Migrating code from ARM to ARM64

Kévin Petit <kevin.petit@arm.com>

LPC14 - 17/10/2014
Outline

Writing portable code
  Best practises
  Type promotion
  How do I migrate?

ARM vs. ARM64
  A few definitions
  Comparing AArch32 and AArch64
  Return instruction
  Stack pointer and zero register
  No load multiple, only pairs
  LDAR / STLR
  Conditional execution example
  NEON
  Legacy instructions

References
Outline

Writing portable code
  Best practises
  Type promotion
  How do I migrate?

ARM vs. ARM64

References
Writing portable code

Best practices

- No assumptions about the type sizes
- Magic numbers
- `size_t` and `ssize_t`
- `printf` formats (%zu, %zd, etc)
- Beware of shifts
- Structure padding / alignment
- And of course: `sizeof(int) != sizeof(void*)`
Writing portable code

Type promotion

- C/C++ have internal promotion rules (size and/or sign)
  - 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.
- Complicated, even experienced programmers get caught
- Understand the order
Consider this example, in which you would expect the result -1 in a:

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

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

- 32-bit: \( a = 0xFFFFFFFF \) (-1)
- 64-bit: \( a = 0x00000000FFFFFFFF \) \( (2^{32} - 1) \)

Not what you expect
The result of the addition is converted to unsigned before it is converted to long

Solution: cast to 64-bit before the cast to unsigned
Writing portable code

Type promotion

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

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

- **32-bit**: $a = 0xFFFFFFFF (-1)$
- **64-bit**: $a = 0xFFFFFFFFFFFFFFFF (-1)$

- Calculation is now all carried out in 64-bit arithmetic
- The conversion to signed now gives the correct result
Writing portable code

How do I migrate?

A mix of:

- Recompile
- Rewrite
  - Better use of 64-bit
  - An opportunity to clean the code
Outline

Writing portable code

ARM vs. ARM64
  A few definitions
  Comparing AArch32 and AArch64
  Return instruction
  Stack pointer and zero register
  No load multiple, only pairs
  LDAR / STLR
  Conditional execution example
  NEON
  Legacy instructions

References
ARM vs. ARM64

A few definitions

ARMv8-A architecture:
- AArch64 is its 64-bit execution state
  - New A64 instruction set
- AArch32 is its 32-bit execution state
  - Superset of ARMv7-A
  - Compatible
  - Can run ARM®, Thumb® code
ARM vs. ARM64
Comparing AArch32 and AArch64

- Presenting only userspace
- See Rodolph Perfetta’s "Introduction to A64" presentation
ARM vs. ARM64

Return instruction

PC not an accessible register anymore

AArch32

MOV PC, LR
or
POP {PC}
or
BX LR

AArch64

RET
ARM vs. ARM64

Stack pointer and zero register

- Register no. 31
- Zero register
  - xzr or wzr
  - Reads as zero
  - A way to ignore results
- Stack pointer
  - 16-byte aligned (configurable but Linux does it this way)
  - No multiple loads
  - Only a few instructions will see x31 as the SP
ARM vs. ARM64

No load multiple, only pairs

**AArch32**

```assembly
PUSH {r0, r1, r2, r3}
```

**AArch64**

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

Keep SP 16-byte aligned
ARM vs. ARM64

LDAR / STLR

### AArch32

- LDR
- STR
- DMB
- LDR
- STR

### AArch64

- LDR ; these two accesses may be observed after the LDAR
- STR
- LDAR ; ""barrier which affects subsequent accesses only"
- STR ; this access must be observed after LDAR

Similarly:

- LDR ; this access must be observed before STLR
- STLR ; ""barrier which affects prior accesses only"
- LDR ; these accesses may be observed before STLR
- STR
ARM vs. ARM64

Conditional execution example

C

```c
int gcd(int a, int b) {
    while (a != b) {
        if (a < b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}
```

AArch32/T32

```
gcd:
    CMP    r0, r1
    ITE
    SUBGT  r0, r0, r1
    SUBLT  r1, r1, r0
    BNE    gcd
    BX     lr
```

AArch64

```
gcd:
    SUBS   w2, w0, w1
    CSEL   w0, w2, w0, gt
    CSNEG  w1, w1, w2, gt
    BNE    gcd
    RET
```
ARM vs. ARM64

Conditional execution example

So, how do I migrate that?

Short answer:
- You’re on your own, be clever

More interesting answer:
- Don’t attempt direct translation
  - Won’t work in a majority of cases
  - Even if it does, it is usually a bad idea
- Opportunity for new optimisations
ARM vs. ARM64

NEON

- Part of the main instruction set / no longer optional
- Set the core condition flags (NZCV) rather than their own
  - Easier to mix control and data flow with NEON

**AArch32**

\[
\text{vadd.u16 d0, d1, d2}
\]

**AArch64**

\[
\text{add v0.4h, v1.4h, v2.4h}
\]
ARM vs. ARM64

NEON

AArch32: 16 x 128-bit registers

AArch64: 32 x 128-bit registers
ARM vs. ARM64

NEON

- Intrinsics are mostly compatible
- Assembly will need a translation and/or rewrite
  - Scripts
  - Register aliasing can *will* get in the way
ARM vs. ARM64

Legacy instructions

- SWP and SWPB
- SETEND
- CP15 barriers
- IT, partially
- VFP short vectors
- ...
- Emulated, slow, (possibly broken for some cases)

If you can avoid them, please do.
Outline

Writing portable code

ARM vs. ARM64

References
References

- Porting to ARM 64-bit, Chris Shore
- ARM C Language Extensions
- ARM Architecture Reference Manuel
- ARMv8 Instruction Set overview
Thank You

The trademarks featured in this presentation are registered and/or unregistered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.