile app.**

**An ISV's Continuing Journey From AMODE 31 to
AMODE 64 in Assembler – Session 21312**

