ARMv8-A CPU Architecture Overview

ARM

Chris Shore

Training Manager, ARM

ARM Game Developer Day, London

03/12/2015

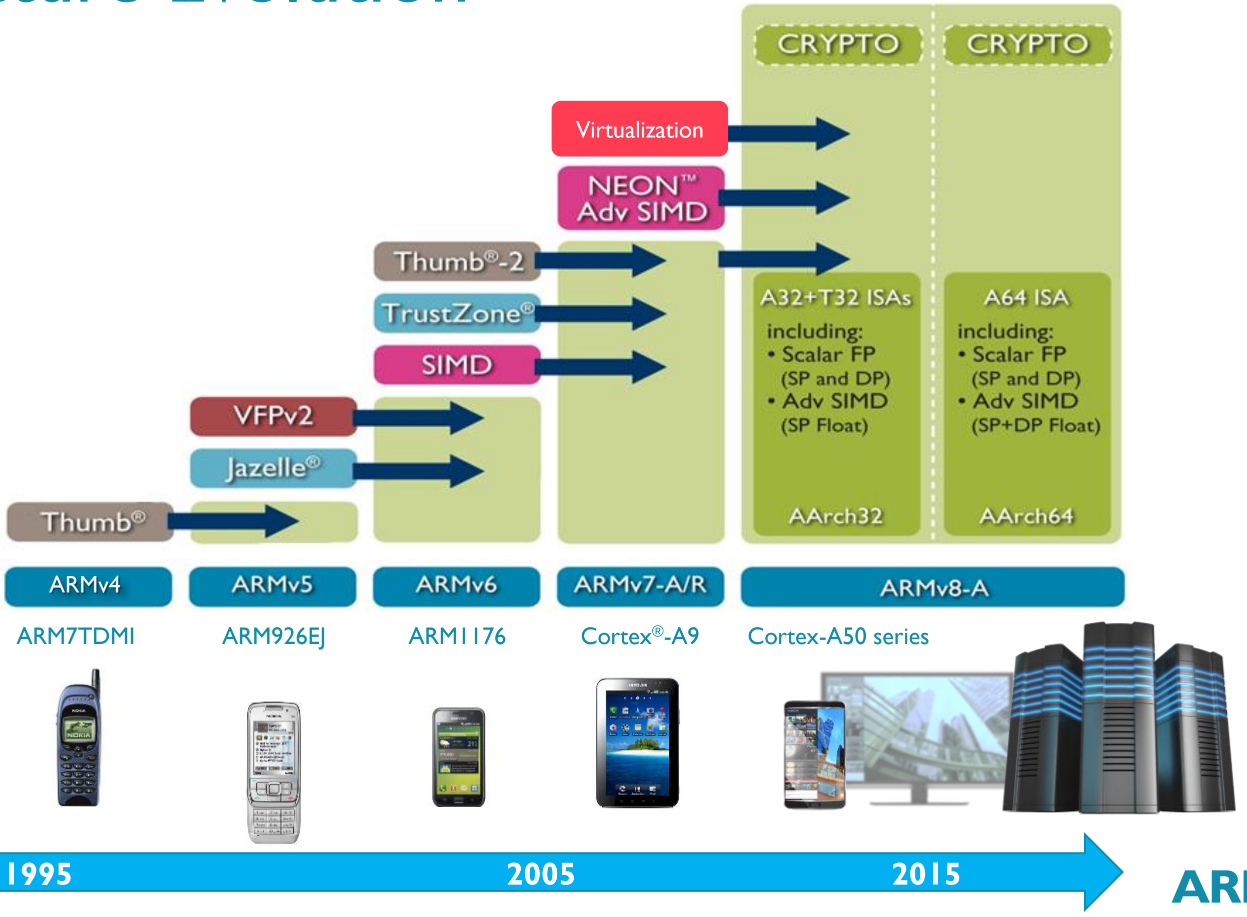
Chris Shore – ARM Training Manager

- With ARM for 16 years
- Managing customer training for 15 years
  - Worldwide customer training delivery
  - Approved Training Centers
  - Active Assist onsite project services

- Background
  - MA Physics & Computer Science, Cambridge University, 1986
  - Background as an embedded software consultant for 17 years
  - Software Engineer
  - Project Manager
  - Technical Director
  - Engineering Manager
  - Training Manager

- Regular conference speaker and trainer

ARM Architecture Evolution

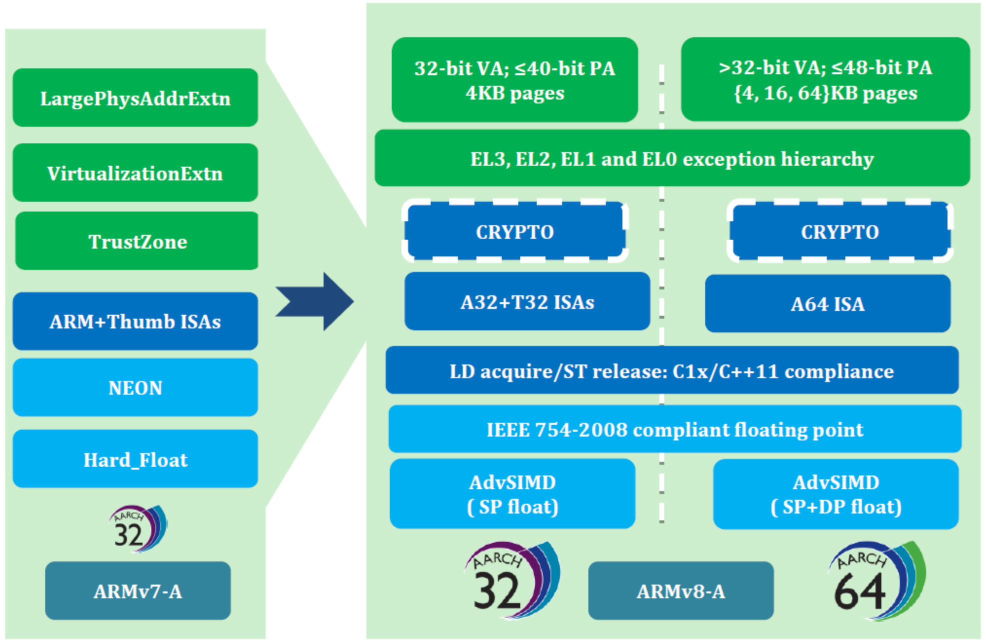


Increasing SoC complexity
Increasing OS complexity
Increasing choice of HW and SW

ARM

ARMv8-A AArch64

- AArch64 supports ALL ARMv8-A features
  - Clean instruction set
  - Larger address space (>4GB memory for Application)
  - Wider data register (64-bit)
  - Better SIMD (NEON)
  - Better floating point
  - Increased number and size of general purpose registers (31 general, 32 FP/NEON/Crypto)
  - Better security architecture, dealing with hypervisors
  - More…



LargePhysAddrExtn
VirtualizationExtn
TrustZone
ARM+Thumb ISAs
NEON
Hard_Float
AARCH 32
ARMv7-A
32-bit VA; ≤40-bit PA 4KB pages
>32-bit VA; ≤48-bit PA {4, 16, 64}KB pages
EL3, EL2, EL1 and EL0 exception hierarchy
CRYPTO
CRYPTO
A32+T32 ISAs
A64 ISA
LD acquire/ST release: C1x/C++11 compliance
IEEE 754-2008 compliant floating point
AdvSIMD (SP float)
AdvSIMD (SP+DP float)
AARCH 32
ARMv8-A
AARCH 64

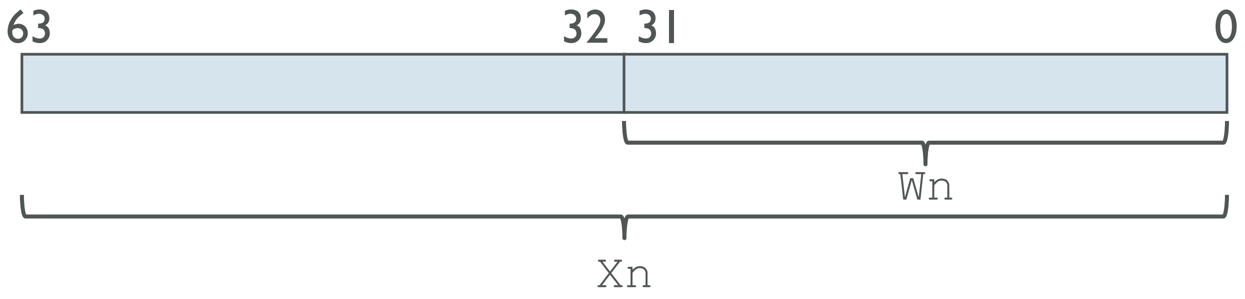
What's new in ARMv8-A?

- ARMv8-A introduces two execution states: AArch32 and AArch64
- AArch32
  - Evolution of ARMv7-A
  - A32 (ARM) and T32 (Thumb) instruction sets
    - ARMv8-A adds some new instructions
  - Traditional ARM exception model
  - Virtual addresses stored in 32-bit registers
- AArch64
  - New 64-bit general purpose registers (X0 to X30)
  - New instructions – A64, fixed length 32-bit instruction set
    - Includes SIMD, floating point and crypto instructions
  - New exception model
  - Virtual addresses now stored in 64-bit registers

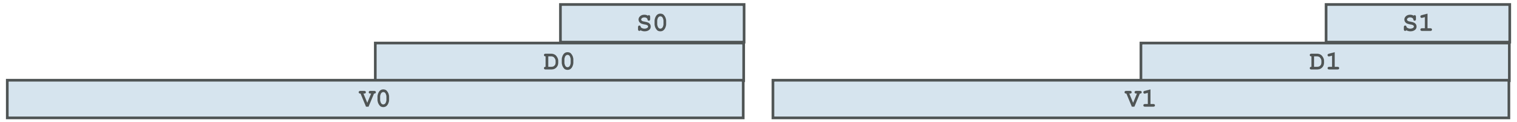
ARM

Register Banks

- AArch64 provides 31 general purpose registers
  - Each register has a 32-bit (w0-w30) and 64-bit (x0-x30) form

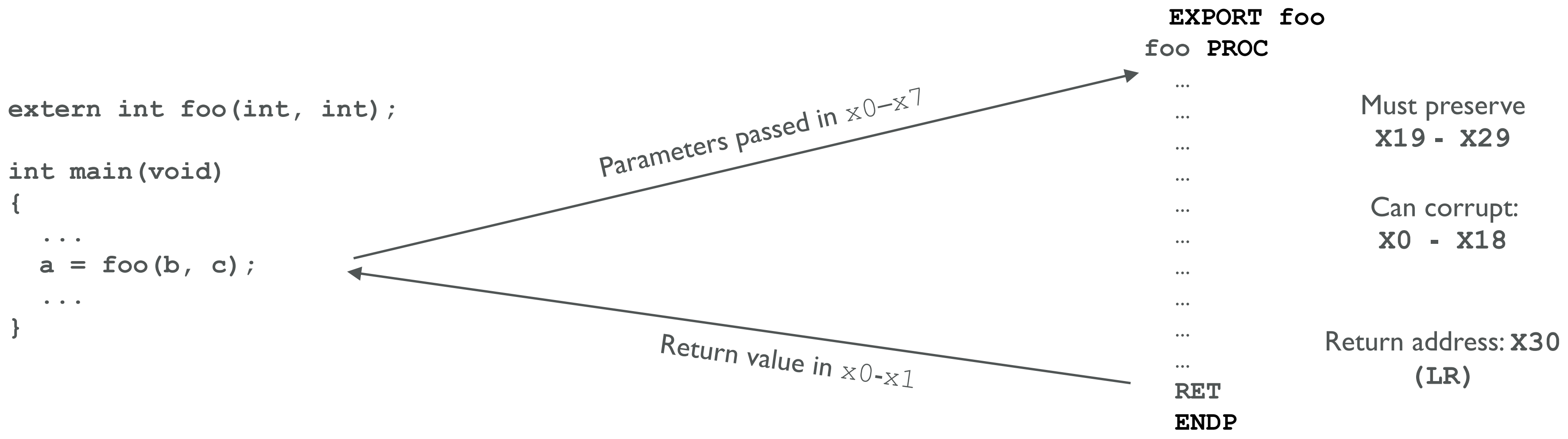


- Separate register file for floating point, SIMD and crypto operations - `Vn`
  - 32 registers, each 128-bits
    - Can also be accessed in 32-bit (Sn) or 64-bit (Dn) forms



Procedure Call Standard

- There is a set of rules known as a Procedure Call Standard (PCS) that specifies how registers should be used:



```
extern int foo(int, int);

int main(void)
{
   ...
   a = foo(b, c);
   ...
}
```

```
  EXPORT foo
foo PROC
   …
   …
   …
   …
   …
   …
   …
   …
   …
   …
   RET
   ENDP
```

A64 Overview

- AArch64 introduces new A64 instruction set
  - Similar set of functionality as traditional A32 (ARM) and T32 (Thumb) ISAs
- Fixed length 32-bit instructions
- Syntax similar to A32 and T32

```
ADD W0, W1, W2          ← w0 = w1 + w2   (32-bit addition)
ADD X0, X1, X2          ← x0 = x1 + x2   (64-bit addition)
```

- Most instructions are not conditional
- Floating point and Advanced SIMD instructions
- Optional cryptographic extensions

Multiprocessing

In the core

ARM NEON tech/SIMD

Use of common parallelizing tools

OpenMP, Renderscript, OpenCL, etc.

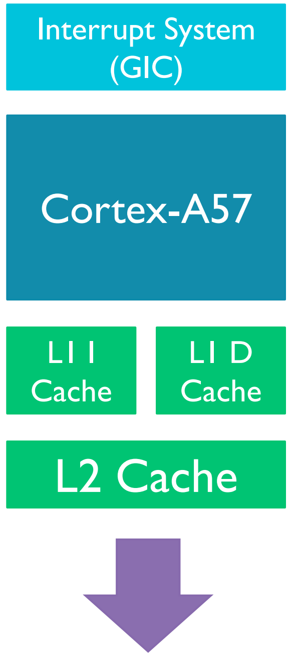
Multi-threading where possible

Never easy, but increasingly necessary

ARM

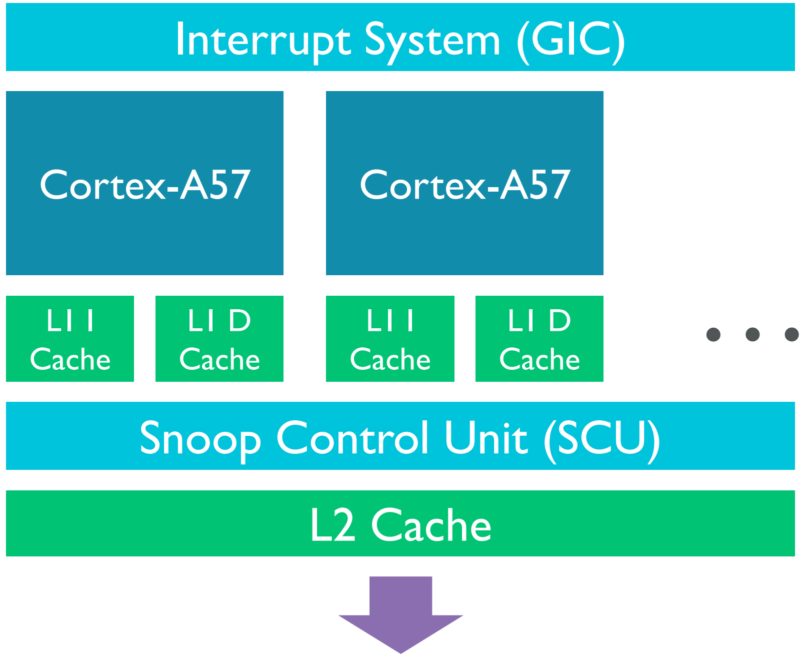
ARM Cortex-A57 MPCore

A useful diagram



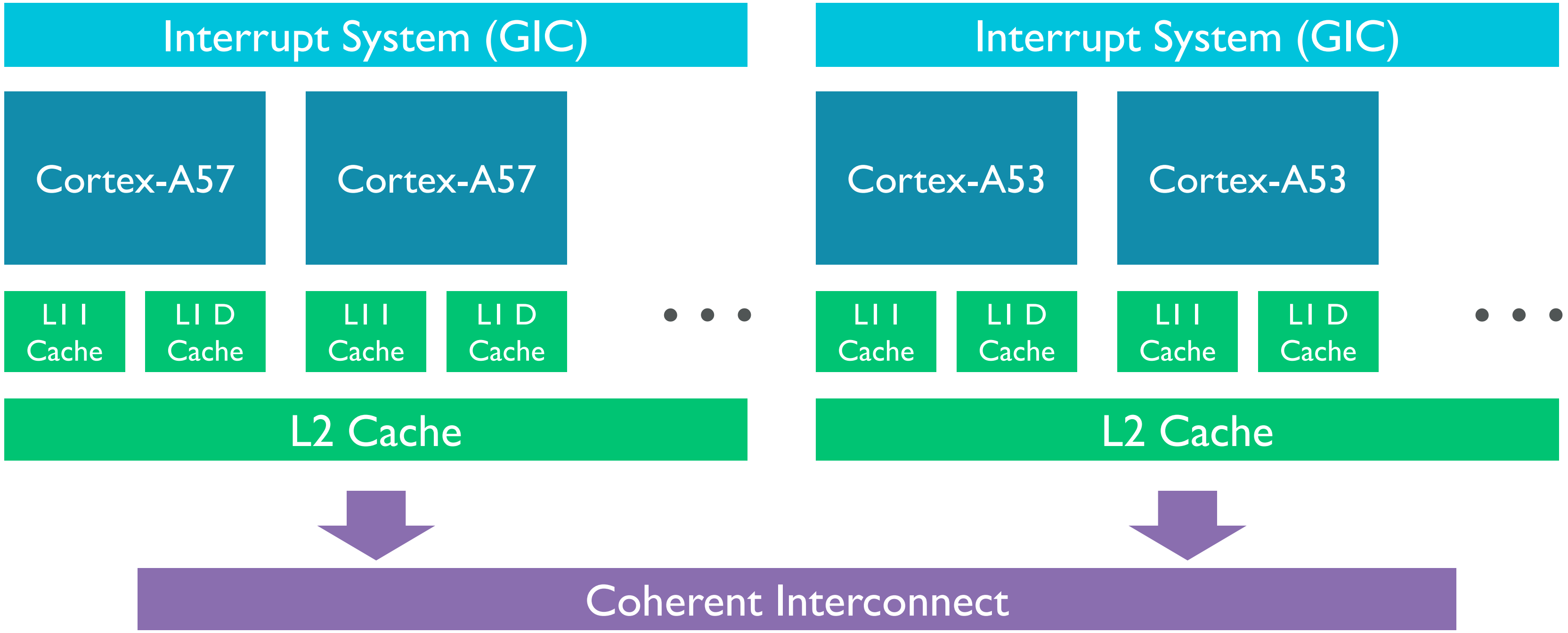
A “Cluster” of Cores

The standard unit in an ARM SMP system



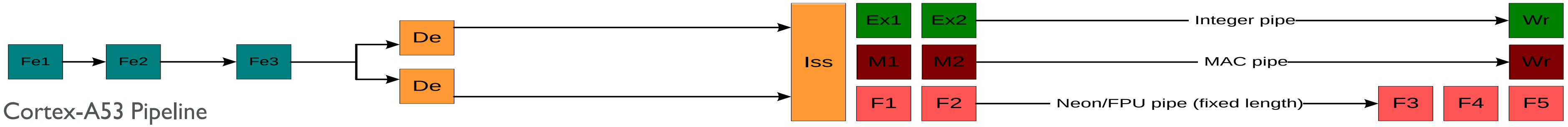
And a Multicluster

System built of more than one cluster, often using different cores

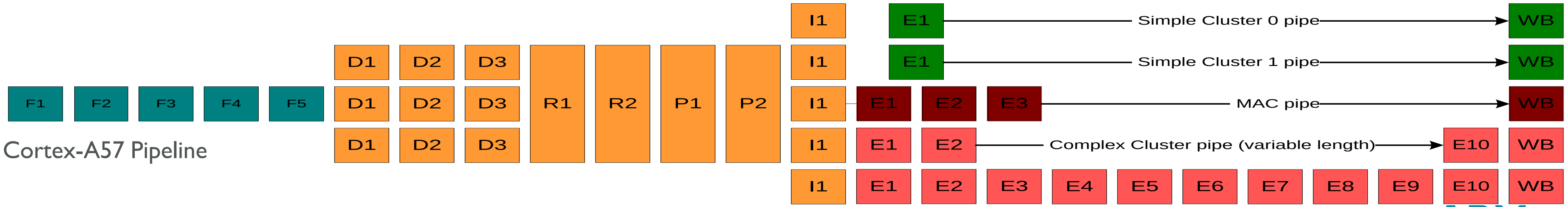


Why is the Power Efficiency so Different?

- Cortex-A53: design focused on energy efficiency (while balancing performance)
  - 8-11 stages, In-Order and limited dual-issue
- Cortex-A57: focused on best performance (while balancing energy efficiency)
  - 15+ stages, Out-of-Order and multi-issue, register renaming
  - Average 40-60% performance boost over Cortex-A9 in general purpose code
    - Instructions per cycle (IPC) improvements



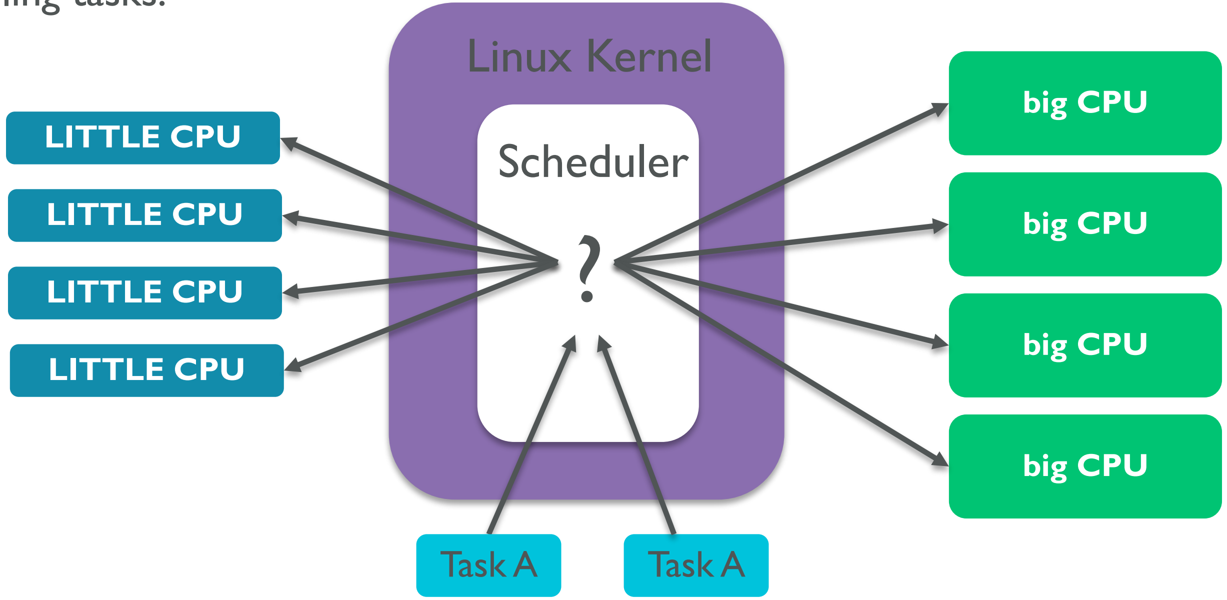
Cortex-A53 Pipeline



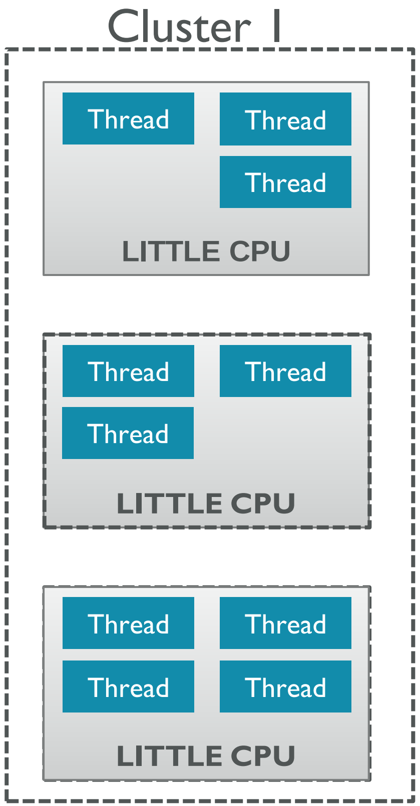
Cortex-A57 Pipeline

Global Task Scheduling (GTS)

- ARM Implementation is called "big.LITTLE MP"
  - Hosted in Linaro git trees but developed by ARM
  - Using GTS, all of the big and LITTLE cores are available to the Linux kernel for scheduling tasks.



Global Task Scheduling (GTS)



1. System starts

Tasks fill up the system

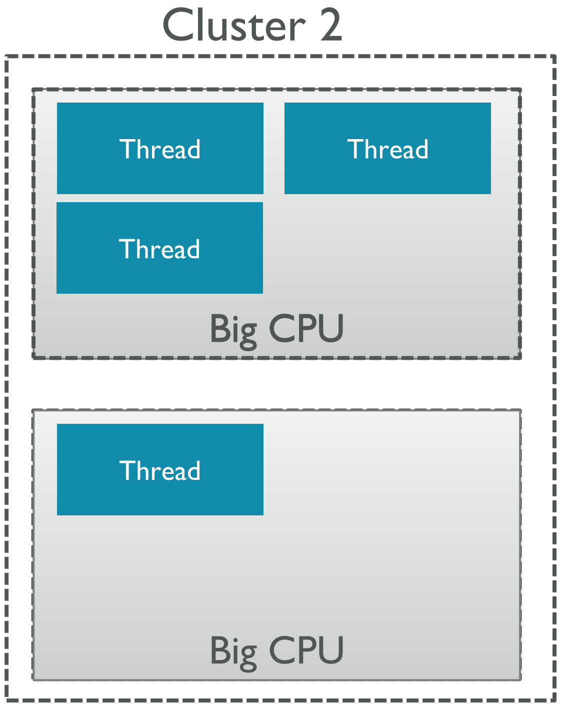
2. Demanding tasks detected

Based on amount of time a thread is ready to run (*run queue residency*)

Moved to a 'big' CPU

3. Task no longer demanding

Moved to a 'LITTLE' CPU



ARM big.LITTLE Development (GTS)

- Trust the scheduler…
  - Linux will schedule for performance and efficiency
  - All new tasks started on big to avoid latency
  - Quickly adapts to a task's needs

…Unless

  - You know a thread is intensive but not urgent
  - Affine to LITTLE, never to big
  - E.g. maybe use this for asset loading on a separate thread

- LITTLE cores are great
  - You'll be using them a lot
  - ARM Cortex-A53 ~20% greater perf than Cortex-A9 cores
  - Most workloads will run on LITTLE
  - More thermal headroom for other SoC components

- big cores are serious powerhouses
  - Think of them as short-burst accelerators – e.g. physics-based special effects
  - Think about the trade-offs during design

ARM big.LITTLE Development

Things to avoid

- Imbalanced threads sharing common data
  - Cluster coherency is excellent but not free

- If you have real-time (RT) threads note that…
  - RT threads are not auto migrated
  - RT threads are a design decision, think carefully about affinity
  - http://linux.die.net/man/2/sched_setaffinity

- Avoid long running tasks on big cores
  - You'll rarely need that processing power for long periods
  - Can the task be parallelized?

Takeaways

- ARM big.LITTLE technology & Global Task Scheduling is easily available
  - Fantastic peak performance
  - Energy-efficient, sustainable compute for long running workloads
- Multi-processing
  - Get ahead of the limits on single thread performance
  - Avoid thermal constraints on performance

What is NEON?

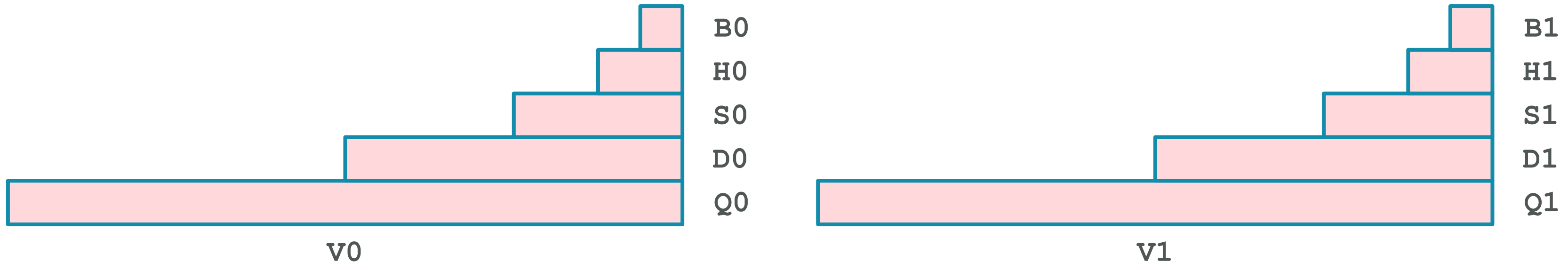
- SIMD data processing engine
  - A **S**ingle **I**nstruction operates on **M**ultiple **D**ata
  - Generally a common operation is carried out in parallel on pairs of elements in vector registers

- Provided as an extension to the instruction and register sets
  - Can be implemented on all Cortex-A series processors
  - NEON instructions are part of the ARM or Thumb instruction stream

- Instructions operate on vectors of elements of the same data type
  - Data types may be floating point or integer
  - Integers may be signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit
  - Single precision floating point (32-bit)
  - Most instructions perform the same operation in all **lanes**

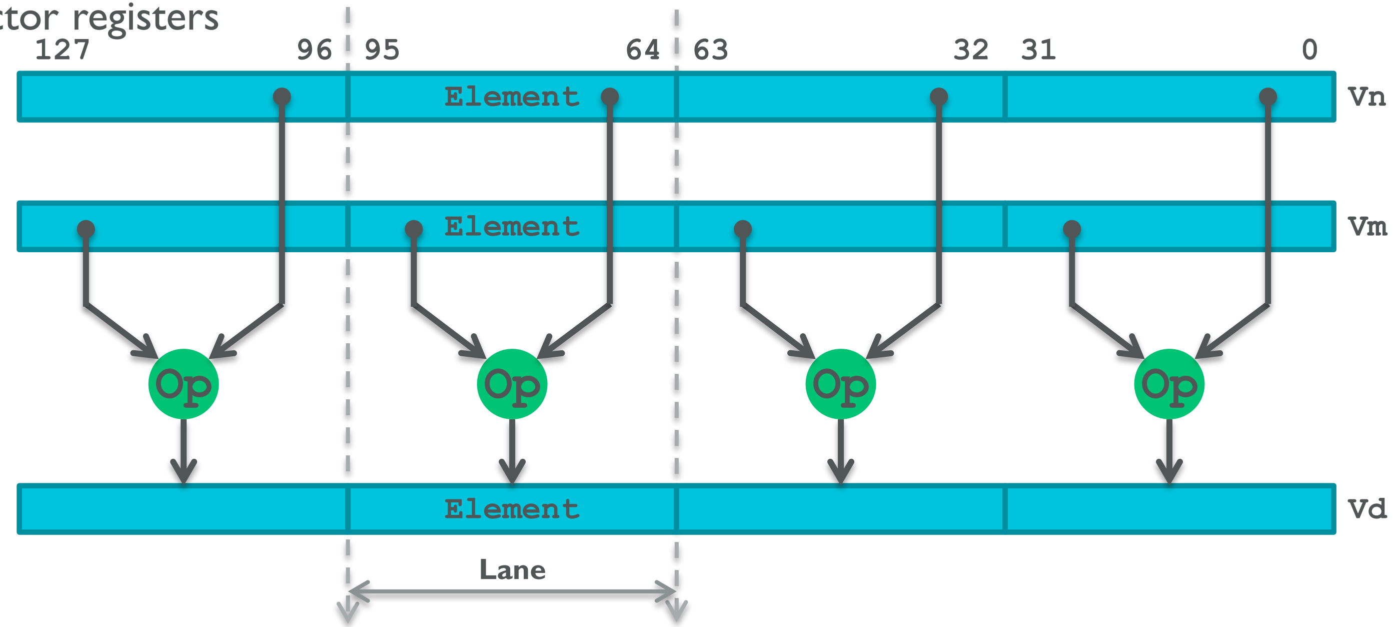
SIMD Registers

- Separate set of 32 registers, each 128-bit wide
  - Architecturally named **V0** – **V31**
  - Used by scalar floating-point and SIMD instructions
- The instruction syntax uses qualified register names
  - **Bn** for byte, **Hn** for half-word, **Sn** for single-word, **Dn** for double-word, **Qn** for quad-word



SIMD Operations

- Operand registers are treated as vectors of individual elements
- Operations are performed simultaneously on a number of "lanes"
  - In this example each lane contains a pair of 32-bit elements, one from each of the operand 128-bit vector registers



ARM

Vectors

- When accessing a SIMD vector, the `Vn` register name is used with an extension to indicate the number and size of elements in the vector
- `Vn.xy`
  - **n** is the register number, **x** is the number of elements, **y** is the size of the elements encoded as a letter

```
FADD V0.2D, V5.2D, V6.2D      ; 2x double-precision floats
```

- Total vector length must be either 128-bits or 64-bits
  - If 64-bits are written, **`Vn[127:64]`** are automatically cleared to 0

```
ADD  V0.8H, V3.8H, V4.8H      ; 8x 16-bit integers
ADD  V0.8B, V3.8B, V4.8B      ; 8x 8-bit integers, clear top of V0
```

- Some instructions refer to a single element of a vector
  - Example: **`V3.B[3]`** – Byte in **`V3[23:16]`**
  - Rest of register is unaffected when an element is written

```
MUL  V0.4S, V2.4S, V3.S[2]    ; Multiply each element of V2 by V3.S[2]
```

Vectorizing Example I

- Optimize for code size
- No vectorization

```
void add_int(int * pa, int * pb, unsigned int n, int x)
{
     unsigned int i;
     for(i = 0; i < n; i++)
          pa[i] = pb[i] + x;
}
```

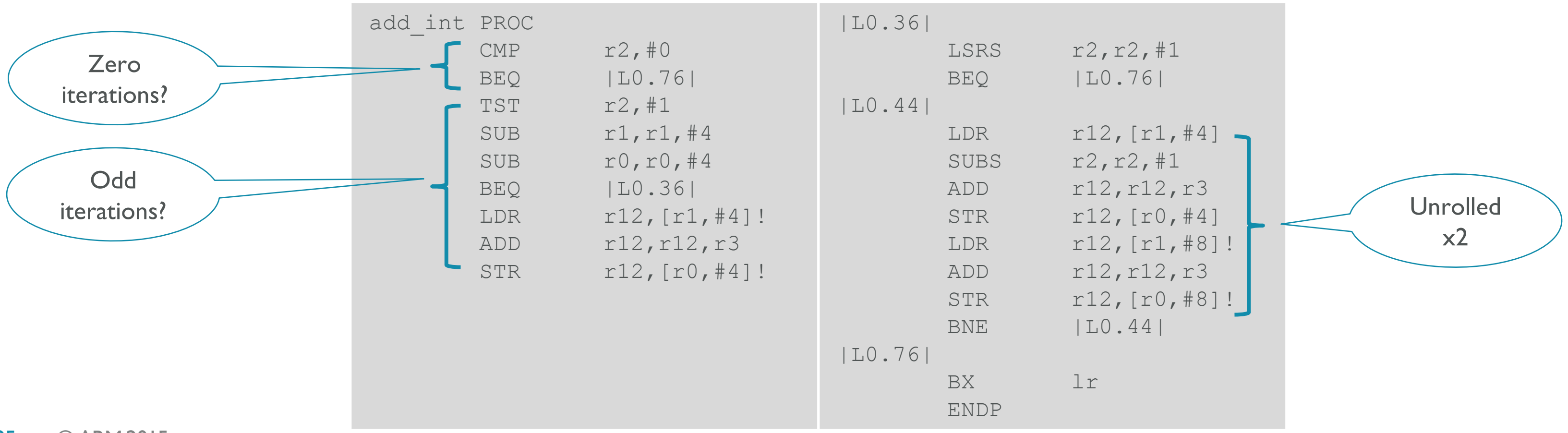
```
armcc --cpu=Cortex-A15 -O3 -Ospace
```

```
add_int PROC
        PUSH     {r4,r5,lr}
        MOV      r4,#0
        B        |L0.28|
|L0.12|
        LDR      r5,[r1,r4,LSL #2]
        ADD      r5,r5,r3
        STR      r5,[r0,r4,LSL #2]
        ADD      r4,r4,#1
|L0.28|
        CMP      r4,r2
        BCC      |L0.12|
        POP      {r4,r5,pc}
        ENDP
```

Vectorizing Example 2

- Optimize for speed
- No vectorization
- Penalty of code size increase

```
void add_int(int * pa, int * pb, unsigned int n, int x)
{
     unsigned int i;
     for(i = 0; i < n; i++)
          pa[i] = pb[i] + x;
}
Compilation options: armcc --cpu=Cortex-A15 -O3 -Otime
```



```
add_int PROC
      CMP          r2,#0
      BEQ          |L0.76|
      TST          r2,#1
      SUB          r1,r1,#4
      SUB          r0,r0,#4
      BEQ          |L0.36|
      LDR          r12,[r1,#4]!
      ADD          r12,r12,r3
      STR          r12,[r0,#4]!
```

```
|L0.36|
        LSRS      r2,r2,#1
        BEQ       |L0.76|
|L0.44|
        LDR       r12,[r1,#4]
        SUBS      r2,r2,#1
        ADD       r12,r12,r3
        STR       r12,[r0,#4]
        LDR       r12,[r1,#8]!
        ADD       r12,r12,r3
        STR       r12,[r0,#8]!
        BNE       |L0.44|
|L0.76|
        BX        lr
        ENDP
```

Vectorizing Example 3

- Optimize for speed
- Number of iterations is a power of 2
- No vectorization

```
void add_int(int *pa, int * pb, unsigned int n, int x)
{
     unsigned int i;
     for(i = 0; i < (n&~3); i++)
          pa[i] = pb[i] + x;
}
Compilation options: armcc --cpu=Cortex-A15 -O3 -Otime
```

Divisible by 4

```
add_int PROC
        BICS     r12,r2,#3
        BEQ      |L0.56|
        LSR      r2,r2,#2
        SUB      r1,r1,#4
        SUB      r0,r0,#4
        LSL      r2,r2,#1
```

```
|L0.24|
        LDR      r12,[r1,#4]
        SUBS     r2,r2,#1
        ADD      r12,r12,r3
        STR      r12,[r0,#4]
        LDR      r12,[r1,#8]!
        ADD      r12,r12,r3
        STR      r12,[r0,#8]!
        BNE      |L0.24|
|L0.56|
        BX       lr
        ENDP
```

Vectorizing Example 4

- Optimize for speed
- Vectorization

```
void add_int(int * pa, int * pb, unsigned int n, int x)
{
     unsigned int i;
     for(i = 0; i < n; i++)
          pa[i] = pb[i] + x;
}
armcc --cpu=Cortex-A15 -O3 -Otime --vectorize
```

Overlap?

```
add_int PROC
        PUSH     {r4-r6}
        SUB      r4,r0,r1
        ASR      r12,r4,#2
        CMP      r12,#0
        BLE      |L0.32|
        BIC      r12,r2,#3
        CMP      r12,r4,ASR #2
        BHI      |L0.76|
|L0.32|
        BICS     r12,r2,#3
        BEQ      |L0.68|
        VDUP.32  q0,r3
        LSR      r2,r2,#2
```

Vector version

```
|L0.48|
        VLD1.32   {d2,d3},[r1]!
        VADD.I32  q1,q1,q0
        VST1.32   {d2,d3},[r0]!
        SUBS      r2,r2,#1
        BNE       |L0.48|
|L0.68|
        POP       {r4-r6}
        BX        lr
|L0.76|
        BIC       r2,r2,#3
        CMP       r2,#0
        MOVNE     r2,#0
        BEQ       |L0.68|
        BLS       |L0.68|
```

Scalar version

```
|L0.96|
        LDR      r6,[r1,r2,LSL #2]
        ADD      r5,r1,r2,LSL #2
        ADD      r6,r6,r3
        STR      r6,[r0,r2,LSL #2]
        LDR      r5,[r5,#4]
        ADD      r4,r0,r2,LSL #2
        ADD      r2,r2,#2
        ADD      r5,r5,r3
        CMP      r12,r2
        STR      r5,[r4,#4]
        BHI      |L0.96|
        POP      {r4-r6}
        BX       lr
        ENDP
```

Vectorizing Example 5

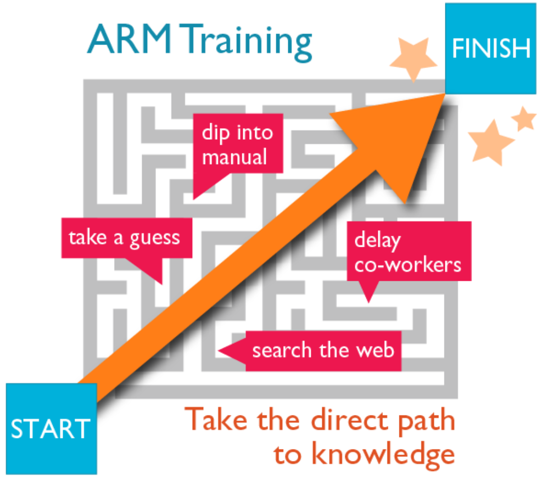
```
void add_int(int* restrict pa, int* restrict pb, unsigned int n, int x)
{
     unsigned int i;
     for(i = 0; i < (n & ~3); i++)
          pa[i] = pb[i] + x;
}
Compilation options:     >armcc --cpu=Cortex-A15 -O3 -Otime --vectorize --c99
```

```
add_int PROC
        BICS      r12,r2,#3
        BEQ       |L0.36|
        VDUP.32   q0,r3
        LSR       r2,r2,#2
|L0.16|
        VLD1.32   {d2,d3},[r1]!
        VADD.I32  q1,q1,q0
        VST1.32   {d2,d3},[r0]!
        SUBS      r2,r2,#1
        BNE       |L0.16|
|L0.36|
        BX        lr
        ENDP
```

- Optimize for speed
- Vectorization
- Number of loop iterations is a power of 2
- Non-Overlapping pointers, use of “restrict”

ARM Technical Training

- Available
  - Covering all ARM cores and GPUs
  - Hardware and Software options
  - Customizable agendas from 2 hours (remote only) to 5 days
- Accessible
  - Private, onsite delivery for corporate customers
  - Public course schedule for open enrolment
  - Live Remote webex for flexible delivery
- Affordable
  - Standard private course fees include all ARM's related expenses
  - Public course pricing accessible to individuals
  - Live Remote is cost effective for small teams
- Learn from the experts



http://www.arm.com/training

For more information visit the Mali Developer Centre:
http://malideveloper.arm.com

- Revisit this talk in PDF and audio format post event
- Download tools and resources

ARMv8-A Architecture Overview

ARM

Thank you