Migrating to 64-bit on ARMv8-A

The Architecture for the Digital World®
ARM®

Agenda

- C code considerations

Porting ARMv7-A NEON/VFP code to AArch64

Assembly code porting

ARM®

C code consideration overview

- **Most of the work is done by the compiler**
- **Porting to AArch64 is similar as for any other 64-bit platform**
- **But some caution must be taken**
  - Some 32-bit code might not work correctly in a 64-bit environment
  - Must be careful of type conversion
  - Pointers need attention as all pointers are 64-bit
  - Need to be aware of the differences between APCS32 and APCS64
- **Some examples in following section**

ARM®

Data model

- **Data model and basic data type size**

	ILP32	LP64 (linux)	LLP64 (Windows)
char	8	8	8
short	16	16	16
int	32	32	32
long	32	64	32
long long	64	64	64
pointer	32	64	64
size_t	32	64	64
float	32	32	32
double	64	64	64

ARM

C general type conversion rules

- **C general type conversion includes**
  - Explicit conversion
    - Type casting
  - Implicit conversion
    - Integral promotion
    - “Usual arithmetic conversions”

- **C general type conversion is important when migrating C/C++ code from a 32-bit platform to a 64-bit platform**
    - Being aware of the rules helps to avoid some potential problems
    - Examples will be provided in later slides.

ARM®

C general type conversion rules

- **Integral promotion**

  “A **character**, a **short** integer, or an **integer bit-field**, signed or unsigned, or an object of **enumeration** type, may be used in an expression wherever an **integer** maybe used.“

  “If an **int** type can represent all values of the original type, the value is converted to an **int**, otherwise it is converted to an **unsigned int.”**

  ----K&R, C Programming Language

Source Type	Result Type
unsigned char	int
char	int
short	int
unsigned short	int
unsigned int: 16	int
unsigned int: 32	unsigned int

ARM®

C general type conversion rules

- **Integral promotion examples**

```
unsigned char va= 0x55;

if (~va == 0xAA)
   return 1;
else
   return 0;
```

va has a small type so its value will be promoted to a **signed int .**

The result of the complement will then be 0xFFFFFFAA, not 0xAA on 32-bit processors.

The two values are not equal.

```
unsigned char i=0, j=10;
if ((i - j) < 15)
   i++;
```

i and j values are promoted to **signed int**

The subtraction result is the signed value -10

The result is less than 15

```
char a = 1;
char b = 16;
int c;
c = a << b;
```

a and b are converted to an integer.

The promoted type of the left operand is int,

so the type of the result is an int. The integer representation of a is left-shifted 16 times.

ARM®

C general type conversion rules

- **Usual Arithmetic Conversions for integers**
  - When performing an arithmetic operation, C transforms both operands into a common type, which is used for the actual operation and as the type of the result

    - The usual arithmetic conversions are performed implicitly for the following operators:
      - Arithmetic operators with two operands: *, /, %, +, and -
      - Relational and equality operators: <, <=, >, >=, ==, and !=
      - The bitwise operators, &, |, and ^
      - The conditional operator, ?: (for the second and third operands)

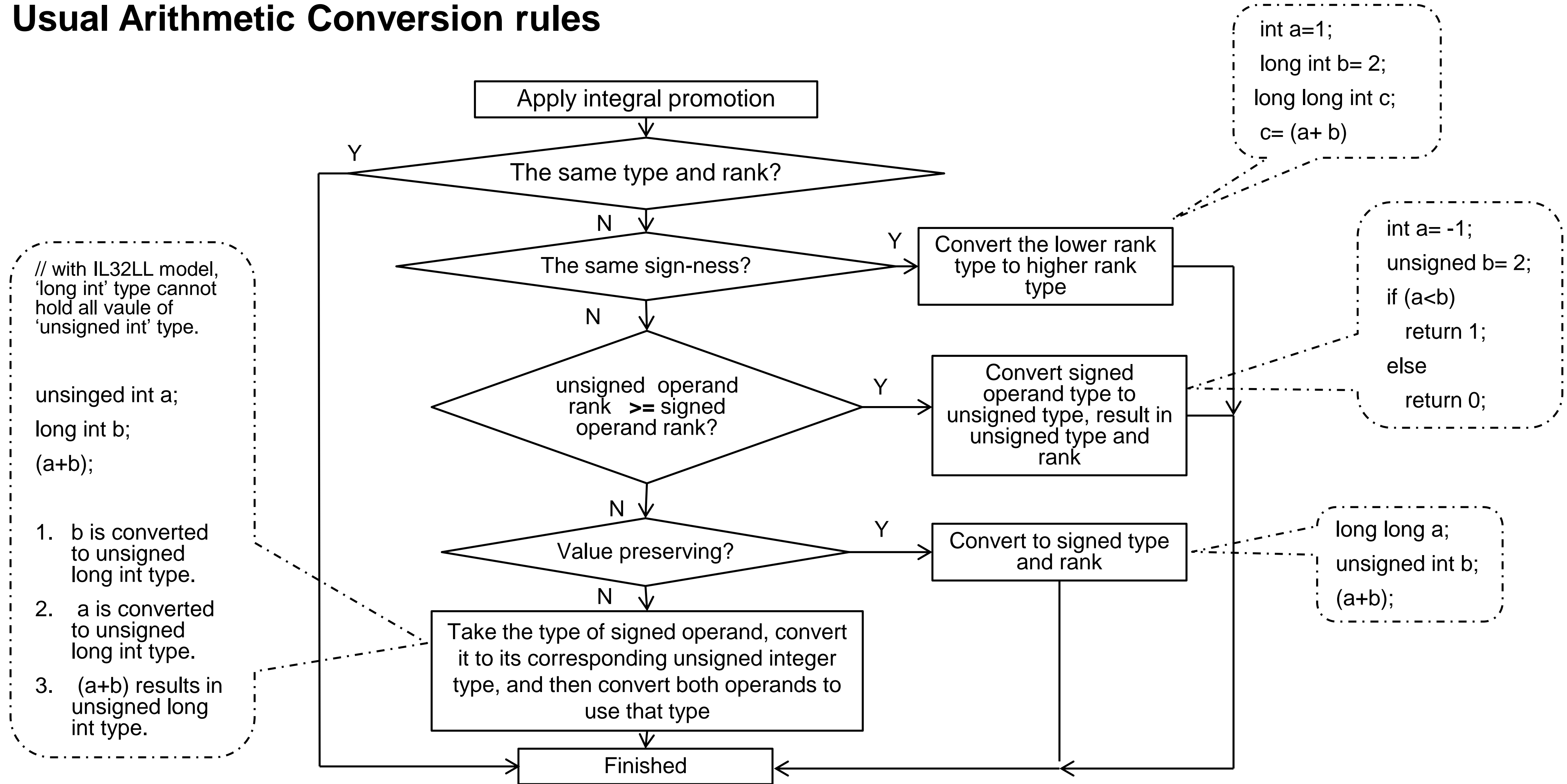
  - Before Usual Arithmetic Conversions, Integer Promotion is applied

  - Integer conversion rank
    - (signed/unsigned) char
< (signed/unsigned) short
< (signed/unsigned) int
< (signed/unsigned) long
< (signed/unsigned) long long

ARM®

C general type conversion rules

- **Usual Arithmetic Conversion rules**



ARM®

When sign and size combine…

- **Sometimes both sign and size are subject to implicit conversions**

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

  - Here the order of the conversions is important…

    …when a negative signed integer is promoted to an unsigned integer of the same or larger type, it is first promoted to the signed equivalent of the larger type, then converted to the unsigned value

```
int a = -1;
unsigned long long b = 2;
unsigned long long c;
c = a + b;
```

1. a is converted to (signed long long):
   -1 (0xFFFF:FFFF) to -1LL(0xFFFF:FFFF:FFFF:FFFF)
2. Then a is converted to (unsigned long long):
   0xFFFF:FFFF:FFFF:FFFF
3. (a + b) = (0x0000:0000:0000:0002 + 0xFFFF:FFFF:FFFF:FFFF)

ARM®

When sign and size combine…

- **It doesn't always work…!**

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

b = -2;
c = 1;
a = b + c; //a = 0x0000:0000:FFFF:FFFF
```

1. b is converted to (unsigned int)
2. (b + c) = 0xFFFF:FFFE + 1 = 0xFFFF:FFFF
3. result is converted to signed long (0xFFFF:FFFF to 0x0000:0000:FFFF:FFFF)

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

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

- **Integer constants**
  - Integer constants (unless modified by a suffix, e.g. 0x8L) are treated as the smallest size that can hold the value
  - Numbers written in hexadecimal may be treated by the compiler as **int**, **long**, or **long long** types, and may be either **signed** or **unsigned** types
  - Decimal numbers are always treated as **signed** types

ARM®

Avoid casting pointers to integers

- **Pointer is now 64-bit on 64-bit platform (was 32-bit on 32-bit platform)**
  - Type casting pointer type to integer type is not a good idea
- **Avoid casting pointers to integers**

0x0000_0008_0000_0000

```
short * test(short *ptr)
{
  int n = (int) ptr + 2;
  ptr = (short *) n;
  return ptr;
}
```

Compiler warning: conversion from pointer to smaller integer

```
ADD     w0, w0, #2
SXTW    x0, w0
RET
```

0x0000_0000_0000_0002

- **Another example**
  - Align an address with page size

```
char *p;
p = (char *) ((int) p & PAGEOFFSET);
```

```
char *p;
p = (char *) ((uintptr_t) p & PAGEOFFSET);
```


ARM®

Avoid casting pointers to integers

- **Solution: Using `ptrdiff_t` type**
  - ptrdiff_t is a C/C++ base signed integer type
  - Its size is chosen so that it will be large enough to hold the difference between any two pointers on a given platform
    - 32-bit on a 32-bit system and 64-bit on a 64-bit system
  - ptrdiff_t can safely store a pointer, it is usually used in loop counters, to index arrays, to store sizes and in address arithmetic

0x0000_0008_0000_0000

```
short * test(short *ptr)
{
  ptrdiff_t n = (ptrdiff_t) ptr + 2;
  ptr = (short *) n;
  return ptr;
}
```

0x0000_0008_0000_0002

```
ADD       x0, x0, #2
RET
```

- **Use #include <stdint.h> and the types defined there**
  - uint64_t, uintptr_t etc.

ARM®

Avoid casting pointers to integers

- **Another example**
  - Align an address with page size

```
char *p;
p = (char *) ((int) p & PAGEOFFSET);
```

```
char *p;
p = (char *) ((uintptr_t) p & PAGEOFFSET);
```


ARM®

Pointer arithmetic

- **Be careful with data type conversion rules when doing pointer arithmetic operations**

```
int   a = -3;
unsigned int b = 2;
int c[4] = {0,1,2,3};

int *p = c + 3;
p = p + (a + b);
```

p=?

```
ADD    x0,x0,#0xc

ADD    w1,w2,w1

ADD    x0,x0,w1,UXTW #2
```

Expresssion	Data Type	Hex value
a	Signed int	0xFFFF:FFFD
b	Unsigned int	0x0000:0002
c	Pointer	0x0000:0000:8000:0000
p = c + 3	Pointer	0x0000:0000:8000:000C
a + b	Unsigned int	0xFFFF:FFFD + 0x000:00002 =0xFFFF:FFFF
p + (a + b)	Pointer	0x0000:0000:8000:000C +0x0000:0000:FFFF:FFFF * 4 =0x0000:0004:8000:0008

- With ILP32 data model, p will be 0x80000008

ARM®

Bit shift

- **Integer constants (unless modified by a suffix, such as 0x8L) are treated as the smallest size that will hold the value**
  - Setting a bit in a 64-bit mask

```
long long SetBitN(long long value, unsigned bitNum)
{
  long long mask = 1 << bitNum;
  return value | mask;
}

long long a = 0x0;

int main ()
{
  a = SetBitN(a, 32);
  return 0;
}
```

```
0x0000:0000:0000:0001
```

```
0x0000:0000:0000:0000:0000:0000:0000:0000
```

```
SetBitN
   MOV     w2,#1
   LSL     w1,w2,w1
   SXTW    x1,w1
   ORR     x0,x0,x1
   RET
```


ARM®

Bit shift

- **Solution: Specify the constant with a suffix**

```
long long SetBitN(long long value, unsigned bitNum)
{
  long long mask = 1LL << bitNum;
  return value | mask;
}

ptrdiff_t a = 0x0;

int main ()

{
 a = SetBitN(a, 32);
 return 0;

}
```

```
0x0000:0000:0000:0000:0000:0000:0000:0001
```

```
0x0000:0000:0000:0001:0000:0000:0000:0000
```

```
SetBitN
  MOV   x2,#1
  LSL   x1,x2,x1
  ORR   x0,x0,x1
  RET
```

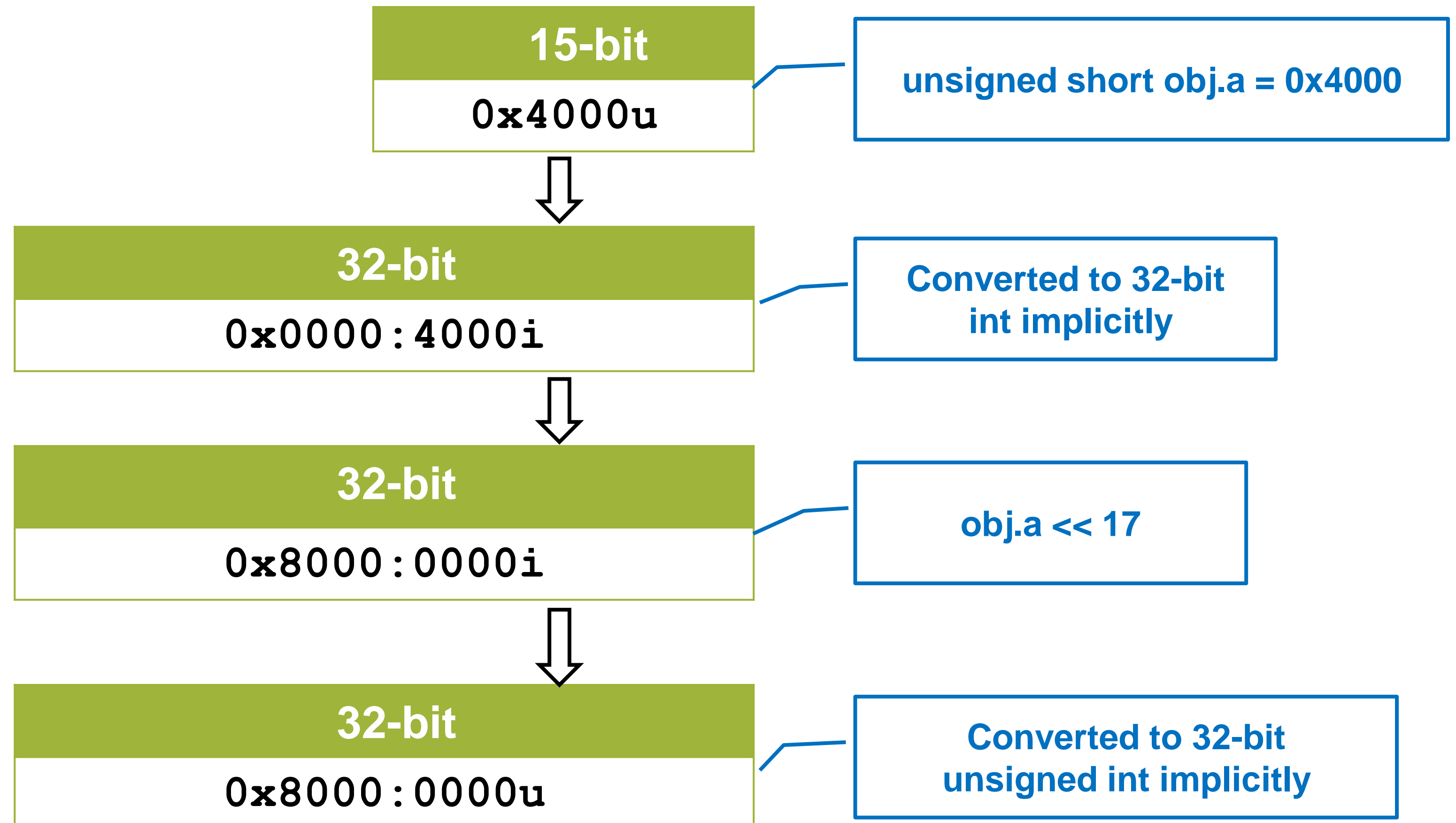
Shift from 0 to 63 bit in X1 register bottom 6 bits

ARM®

Bit field operations and sign extension

- **On 32-bit system (ILP32)**

```
struct BitFieldStruct
{
  unsigned short a:15;
  unsigned short b:13;
};

struct BitFieldStruct obj;

unsigned long x;

int main()
{


  obj.a = 0x4000;

  x = obj.a << 17;

  return 0;
}
```

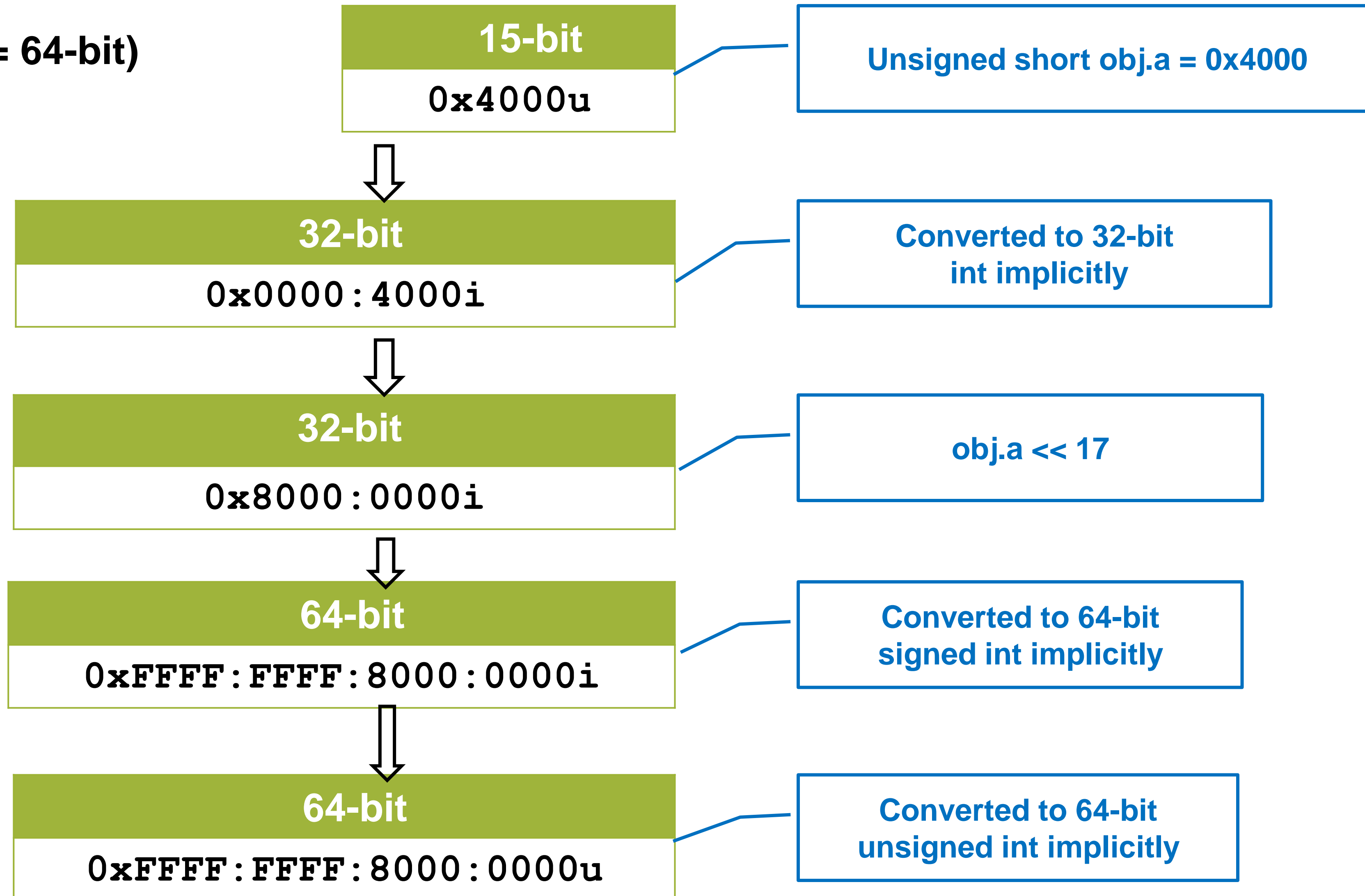


ARM®

Bit field operations and sign extension

- **On 64-bit system(LP64, long = 64-bit)**

```
struct BitFieldStruct
{
  unsigned short a:15;
  unsigned short b:13;
};

struct BitFieldStruct obj;

unsigned long x;

int main()
{


  obj.a = 0x4000;

  x = obj.a << 17;

  return 0;
}
```

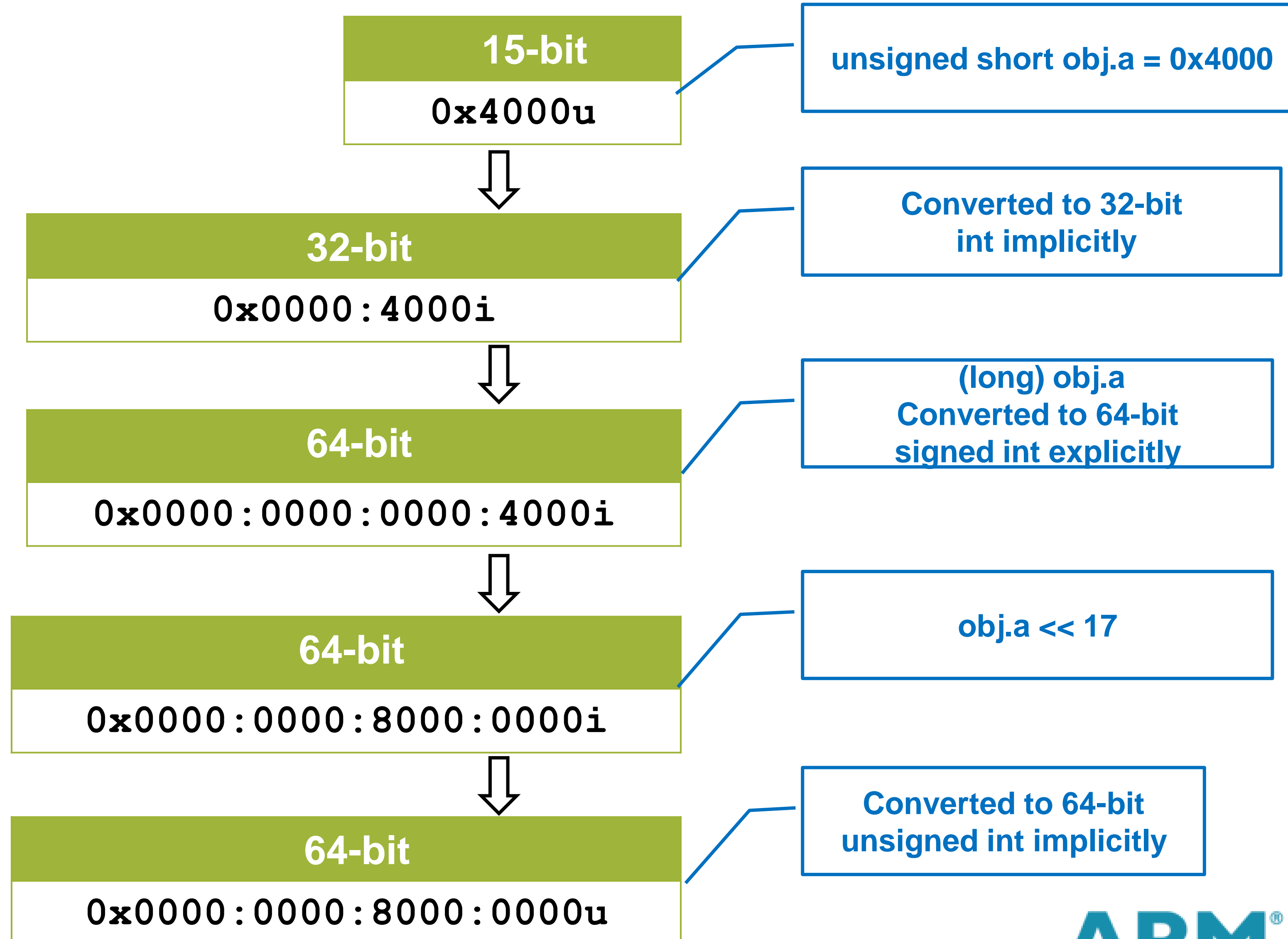


ARM®

Bit field operations and sign extension

- **On 64-bit system(LP64)**

```
struct BitFieldStruct
{
  unsigned short a:15;
  unsigned short b:13;
};

struct BitFieldStruct obj;

unsigned long x;

int main()
{


  obj.a = 0x4000;

  x = (long) obj.a << 17;

  return 0;
}
```



Function Prototypes

- **If a function is called in C without function prototypes, the return value is int**

```
int main()
{
  const size_t Gbyte = 1024 * 1024 * 1024;
  size_t i;
  char *pointers[3];

  // Allocate
  for (i = 0; i != 3; ++i)
    pointers[i] = (char *) malloc(Gbyte);

  // Use
  for (i = 0; i != 3; ++i)
    pointers[i][0] = 1;

  // Free
  for (i = 0; i != 3; ++i)
    free(pointers[i]);
}
```

Compiler warning reported:
Function “malloc” declared implicitly

```
BL      malloc
SXTW    x0, w0
STR     x0, [x21, x20, LSL #3]
```

Does not work on 64-bit system

ARM®

Function Prototypes

- **Solution: Explicitly declare the function prototype**

```
#include <stdlib.h>
int main()
{
  const size_t Gbyte = 1024 * 1024 * 1024;
  size_t i;
  char *pointers[3];

  // Allocate
  for (i = 0; i != 3; ++i)
    pointers[i] = (char *) malloc(Gbyte);

  // Use
  for (i = 0; i != 3; ++i)
    pointers[i][0] = 1;

  // Free
  for (i = 0; i != 3; ++i)
    free(pointers[i]);
}
```

```
BL        malloc
STR       x0, [x21, x20, LSL #3]
```

Works on 64-bit system

ARM®

Magic Numbers

- **Magic numbers could be dangerous**
- **When porting code to a 64-bit platform, take care with magic numbers in address calculations, objects sizes or bit operations**

- **Examples of magic numbers**

Magic number	Description
`4`	Pointer size
`32`	Bits of a pointer type
`0x7FFFFFFF`	Maximum value of a 32-bit signed variable
`0x80000000`	Minimum value of a 32-bit signed variable
`0xFFFFFFFF`	Maximum value of a 32-bit variable - an alternative ‘-1’ as an error sign

ARM®

Magic Numbers

- **Pointer size**

Bad

```
size_t  num= 10;

int **pointerArray =
 (int **) malloc(num * 4);
```

Good

```
size_t  num= 10;

int **pointerArray =
 (int **) malloc(num * size_of(int *));
```

- **'-1' and 0xFFFF_FFFF are often used to return an error**

Bad

```
#define ERROR_CODE 0xffffffff
size_t  bar()
{
    return ERROR_CODE;
}
size_t  foo()
{
    if (bar() == -1)
     …
}
```

Good

```
#define ERROR_CODE ((size_t)(-1))
size_t  bar()
{
    return ERROR_CODE ;
}
int foo()
{
    if (bar()== -1)
      …
}
```


ARM®

Structure padding

```
struct foo
{
    int a;
    void* p;
    int b;
};
```

32-bit layout	64-bit layout

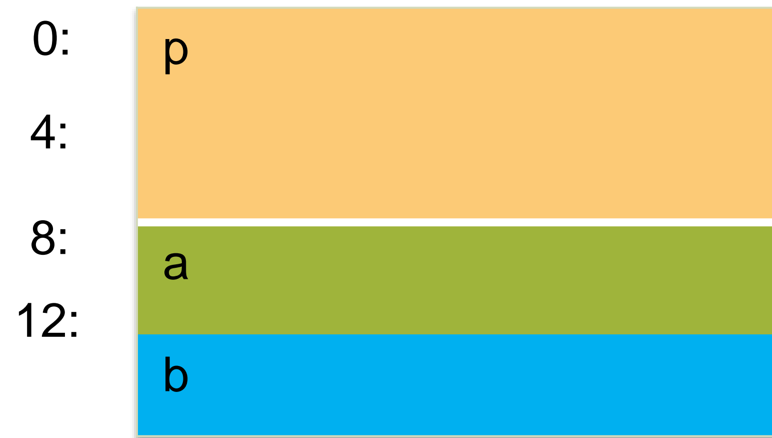
ARM®

Performance consideration

- **This is much better…**

```
struct foo
{
    void* p;
    int a;
    int b;
};
```

32 bit layout	64 bit layout



ARM®

Agenda

C code considerations

- Porting ARMv7-A NEON/VFP code to AArch64

Assembly code porting

ARM®

Porting NEON code overview

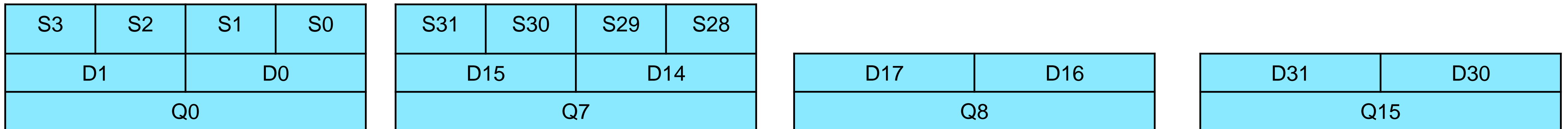
- **ARMv7-A and ARMv8-A AArch64 NEON are very similar**
  - Similar functionality and data types
  - It should be fairly easy to port ARMv7-A NEON code to ARMv8-A AArch64

- **Note the following differences**
  - Number of NEON registers and their layout
  - NEON instruction syntax
  - A few new NEON instructions
  - AArch64 NEON supports double precision floating point type (F64)
  - AArch64 NEON floating point is fully IEEE754 compliant
  - New “scalar” instructions for loop head/tail in vector registers

ARM®

NEON register layout

- **ARMv7-A NEON has a 256-byte register file**
  - This can be accessed as either 32 x 32-bit S registers, 32 x 64-bit D registers, or 16 x 128-bit Q registers
  - The registers overlap as shown:



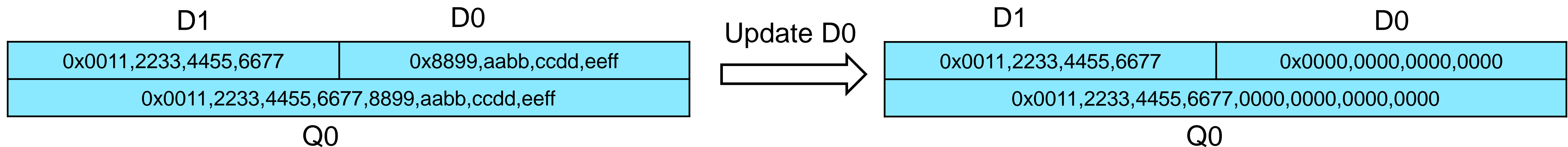
- **ARMv8-A AArch64 NEON has a 512-byte register file**
  - This can be accessed as 32 x 8-bit B registers, 32 x 16-bit H registers, 32 x 32-bit S registers, 32 x 64-bit D registers or 32 x 128-bit V registers
  - The registers overlap differently, as shown:

  - Higher performance can be obtained by making use of more NEON registers

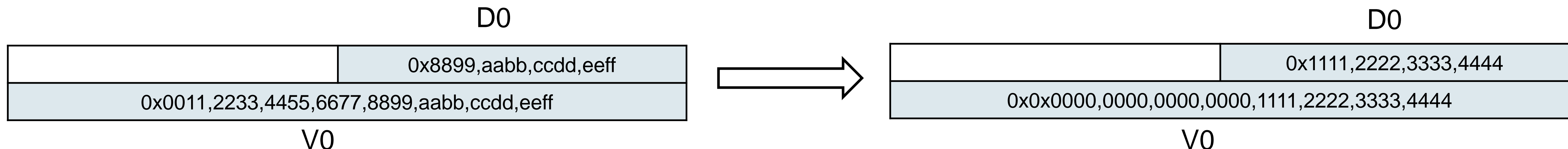
ARM®

NEON register

- **ARMv7-A NEON: when writing D registers, unwritten bytes in the corresponding Q register are not modified**
  - For example, `VMOV.i8 D0, #0`



- **ARMv8-A AArch64 NEON: when writing vectors held in the least significant 64 bits of a V register, unwritten bytes of this V register are set to zero**
  - For example, `MOVI V0.8B, #0x1111222233334444`



ARM®

NEON instruction mnemonic

- **Difference in the mnemonics between ARMv7-A and ARMv8-A AArch64 NEON**
  - ARMv7-A mnemonic prefix ‘V’ has been removed or replaced by S/U/F/P which indicates the data type as signed/unsigned/float/polynomial
  - Vector organization (element size/number of lanes) is defined by the register qualifiers and not by the mnemonics as in ARMv7-A
    - When accessing a SIMD vector in AArch64, the V register name is used, with an extension to indicate the number and size of elements in the vector
    - For example, `V7.4H` – Four 16-bit elements in `V7[63:0]`
  - V suffix for new “across-lanes” reduction operations (`ADDV, SMINV`, etc.)

AArch64	ARMv7-A
`ADD v0.8H, v3.8H, v4.8H`	`VADD.I16 q0, q3, q4`
`FADD v5.2S, v1.2S, v9.2S`	`VADD.F32 d10, d2, d18`
`MOV v7.B[3], W4`	`VMOV.I8 d14[3], r4`
`FADD v5.2D, v1.2D, v9.2D`	`N/A`
`ADDV B1, v2.8B`	`N/A`

- **NEON intrinsics are recommended for generating portable NEON code**

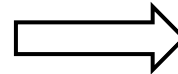
ARM®

Porting example

- **In most cases, porting ARMv7-A NEON code to ARMv8-A AArch64 is straight-forward**
- **A Matrix Multiply example**

ARMv7-A NEON

```
__asm void mutiply_neon(float *a,  float *b,
float *c)
{
   VLD1.F32  {Q1,Q2},    [r2]!
   VLD1.F32  {Q3,Q4},    [r2]!

   MOV  r4,  #4
Loop4x4
   VMOV.F32      Q5,    #0

   VLD1.F32      Q0,    [r1]!

   VMUL.F32      Q5,    Q1,  Q0[0]
   VMLA.F32      Q5,    Q2,  Q0[1]
   VMLA.F32      Q5,    Q3,  Q0[2]
   VMLA.F32      Q5,    Q4,  Q0[3]

   VST1.F32     Q5,   [r0]!
   SUBS       r4,    r4,   #1
   BNE        Loop4x4
   BX         lr
}
```



ARMv8-A AArch64 NEON

```
__asm void mutiply_neon(float *a,  float *b,
float *c)
{
  LD1 {V0.4S, V1.4S, V2.4S, V3.4S}, [X2]


  MOV   x4,  #4
Loop4x4
  MOVI V5.4S, #0

  LD1 {V4.4S},  [X1], #16

  FMUL V5.4S,  V0.4S,   V4.S[0]
  FMLA V5.4S,  V1.4S,   V4.S[1]
  FMLA V5.4S,  V2.4S,   V4.S[2]
  FMLA V5.4S,  V3.4S,   V4.S[3]

  ST1  {V5.4S},  [X0], #16
  SUBS  X4,  X4,  #1
  B.NE  Loop4x4
  RET
}
```

ARM®

New “scalar” instructions

- **Allow full loop, head/body/tail to be implemented within NEON**

ARMv7-A	ARMv8-A AArch64
`VMAX.F32 Q0, Q1, Q2`	`FMAX v0.2D, v1.2D, v2.2D` `FMAX D0, D1, D2`
`VNEG.F64 D1,D2`	`FNEG v1.2S, v2.2S` `FNEG D1, D2`

- **Other examples of instructions that support both scalar and vector form:**

Vector AArch64	Scalar AArch64
`FADD V0.2S, V1.2S, V2.2S`	`FADD D0, D1, D2`
`FSQRT V0.2S, V1.2S, V2.2S`	`FSQRT D0, D1, D2`
`FCVTPS V0.2S, V1.2S`	`FCVTPS X1, D1`

ARM®

New A64 NEON instructions

AArch64	Description
FCVTxS/FCVTxU	Vector floating-point convert to signed/ unsigned integer (round to x)
FCVTXN	Vector convert double-precision to single-precision (rounding to odd)
FDIV	Vector floating point divide
INS	Insert single element in another element
FMAXNM/FMINNM	Floating-point vector maxNum/minNum
FMULX	Floating-point vector multiply extended (0xINF→ 2)
FMINNMP/FMAXNMP	Floating-point vector minNum/maxNum pair
SUQADD/USQADD	Signed/Unsigned integer saturating vector accumulate of unsigned/Signed value
RBIT	Vector reverse bits in bytes
ADDV/SADDLV/UADDLV/FMAXV/FMINV	New “across-lanes” reduction operations
FRECPX/FSQRT	Floating-point reciprocal Exponent/vector square root

ARM

Agenda

C code considerations

Porting ARMv7-A NEON/VFP code to AArch64

- Assembly code porting

ARM®

Assembly code porting overview

- Assembly code needs to be rewritten, due to differences between A64 and ARMv7-A
- Porting of existing ARM assembly code is reasonably straightforward
- Refer to ISA, exception, memory management documentation for more details

ARM®

Mapping ARMv7-A and A64 instructions

- A64 provides a set of functionality similar to that of traditional A32 (ARM) and T32 (Thumb)
- In many cases the assembly code looks the same apart from register names - for instance:

ARMv7-A	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`

ARM®

LDM/STM

- There are no `LDM`/`STM`, `PUSH`/`POP` instructions in A64
- Use `LDP`/`STP` instead
  - Examples:

ARMv7-A	A64
`PUSH {r0-r1}`	`STP w0, w1, [sp, #-0x8]!`
`POP {r0-r1}`	`LDP w0, w1, [sp], #0x8`
`LDM r0, {r1,r2}`	`LDP w1, w2, [x0]`
`STM r0, {r1, r2}`	`STP w1, w2, [x0]`

ARM®

Control processor state

- There is no dedicated CPSR register in AArch64
- Instead, there are different ways to access flag bits, enable/disable IFQ/FIQ and set endianness

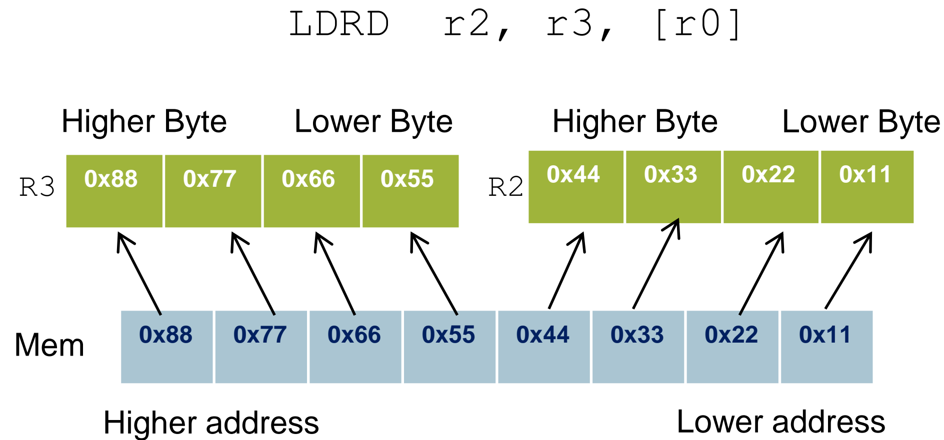
<table>
<tr><th></th><th>ARMv7-A</th><th>A64</th></tr>
<tr><td rowspan="2">Disable IRQ</td><td><code>MRS r0, CPSR
ORR r0, r0, #IRQ_Bit
MSR CPSR_c, r0</code></td><td rowspan="2"><code>MSR DAIFSET, #IRQ_bit</code></td></tr>
<tr><td><code>CPSID I</code></td></tr>
<tr><td>Access ALU Flag bits</td><td><code>MRS r0, CPSR
MSR CPSR_f, r0</code></td><td><code>MRS x0, NZCV
MSR NZCV, x0</code></td></tr>
<tr><td rowspan="2">Set Endianness</td><td><code>MRS r0, CPSR
ORR r0, r0, #E_bit
MSR CPSR_c, r0</code></td><td rowspan="2"><code>SCTLR_ELn.EE</code> controls ELn data endianness
<code>SCTLR_EL1.E0E</code> controls EL0 data endianness

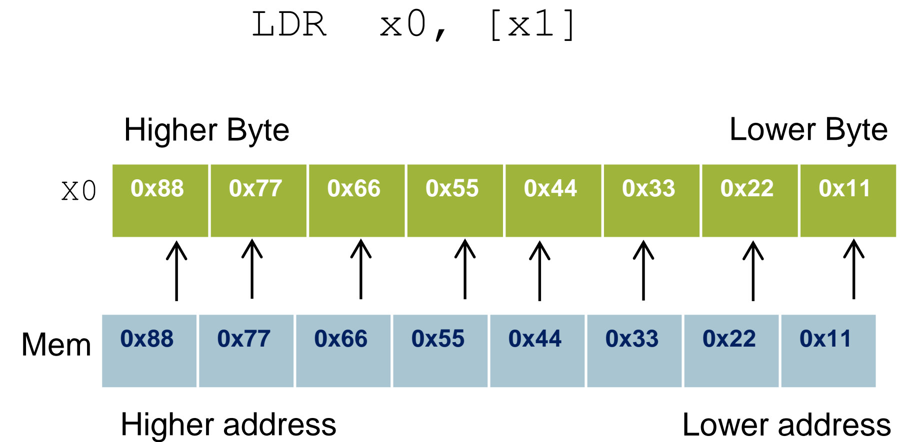
<code>MRS x0, SCTLR_EL1
ORR x0, x0, #EE_bit
MSR SCTLR_EL1, x0</code></td></tr>
<tr><td><code>SETEND BE</code></td></tr>
</table>

ARM®

Little and big endian

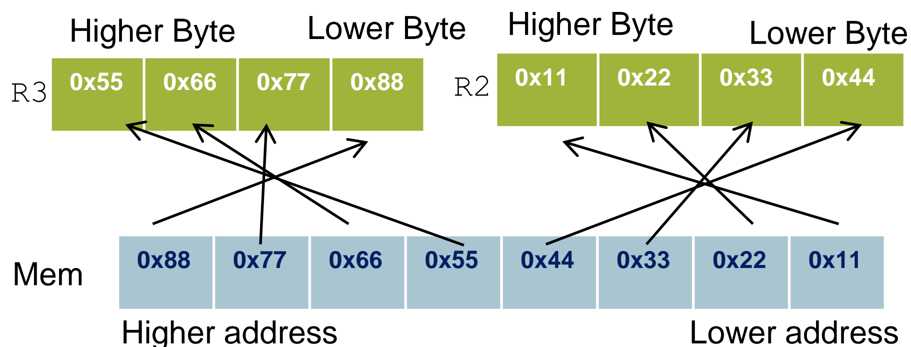
- **Be careful with difference between endianness of 64-bit and 32-bit load/store instructions**

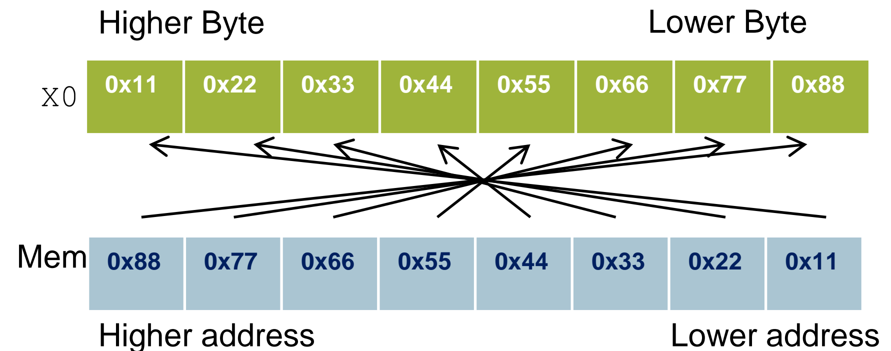




Little Endian

Big Endian





ARM®

Conditional execution

- **Most A64 instructions do not support conditional execution**
  - Flag-setting data processing instructions
    - ADCS, ADDS, ANDS, BICS, SBCS, SUBS, NGCS and the aliases CMN, CMP, and TST
    - Flag setting instructions now set ALL flags (i.e. N Z C V)
  - Compare and branch instructions
    - CBZ/CBNZ/TBZ/TBNZ
- **A64 supports conditional operations**
- **Consider A64 conditional operations if possible for better performance**

```
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

ARM®

Exception return

- **Unlike ARMv7-A, A64 has dedicated instructions for subroutine return and exception return**

	ARMv7-A	A64
Subroutine return	`MOV pc, lr`	`RET`
	`POP {pc}`	
	`BX lr`	
Exception return	`SUBS pc, lr, #4` `MOVS pc, lr`	`ERET`
	`POP {pc}^`	
	`RFE sp!`	

ARM®

The Program Counter (PC)

- **In AArch64, the PC cannot be directly modified by pseudo-instructions**

```
LDR  x0,  =label
LDR  x0,  label
```

- **Indirect branches in the vector table are also different:**

ARMv7-A vector table

```
    LDR pc, = IRQ_handler_addr
    ……
IRQ_handler_addr DCD IRQ_handler
```

AArch64 vector table

```
    LDR  x0, = IRQ_handler_addr
    BX   x0
    ……
IRQ_handler_addr  DCD IRQ_handler
```

- **When calculating PC-relative addresses explicitly**
  - ARMv7-A A32: PC = current instruction + 8
  - ARMv7-A T32; PC = current instruction + 4
  - ARMv8-A A64: PC = current instruction

ARM®

System control

- A64 does not use MCR/MRC coprocessor instructions for system control
- MSR/MRS are used with named system registers
- Most ARMv7-A system control registers have named equivalents

	ARMv7-A	A64
System control register	MRC p15,0,<Rt>,c1,c0,0 MCR p15,0,<Rt>,c1,c0,0	MRS <Xt>, SCTLR_ELx MRS SCTLR_Elx, <Xt>
Auxiliary Control Register	MRC p15,0,<Rt>,c1,c0,1 MCR p15,0,<Rt>,c1,c0,1	MRS <Xt>, ACTLR_EL3 MSR ACTLR_EL3, <Xt>
Debug Breakpoint Control Registers	MRC p14,0,<Rt>,c0,<CRm>,5 MCR p14,0,<Rt>,c0,<CRm>,5	MRS <Xt>, DBGBCR<n>_EL1 MSR DBGBCR<n>_EL1, <Xt>

ARM®

System Control Registers

- **In ARMv7-A, many system control registers are banked between secure and non-secure world**

- **In ARMv8, those registers are not banked – instead they have distinct names**
  - Many registers map between AArch32 and AArch64 state
  - Secure banked copies of registers map to EL3
  - Some registers naturally become 64-bit wide
  - Registers explicitly named with EL

AArch32/v7-A	AArch64	AArch32/v7-A	AArch64
VPIDR	VPIDR_EL2	HADFSR	ADFSR_EL2
VMPIDR	VMPIDR_EL2	HAIFSR	AIFSR_EL2
CSSELR* (NS)	CSSELR_EL1	HSR	ESR_EL2
SCTLR* (NS)	SCTLR_EL1	DFAR* (NS)	FAR_EL1[31:0]
HCTLR	SCTLR_EL2	IFAR* (NS)	FAR_EL1[63:32]
ACTLR* (NS)	ACTLR_EL1/EL2	HDFAR	FAR_EL2[31:0]
CPACR	CPACR_EL1	HIFAR	FAR_EL2[63:32]
HCPTR	HCPTR_EL2	HPFAR	HPFAR_EL2[31:0]
HCR	HCR_EL2	PAR* (NS)	PAR_EL1
HSTR	HSTR_EL2	PRRR/MAIR0* (NS)	MAIR_EL1[31:0]
TTBR0* (NS)	TTBR0_EL1	NMRR/MAIR1* (NS)	MAIR_EL1[63:32]
TTBR1* (NS)	TTBR1_EL1	HMAIR0	MAIR_EL2[31:0]
HTTBR	TTBR0_EL2	HMAIR1	MAIR_EL2[63:32]
VTTBR	VTTBR0_EL2	VBAR* (NS)	VBAR_EL1[31:0]
TTBCR* (NS)	TCR_EL1	HVBAR	VBAR_EL2[31:0]
HTCR	HCR_EL2	CONTEXTIDR* (NS)	CONTEXTIDR_EL1[31:0]
VTCR	VTCR_EL2	TPIDRURW* (NS)	TPIDR_EL0[31:0]
DACR* (NS)	DACR32_EL2	TPIDRURO* (NS)	TPIDRRO_EL0[31:0]
DFSR* (NS)	ESR_EL1	TPIDRPRW* (NS)	TPIDR_EL1[31:0]
IFSR* (NS)	IFSR32_EL2	HTPIDR	TPIDR_EL2[31:0]
ADFSR* (NS)	ADFSR_EL1	SDER	SDER32_EL3
AIFSR* (NS)	AIFSR_EL1	HDCR	MDCR_EL2

Note: * means banked in secure and non-secure world

ARM®

Cache maintenance

- **ARMv7-A and ARMv8-A have a similar set of cache/TLB maintenance operations**
  - Cache clean, invalidate, clean and invalidate
- **Instruction mnemonics differ**

	A32/T32	A64
Data Cache line Clean and Invalidate by VA to PoC	`MCR p15,0,<Rt>,c7,c14,1`	`DC CIVAC, <Xt>`
Instruction Cache line Invalidate by VA to PoU	`MCR p15,0,<Rt>,c7,c5,1`	`IC IVAU, <Xt>`

- **Data Cache Zero instruction added (`DC ZVA`) to AArch64**
  - Zeros out a block of memory (of Implementation Defined size)

ARM®

ARMv6 SIMD and saturating arithmetic

- **No ARMv6 SIMD or saturating arithmetic on the general registers**
  - NEON must be used for all SIMD and saturating arithmetic

```
QADD  r0, r1, r2
```

```
SQADD s1, s2, s3
```


ARM®

ARMv8 64-bit Migration

The Architecture for the Digital World® ARM®