WHITE PAPER

Porting to 64-bit ARM

Chris Shore, ARM March 2014

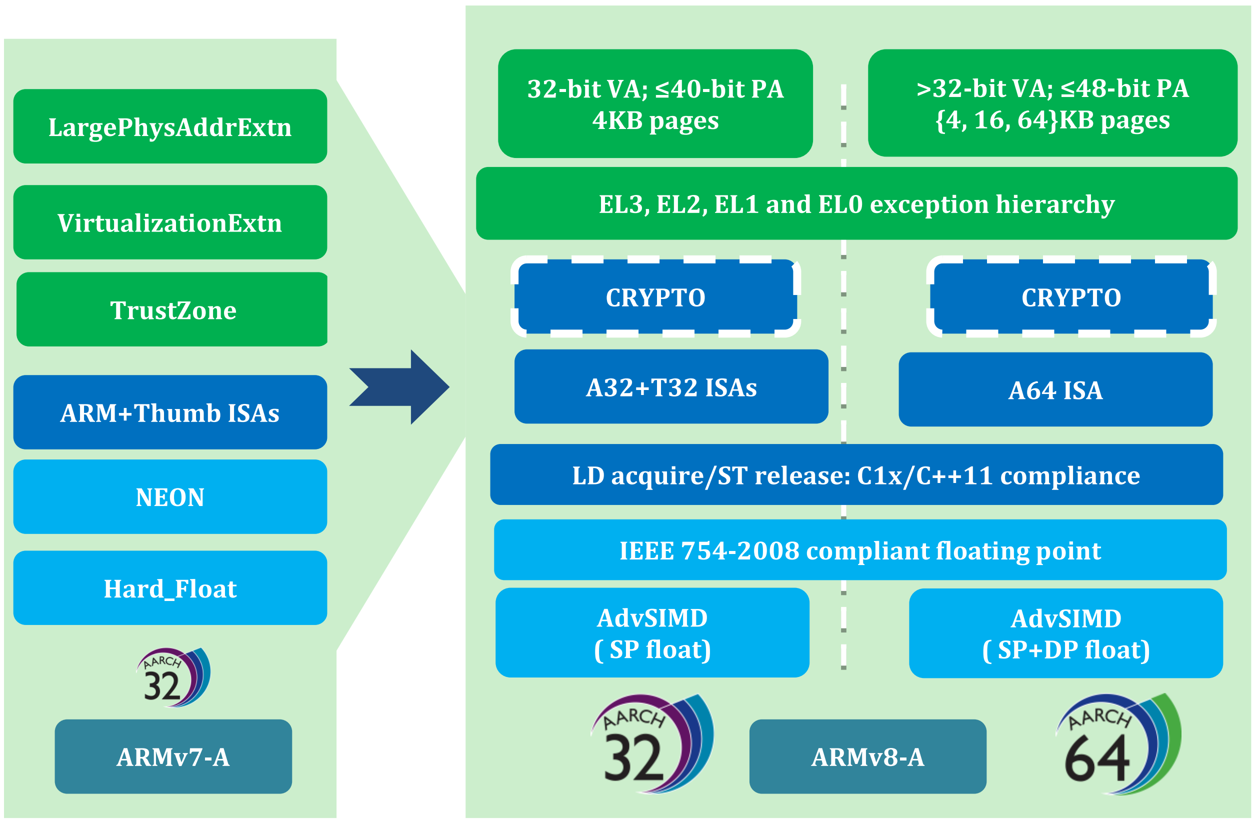
Introduction

Why 64-bit? It seems that is a question with many answers! For some, it will be the need to address more than 2GB or 4GB of memory, for others the need for wider registers and greater accuracy of 64-bit data processing, for still others the attraction of a larger register set. Either way, the 64-bit train is gathering speed and if you don't get on board it may leave you behind!

Whatever your reason for looking to move to 64-bit, it is likely that you will have a body of legacy software which will need porting as well as new code which needs writing. This paper is designed to help with both processes.

We'll start with a quick look at the evolution of the ARM architecture which has brought 64-bit to reality.

Evolution of the ARM architecture



The diagram shows how all the features present in ARMv7-A have been carried forward into ARMv8-A. But ARMv8 supports two execution states: AArch32, in which the A32 and T32 instruction sets (ARM and Thumb in ARMv7-A) are supported and AArch64, in which the new A64 instruction set is introduced.

Although backwards compatible with ARMv7-A, the exception, privilege and security model has been significantly extended and is now classified as a set of exception levels, EL0 to EL3, in a four-level hierarchy.

In AArch32, the ARMv7-A Large Physical Address Extensions are supported, providing 32-bit virtual addressing and 40-bit physical addressing. In AArch64, this is extended, again in a backward compatible way, to provide 64-bit virtual addresses and a 48-bit physical address space.

Other additions include cryptographic support at instruction level.

Overview of AArch64 in ARMv8-A

The A64 instruction set, defined in AArch64, has been designed from the ground up as a clean, modern instruction set which operates on 64-bit or 32-bit native datatypes or registers. A64 is a fixed-length instruction set in which all instructions are 32 bits in length. It does, as you might expect, have many similarities with the A32 instruction set which you'll be familiar with from earlier ARM architectures. There are some things you'll find which are new and some things which you'll go looking for and aren't there!

What you'll find...

You'll find more and larger registers. General-purpose registers are all 64-bit and can be accessed by most instructions either as 64-bit doublewords or as 32-bit words. If written as a 32-bit word, then the top half of the register is cleared.

Floating point support, single and double precision, is officially an optional part of the instruction set. However, it is expected to be included in all but a very few parts targeted at specific applications (e.g. networking) which have no need of floating point capability. Specifically, the ARM ABI for ARMv8-A does not provide for any soft floating-point linkage variant. All systems which support standard operating systems with rich application environments are expected to provide hardware support for floating point and Advanced SIMD.

System software can read the ID_AA64PFR0_EL1 to check whether these features are present on a particular system.

The Floating Point and Advanced SIMD (NEON) functionality has been carried over and the A64 support for these is familiar but a little different in places. In particular, check out the register mapping changes described below. Note that VFP and NEON are not separately optional in ARMv8-A (as they were in ARMv7-A) – a part will either include all or none of it. NEON now supports double-precision floating point and IEEE-754 arithmetic.

On the integer side, hardware divide support is included as standard.

You'll still find a load/store architecture.

...and what you won't

If you are familiar with ARMv7-A, you'll know that many instructions can be conditionally executed. In A32, this is supported via a condition field in the instruction itself; in T32, we have the IT (if-then) instruction for building conditional sequences. This isn't supported in A64 and we have a different set of specific conditional instructions. You can find examples below.

The ability to "embed" shift and rotate operations into data processing instructions is not supported in the same way in A64, although it is still possible to shift, rotate and sign-extend or zero-extend the second operand.

The Program Counter (PC) is no longer generally accessible. In particular, it can't be read or modified like other general purpose registers. There are pseudo-instructions which can be used to use it indirectly (for instance, to generate PC-relative addresses at run-time).

Historically, the ARM instruction set has included a space for «coprocessors». Originally, these were external blocks of logic which were connected to the core via a dedicated coprocessor interface. More recently, this support for external coprocessors has been dropped and the instruction set space is used for extension instructions. One specific use of it has been to provide for system configuration and control operations via the notional «coprocessor 15». You won't find anything like this in A64.

The load and store multiple instructions have been replaced with instructions which load and store pairs of 64-bit registers. These are used for stack operations as well, in place of the earlier PUSH and POP.

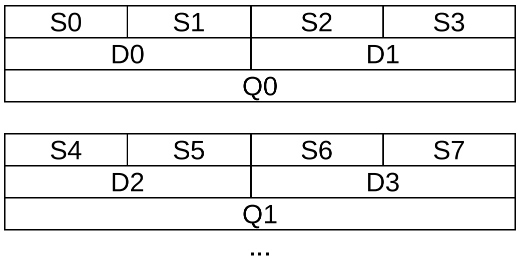
Register set

The register set offers 31 64-bit general purpose registers. All can be used either as 64-bit "Xn" registers or 32-bit "Wn" registers operating on just the lower 32 bits. There is also a dedicated zero register which adds flexibility to many operations.

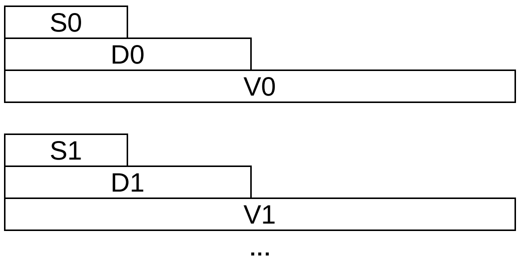
For memory access, all base pointers are now 64-bit registers allowing 64-bit virtual addressing.

The increased number and width of the registers offers some immediate and obvious advantages in greater ease of 64-bit data processing and less need to spill to the stack. The Procedure Call Standard also makes use of the greater freedom allowing up to 8 doubleword parameters to be passed in registers.

There is a separate register bank for SIMD and Floating Point. This offers 32 registers, all 128 bits wide. These can be addressed as vectors of elements ranging from bytes up to 128-bit quadword. One key difference for those used to earlier ARM architecture is the change in the way these registers map. Previously, in the NEON architecture, the mapping of word (S), doubleword (D) and quadword (Q) registers looked like this:



You can see that S0 and S1 combine to form D0, D0 and D1 combine to form Q0 and so on. For some use cases, this isn't ideal. So, in AArch64, the mapping has changed to this:



Here, S0 is the bottom half of D0, which is the bottom half of V0; S1 is the bottom half of D1, which is the bottom half of V1 and so on. This eliminates many of the problems compilers have in auto-vectorizing high level code.

Instruction set

The new A64 instruction set has a fixed 32-bit instruction length. If you are familiar with ARMv7-A A32 and T32 instruction sets you will find much which is familiar. Changes include:

- Addition of Load-Acquire (LDAR) and Store-Release (STLR) instructions which combine a load or store with a memory barrier. These simplify the implementation of critical sections.
- Optional cryptographic acceleration support at instruction level. Operating on the vector bank, these instructions provide common building block operations for efficient implementation of e.g. AES and SHA encryption algorithms.

Those two items have also been retro-fitted to the A32 instruction set in ARMv8-A.

Unlike earlier versions of the instruction set, A64 does not support conditional execution of individual instructions (like the ARM instruction set) or groups of instructions (like the Thumb instruction set). Instead, it supports a range of instructions (like CSINC – Conditional Select and Increment) whose behavior is modified by the current state of the condition code flags. Coupled with the full set of conditional branches, these make for very compact and efficient control flow.

Memory model

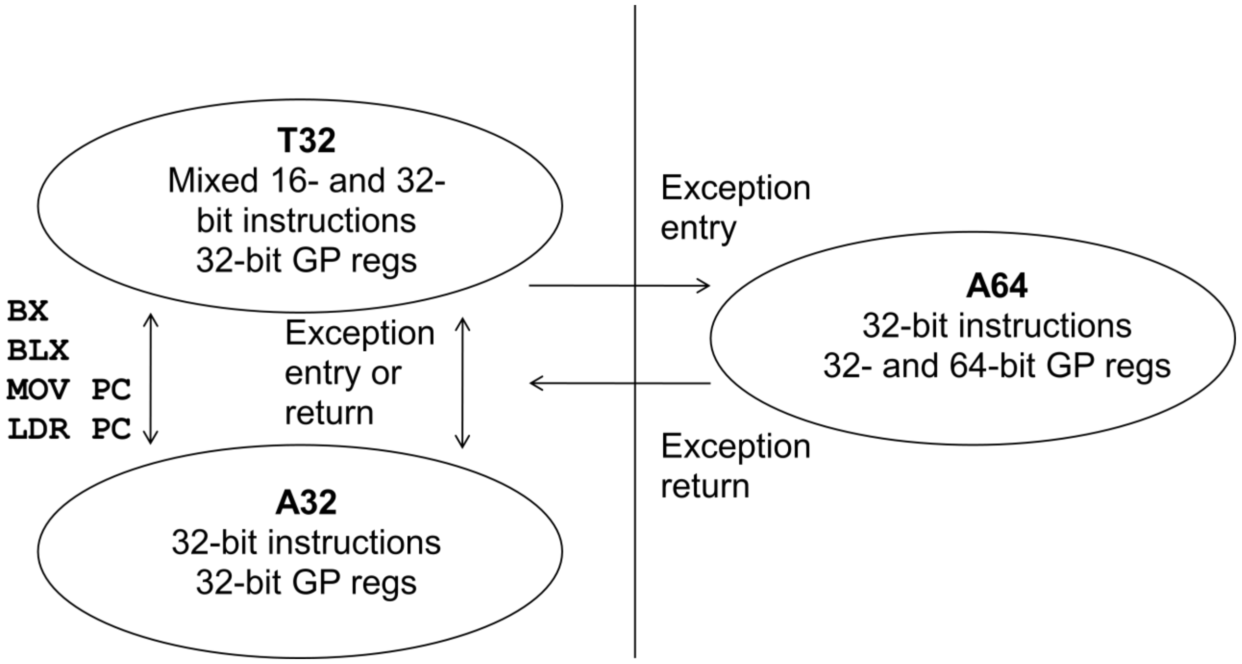
Looking at the memory architecture, again much is carried over from ARMv7-A.

- Unaligned access is supported in Normal memory for the loads and stores (with very few exceptions, such as the Load and Store Exclusive instructions for instance).
- The data and instruction interfaces are natively little-endian, although the data interface can be built for big-endian support. While individual applications cannot change their own endianness, an Operating System can host both big-endian and little-endian applications.
- There is a greater range of preload hint instructions. These allow preload for load and for store as well as providing hints about whether preloaded data should be cached or not.
- A pair of “one-way” barrier instructions, Load-Acquire (LDAR) and Store-Release (STLR), offer greater flexibility than the DMB/DSB. There is an example shown in the Assembly Code section below.
- Strongly-Ordered memory is not supported in AArch64 (it was deprecated in ARMv7-A) and Device memory has acquired some extra features to make it more flexible (the ability to define separate restrictions for Gathering, Re-ordering and Early Write Acknowledgement).

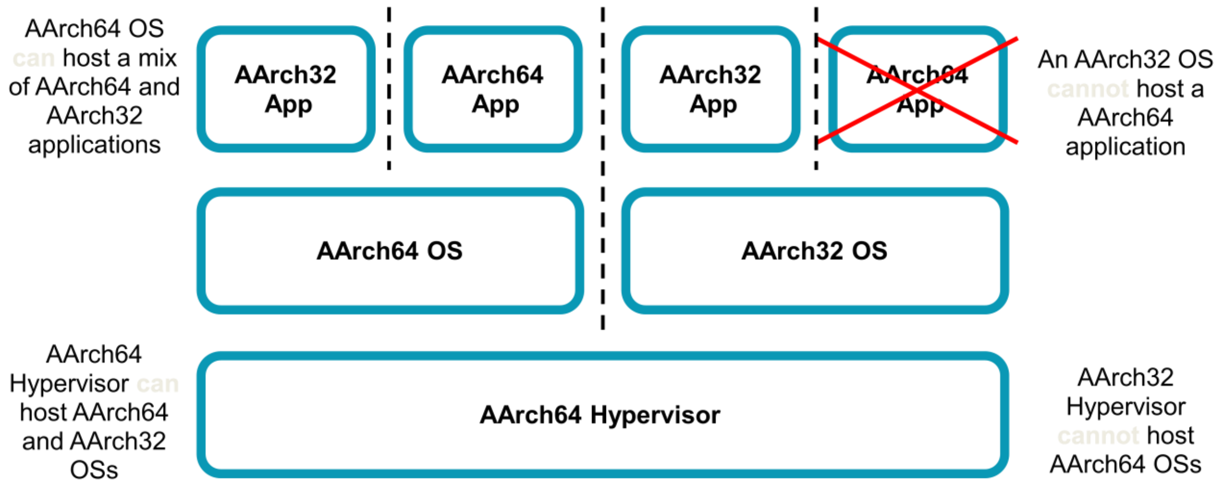
Changing Execution state and Instruction set

A fully-populated ARMv8-A processor supports both AArch32 and Aarch64 execution states. Transition between the two is always across an exception boundary. This differs from ARMv7-A in which a change of instruction set is triggered by an interworking branch (e.g. BLX).

The diagram shows the relationship between the T32, A32 and A64 instruction sets and the events which can cause a switch between them.



When taking an exception, the execution state can stay the same or go from 32-bit to 64-bit; when returning from an exception, the execution state can stay the same or go from 64-bit to 32-bit. This introduces a natural hierarchy of 64-bit and 32-bit support at each level.


AArch64 OS can host a mix of AArch64 and AArch32 applications
AArch32 App
AArch64 App
AArch32 App
AArch64 App
An AArch32 OS cannot host a AArch64 application
AArch64 OS
AArch32 OS
AArch64 Hypervisor can host AArch64 and AArch32 OSs
AArch64 Hypervisor
AArch32 Hypervisor cannot host AArch64 OSs

The AArch64 register bank is mapped in a fixed way to the AArch32 registers allowing 64-bit code access to all the user and privileged registers available to 32-bit code. This allows, for instance, a 64-bit operating system or hypervisor to perform a complete context save for a 32-bit process or guest OS.

64-bit data models

There are various standard data models in use today. They differ mainly in the size defined for integers, longs and pointers.

	ILP32	LP64	LLP64	ILP64
`char`	8	8	8	8
`short`	16	16	16	16
`int`	32	32	32	64
`long`	32	64	32	64
`long long`	64	64	64	64
`size_t`	32	64	64	64
`pointer`	32	64	64	64

64-bit Linux implementations use LP64 and this is supported by the defined A64-LP64 Procedure Call Standard A64-LP64. Other PCS variants may be defined which support LLP64 (used by 64-bit Windows) and ILP32 (a variant which can ease porting 32-bit software which doesn't need 64-bit addressing and can also improve performance of applications which use a lot of pointers but do not require 64-bit addressing).

When compiling for ARMv7-A, you will most likely have used a data model equivalent to what is shown as ILP32 in the table.

Note that the ARM compiler for ARMv8 accepts __int64 as a synonym for "long long".

Re-compile or re-write

Any port will inevitably require an element of both! The objective in most cases will be to maximize the former and minimize the latter.

The good news is that much code will simply recompile. However, you need to exercise due caution as the size of many fundamental types will have changed. Although well-written C code should not have many dependencies on the size of individual types, it is inevitable that you will come across some.

So, best practice must be to enable all warnings and errors when re-compiling and make sure you take notice of any and all warnings issues by the compiler, even if the code appears to compiler error-free.

In particular, pay very close attention to any explicit type casts in your code as these are often the source of errors when the sizes of the underlying types change.

Compiler options for ARMv8-A

In order to enable code generation or an ARMv8-A target, it is important to supply the correct options to the compiler. The following are available:

```
--cpu 8A.32
--cpu 8A.32.crypto
--cpu 8A.32.no_neon
--cpu 8A.64
--cpu 8A.64.crypto
--cpu 8A.64.no_neon
```

Some observations on these options:

- Compiling for A32 in ARMv8-A (code which will execute in AArch32 state) is very similar to compiling for ARMv7-A. Code compiled for A32 will be able to make use of a few new instructions (like Load Acquire and Store Release) and will avoid instructions which were deprecated in ARMv7-A and have now been removed in A32 (e.g. SWP).
- Compiling with the no_neon option will avoid any use of NEON/VFP instructions or registers. This might be useful for systems in which the SIMD unit will never be powered up or for particular code segments (reset code and exception handlers, for example) in which it is important to ensure that NEON/VFP is not used.

Pre-defined macros relating to ARMv8 compilation

The compiler defines a number of pre-processor macros which may be used when determining the current compilation options.

Current compilation mode

__a32__	Compiler is in A32 (ARM) mode e.g. with the command line option “--cpu=8-A.32 –arm”
__a64__	Compiler is in A64 mode e.g. with the command line option --cpu=8-A.64”
__arm__	Always defined for the ARM compiler, even when using A64 or Thumb instruction sets
__thumb__	Compiler is in Thumb state
__t32__	Compiler is in T32 (Thumb) mode e.g. with the command line option “--cpu=8-A.32 – thumb”

Available features

__ARM_NEON__	Defined when the --cpu and --fpu options indicate that NEON is available. You could use this, for instance, to conditionally include and use the NEON intrinsics.
__TARGET_ARCH_8_A	Defined when the target architecture is ARMv8-A e.g. with the command line option “--cpu=8-A.64”
__TARGET_PROFILE_A	Defined when any of the “--cpu=8-A” options are specified

Data sizes

__sizeof_int	4 as sizeof(int)
__sizeof_long	4 or 8 as sizeof(long)
__sizeof_ptr	4 or 8 as sizeof(void *)

Assembly code

Clearly assembly code will need rewriting. There is no easy solution here. However, much can be fairly simply translated. The following table shows the close match in many areas between the A32/T32 and A64 instruction sets.

A32	A64
`ADD  Rd,Rn,#7`	`ADD  Wd,Wn,#7`
`ADDS Rd,Rn,Rm,LSL #2`	`ADDS Wd,Wn,Wm,LSL #2`
`B    label`	`B    label`
`BFI  Rd,Rn,#lsb,#wid`	`BFI  Wd,Wn,#lsb,#wid`
`BL   label`	`BL   label`
`CBZ  Rn,label`	`CBZ  Wn,label`
`CLZ  Rd,Rm`	`CLZ  Wd,Wm`
`LDR  Rt,[Rn,#imm]`	`LDR  Wt,[Xn,#imm]`
`LDR  Rt,[Rn,#imm]!`	`LDR  Wt,[Xn,#imm]!`
`MOV  Rd,#imm`	`MOV  Wd,#imm`
`MUL  Rd,Rn,Rm`	`MUL  Wd,Wn,Wm`
`RBIT Rd,Rm`	`RBIT Wd,Wm`

Note that this table is not intended as an indication of direct translations! It does illustrate the similarity in syntax and semantics in many cases, though.

However, there are differences in many areas which will need rewrites. The following tables show some of these.

A32	A64
LDM/STM and PUSLH/POP instructions are replaced with LPD/STP (Load and Store Pair)	
`PUSH {r0-r1}`	`STP  x0, x1, [sp, #-8]!`
`POP {r0-r1}`	`LDP x0, x1, [sp], #8`
`LDMIA r0, {r1, r2}`	`LDP x1, x2, [x0], #8`
`STMIA r0, {r1, r2}`	`STP x1, x2, [x0], #8`

Note that the 64-bit APCS requires 128-bit stack alignment. This explains why X registers are used in the A64 examples in the table (as pushing/popping a pair of W registers would not preserve alignment).

CPSR is replaced by named fields within PSTATE e.g.

	A32	A64
CPSR is replaced with a set of separate registers and fields		

Disable IRQ	`MRS R0, CPSR`
	`ORR R0, R0, #IRQ_Bit`
	`MSR CPSR_c, R0`
		`MSR DAIFSET, #IRQ_bit`

	`CPSID  i`	

ALU Flags	`MRS R0, CPSR`
	`MSR  CPSR_f, R0`
		`MRS X0, NZCV`
		`MSR NZCV,  X0`

Set Endianness	`MRS R0,  CPSR`
	`ORR R0, R0, #E_bit`
	`MSR CPSR_c, R0`
		`SCTLR_ELn.EE controls ELn data endianness`
		`SCTLR_EL1.E0E controls EL0 data endianness`
		`MRS X0, SCTLR_EL1`
		`ORR  X0, X0, #EE_bit`
		`MSR  SCTLR_EL1, X0`

The A32 condition execution scheme allowed the following sequence to be compiled as shown in the left column of the table. In A64, it would make use of the new conditional select instructions as shown in the right column.

```
if (x == 0)
{
  y = y + 1;
}
else
{
  y = y - 1;
}
```

A32 Conditional execution	A64 Conditional operations

`CMP r0, #0`
`ADDEQ r1, r1, #1`
`SUBNE r1, r1, #1`
	`CMP w0, #0`
	`SUB w2, w1, #1`
	`CSINC w1, w2, w1, NE`

Simpler function/exception return:

	A32	A64
Dedicated function and exception return instructions		
Subroutine return	`MOV PC, LR`	`RET`
	`POP {PC}`	
	`BX LR`	

Exception return	`SUBS PC, LR, #4`
	`MOVS PC, LR`
		`ERET`

Note that it is still possible to branch in some circumstances by copying an address directly to the Program Counter. However, the RET instruction provides an explicit hint to the processor that the branch is a function return - this can improve branch prediction (using a return stack) considerably.

Floating Point and NEON

The Floating Point and NEON instructions are included in the main instruction set (rather than targeting a coprocessor as in earlier architectures). This means that these instructions set the core condition flags (NZCV) directly rather than having a separate set of status flags. This makes mixing control and data flow much easier when using the NEON/FP register bank.

The Stack Pointer

The Stack Pointer and the Zero Register are both encoded via ARM core register 31. Only a few instructions recognize this coding as the Stack Pointer (the remainder see it as the Zero Register). The following instructions can access the Stack Pointer:

- All loads and stores can use SP as the base register
- AND, ORR and EOR (with immediate and without flag setting) can use SP as the destination
- ADD/SUB with immediate can use SP as destination or first operand
- ADD/SUB extend can use SP as destination or first operand

It is worth noting that some instructions (e.g. CMP) alias to these instructions and that the alias versions can also access SP in the same way.

The table above shows PUSH and POP operations using LDP/STP instructions. The examples showed transfers involving pairs of 64-bit X registers as transferring pairs of 32-bit W registers would not preserve 128-bit alignment of the Stack Pointer. It is of course possible to PUSH/POP W registers but they need to be transferred in groups of four to preserve alignment. However, the cannot simply be transferred as two pairs with the Stack Pointer updated by 8 bytes each time as the second instruction would not then be using an aligned Stack Pointer value and this would be trapped by hardware.

An example PUSH operation for four W registers might look like this:

```
// PUSH (w0, w1, w2, w3)
STP w3, w2, [sp, #-16]! // push first pair, create space for second
STP w1, w0, [sp, #8]    // push second pair
```

Addressing Modes

A significant feature of the A64 load and store instructions is that the addressing mode is orthogonal to the register type. For instance, a load to one of the NEON/FP registers has the same addressing modes, range and capabilities as an integer load to a core register.

In addition, the NEON/FP register bank supports de-interleaving or “structured” load and store instructions similar to A32.

The Zero Register

While not necessarily of immediate use to the assembler programmer, the Zero Register simplifies the encoding of many instructions. Specifically, it always reads as zero and writes to it are ignored. This means that the CMP instruction can be encoded as a SUB with the Zero Register as the destination.

Load-Acquire (LDAR) and Store-Release (STLR)

These new instructions function as “one-way” barriers and can simplify some cases where barriers are required.

For example, in the following sequence, all accesses after the LDAR are observed after the LDAR whereas access before the LDAR are not affected.

```
LDR   ; these two accesses may be observed after the LDAR
STR
LDAR  ; “barrier” which affects subsequent accesses only
LDR   ; these accesses must be observed after LDAR
STR
```

Similarly:

```
        LDR   ; these two accesses must be observed before STLR
        STR
        STLR  ; "barrier" which affects prior accesses only
        LDR   ; these accesses may be observed before STLR
        STR
```

General language issues

The 64-bit AAPCS makes extensive use of the extra registers to make for more efficient procedure calls.

X0-X7	X8-X15	X16-X23	X24-X30
	XR (X8)	IP0 (X16)	Callee-saved (X24-28)
		IP1 (X17)	
		PR (X18)	
Parameter / result registers (X0-7) (Otherwise corruptible)	Corruptible Registers (X9-15)		
		Callee-saved (X19-23)	FP (X29) (callee-saved)
			LR (X30)

There are 8 64-bit registers are available for passing parameters and returning results. In general, a single 64-bit result is returned in X0 or a single 128-bit result is returned in X1:X0. X8 is designated as XR (the Indirect Result Location Parameter) and can be passed in by a caller to point to an area of memory for returning a structure.

Seven registers, X9 to X15, are corruptible by called functions and are therefore available as scratch space.

IP0 and IP1, the Intra-Procedure Call temporary registers, are available for use by e.g. veneers or branch islands during a procedure call. They are otherwise corruptible.

The Platform Register (PR) is reserved for specific use by particular ABIs and allocators should not use it for any other purpose. If a PR is not required on a particular platform, it may be used as a scratch register but such code will not necessarily be portable to other platforms.

X19-X28 are designated as callee-saved and must therefore be preserved by a function call.

The Frame Pointer (X29) is used for linking stack frames.

Although not available as a general purpose register, the Stack Pointer must be 16-byte aligned at any public interface. It must also be 16-byte aligned at any point where it is used to access memory. This is enforced in hardware. Note that the alignment check is on the stack pointer and not on the address which is actually accessed.

The table shows the built-in types when compiling C code for A32 and LP64.

Programming Type	Size in A32	Size in A64-LP64
`char`	8-bit	8-bit
`short`	16-bit	16-bit
`int`	32-bit	32-bit
`long`	32-bit	64-bit
`long long`	64-bit	64-bit
`float`	32-bit	32-bit
`double`	64-bit	64-bit
`size_t`	32-bit	64-bit
`void * (pointer)`	32-bit	64-bit
`intptr_t`	32-bit	64-bit
`ptrdiff_t`	32-bit	64-bit

As long as your code has used pointers in a portable way (e.g. you have never stored a pointer in an int variable and have never cast a pointer to a int or vice versa, for instance), the main thing to note is that sizeof(int) != sizeof(long).

Among other effects, this will change the size, and possibly the alignment, of structures an parameter lists. You should take care to use the int32_t and int64_t types from stdint.h in cases where storage length matters.

Note also that size_t and ssize_t are both long (64-bit) in A64-LP64.

Although it may seem that pointers may be stored in “long” variables, you should use types like intptr_t for maximum portability and safety.

Hints, tips and gotchas

Explicit and implicit type conversions

The internal promotion and type conversion in C/C++ can caused some unexpected problems when data types of different length and/or sign are missed in expressions. In particular, it is sometimes important to understand at what point conversions are made in the evaluation of an expression.

For instance:

```
int + long -> long
unsigned + signed -> unsigned
```

If the second conversion (loss of sign) is carried out before the second (promotion to long) then the result may be incorrect when assigned to a signed long.

In cases where unsigned and signed 32-bit integers are mixed in an expression and the result assigned to a signed long, one solution is to cast one of the operands to its 64-bit type. This will cause the other operands to be promoted to 64-bits and no further conversion is needed when the expression is assigned. Another solution is to cast the entire expression such that sign extension occurs on assignment.

Consider this example, in which you would expect the result -1 in a:

```
long long a;
int b;
unsigned int c;

b = -2;
c = 1;
a = b + c;
```

This will leave a as 0x00000000FFFFFFFF (4294967295 in decimal|) and is clearly an unexpected and incorrect result. This is because the result of the addition is converted to unsigned before it is converted to long long.

```
long long a;
int b;
unsigned int c;

b = -2;
c = 1;
n = (long) b + c;
```

This gives a result in a of 0xFFFFFFFFFFFFFFFF (-1 in 2's complement signed notation) and is the expected result. The calculation is now all carried out in 64 bit arithmetic and the conversion to signed now gives the correct result.

Bit manipulation operations

Be careful that bitmasks are the correct width. There is the possibility that implicit type conversions in C expressions can have some unexpected effects. Consider the following function or setting a specified bit in a 64-bit variable:

```
long SetBitN(long value, unsigned bitNum)
{
  long mask;

  mask = 1 << bitNum;

  return value | mask;
}
```

This function will work fine in a 32-bit environment and allows bits to be set in positions 0 thru 31. In order to port it to a 64-bit system, you might think it sufficient simply to change the type of the mask to allow bits to be set in positions 0 thru 63.

```
long long SetBitN(long long value, unsigned bitNum)
{
  long long mask;

  mask = 1 << bitNum;

  return value | mask;
}
```

This doesn't work correctly as the numeric literal '1' has int type. The exact behavior depends on the configuration and assumptions of the individual compiler.

To make the code function correctly, you need to give the constant the same type as the mask:

```
long long SetBitN(long long value, unsigned bitNum)
{
  long long mask

  mask = (long long) 1 << bitNum;

  return value | mask;
}
```

You should also take care to specify the type for literal values, e.g.:

```
1L    // (long)
1LL   // (long long)
1U    // (unsigned)
1UL   // (unsigned long)
1ULL  // (unsigned long long)
```

This will also go some way to avoiding this type of problem.

Magic numbers

All code includes constants of some description. However, beware of constants which make assumptions about the size of basic types e.g.

```
#define BYTES_IN_WORD 4
```

Indexes

When using large arrays or objects in a 64-bit environment, be aware that an int may no longer be large enough to index all entries. In particular be careful when using iterating over an array using an int index.

```
size_t Count = BIG_NUMBER;
for (unsigned int index = 0; index != Count; index++) ...
```

Since size_t is a 64-bit type and unsigned int is a 32-bit type, it is possible to define the size of the object such that the loop will never terminate.

Pointers and ints

Given the following:

```
int i, *p, *q;

p = &i;
q = (int *) (int) &i;
```

In A32, p == q. In A64-LP64, p != q.

To guard against this, you should use intptr_t (from stdint.h) for pointer types.

In pointer arithmetic, variables which are added or subtracted from pointers should be declared as ptrdiff_t as an int (as well as being signed) may not be large enough to hold the difference between two pointers.

Structure padding

Changes in the sizes of individual elements and in their respective alignment requirements will change the size of many structures.

```
struct foo
{
  int a;
  long l;
  int x;
}
```

In ILP32, this structure has size 12 (bytes) and there is no padding between the elements.

In LP64, it has size 20. The ”long” has increased from 4 bytes and 8 bytes and must now be double-word aligned. This introduces four bytes of padding between the end of the first ”int” and the ”long”.

References

1. ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile
   ARM DDI 0487
2. ARMv8 Instruction Set Architecture
   ARM GENC 010197
3. Procedure Call Standard for the ARM 64-bit Architecture (AArch64)
   ARM IHI 0055
4. ARM Compiler for ARMv8 – Introducing the ARM Compiler Toolchain
   ARM DUI 0633
5. ARM Compiler for ARMv8 – Using the Compiler
   ARM DUI 0621
6. ARM Compiler for ARMv8 - Compiler Reference
   ARM DUI 0628