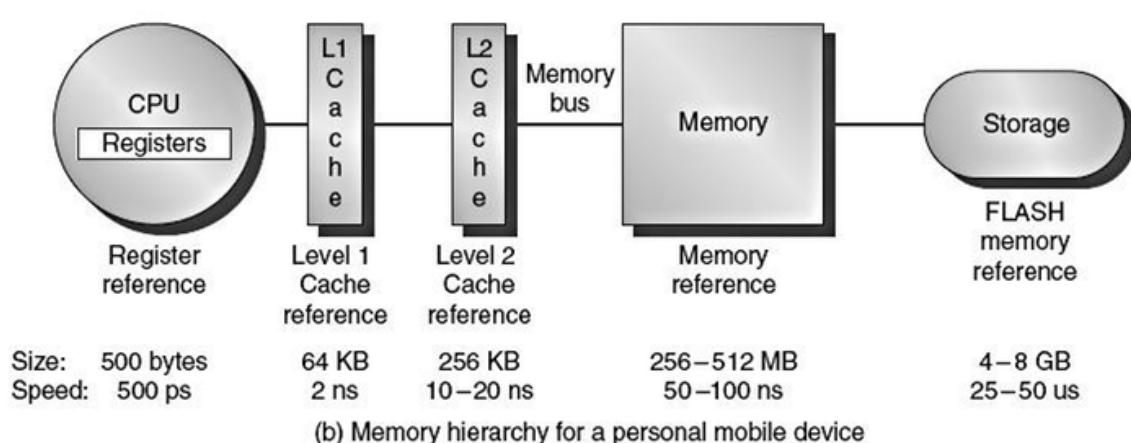
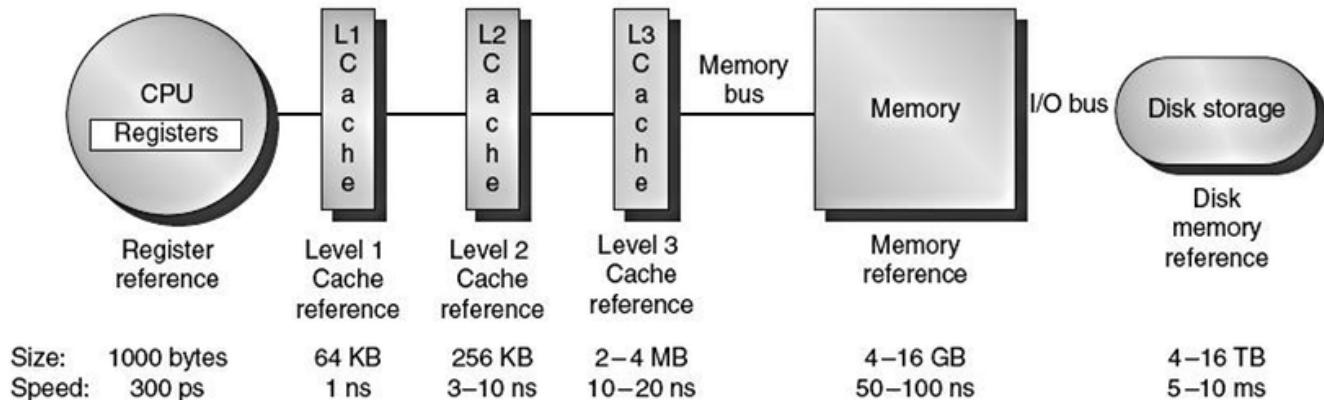


# ARM Architecture

(返回[2016年暑期系統軟體課程：台北場次](#))

## ARM 市場訊息

- [The backstory of how ARM reached a milestone of 86 billion chips in 25 years](#)
- [採精簡核心設計思維的 A73](#)
- [Softbank 收購 ARM 的啟示](#)



CMU [CSAPP](#) 的示意圖來說明記憶體技術演化

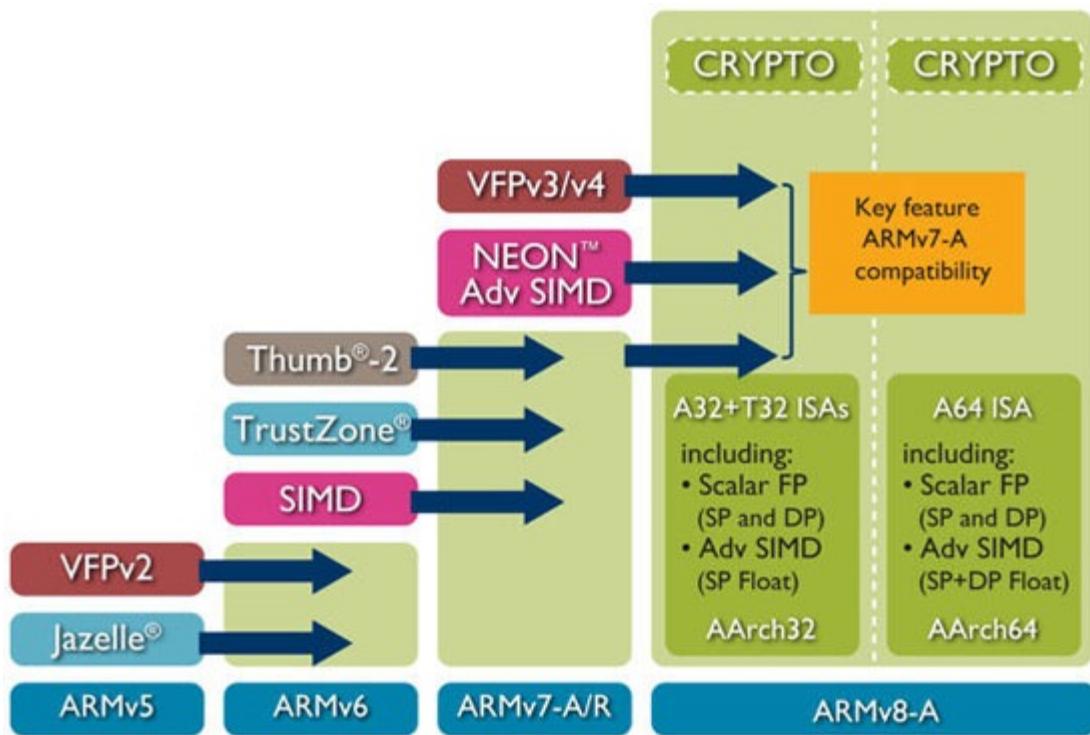
- 上圖是典型伺服器系統中的表現:
  - CPU register (300 ps / 1KB) -> L1 cache (1 ns / 64 KB) -> L2 cache (3-10 ns / 256 KB) -> RAM (50-100 ns / 4-16 GB)
- 下圖為 2010 年智慧型手機的表現:
  - CPU register (500 ps / 500 B) -> L1 cache (2 ns / 64 KB) -> L2 cache (10-20 ns / 256 KB) -> RAM (50-100 ns / 256-512 MB)

## ARM 與 MIPS

	<b>ARM Cortex-A7</b>	<b>ARM Cortex-A9</b>	<b>Imagination MIPS interAptiv</b>	<b>Imagination MIPS interAptiv</b>
<b>CPU Arch.</b>	ARMv7A	ARMv7A	MIPS32-R5	MIPS32-R5
<b>SMP</b>	2–4x SMP, hardware coherence	2–4x SMP, hardware coherence	2–4x SMP, hardware coherence	2–4x SMP, hardware coherence
<b>Pipeline</b>	8 stages	9–10 stages	9 stages	9 stages
<b>Instr-Issue Rate</b>	1 per cycle	2 per cycle	1 per cycle	1 per cycle
<b>CoreMarks per MHz</b>	2.6 CM/MHz§	3.9 CM/MHz	3.2 CM/MHz (2 threads)	3.2 CM/MHz (2 threads)
<b>DMIPS per MHz</b>	1.9 DMIPS/MHz	2.5 DMIPS/MHz	1.57 DMIPS/MHz	1.57 DMIPS/MHz
<b>Max CPU Speed</b>	1.6GHz (typical)	2.0GHz (typical)	2.0GHz (typical)§	2.0GHz (typical)§
<b>Closely Coupled Memories</b>	None	None	Up to 1MB instr + 1MB data	Up to 1MB instr + 1MB data
<b>L1 Instr &amp; Data Caches</b>	Up to 64KB each	Up to 64KB each	Up to 64KB each	Up to 64KB each
<b>L2 Cache</b>	Up to 1MB	Up to 8MB	Up to 8MB	Up to 8MB
<b>I/O (Accel) Coherency</b>	Yes	Yes	Yes	Yes
<b>FPU</b>	SP/DP	SP/DP	SP/DP	SP/DP
<b>MACs per Cycle</b>	1x 32x32-bit or 2x 16x16-bit	1x 32x32-bit or 2x 16x16-bit	2x 16x16-bit	2x 16x16-bit
<b>Extra Features</b>	Neon SIMD extensions, 40-bit memory address	Neon SIMD extensions, Java acceleration	Dual threading, virtualization	Dual threading, virtualization
<b>SoC Interface</b>	2x 32- or 64-bit AXI, AHB-Lite, BVCI	128-bit AMBA4 (ACE), APB slave	2x 64-bit AMBA3 AXI masters	64-bit OCP

- [ARM Processor Architecture](#), ARM Ltd.
- [MIPS Processor](#), Imagination Technologies Ltd.

## ARM Processor Architecture



ARMv8 中文翻譯可見此：<http://wiki.csie.ncku.edu.tw/embedded/ARMv8>

**A64** is a new 32-bit fixed length instruction set to support the AArch64 execution state. The following is a summary of the A64 ISA features.

- Clean decode table based on 5-bit register specifiers
- Instruction semantics broadly the same as in AArch32
- 31 general purpose 64-bit registers accessible at all times
- No modal banking of GP registers - Improved performance and energy
- Program counter (PC) and Stack pointer (SP) not general purpose registers
- Dedicated zero register available for most instructions

ARM, generically known as **A32**, is a fixed-length (32-bit) instruction set. It is the base 32-bit ISA used in the ARMv4T, ARMv5TEJ and ARMv6 architectures. In these architectures it is used in applications requiring high performance, or for handling hardware exceptions such as interrupts and processor start-up.

The ARM ISA is also supported in the **Cortex™-A** and **Cortex-R** profiles of the Cortex architecture for performance critical applications, and for legacy code. Most of its functionality is subsumed into the Thumb instruction set with the introduction of Thumb-2 technology. Thumb (**T32**) benefits from improved code density.

ARM instructions are 32-bits wide, and are aligned on 4-byte boundaries.

Most ARM instructions can be "conditionalized" to only execute when previous instructions have set a particular condition code. This means that instructions only have their normal effect on the programmers' model operation, memory and coprocessors if the N, Z, C and V flags in the Application Program Status Register satisfy a condition specified in the instruction. If the flags do not

satisfy this condition, the instruction acts as a NOP, that is, execution advances to the next instruction as normal, including any relevant checks for exceptions being taken, but has no other effect. This conditionalization of instructions allows small sections of if- and while-statements to be encoded without the use of branch instructions.

==>

**A64's Key differences from A32 are:**

- New instructions to support 64-bit operands. Most instructions can have 32-bit or 64-bit arguments
- Addresses assumed to be 64-bits in size. LP64 and LLP64 are the primary data models targeted
- Far fewer conditional instructions than in AArch32 conditional {branches, compares, selects}
- No arbitrary length load/store multiple instructions LD/ST 'P' for handling pairs of registers added A64

==> [64-bit data model](#)

## ARM Architecture

---

ARM 是在全球中被廣泛使用的 processor cores，像是 PDA、手機、多媒體播放器、數位電視、相機等等。ARM processors 有許多 family，如 ARM7、ARM9、ARM11、ARMv7，同一個 family 的設計使用相似的設計原則及同一個common instruction set。他的設計哲學是用有限的硬體( 有限的 memory 和 physical size restrictions) 資源達到 high code density。

### ARM 的命名方式 (Cortex系列以前) - ARMxyzTDMIEJFS

(在 ARM Cortex-A/R/M之前的 "ARM Classic")

- x: 處理器系列
- y: 記憶體管理單元 (MMU)
- z: cache
- T: 支援 Thumb 指令集
- D: 支援 debugger
- M: 支援快速乘法 ( Multiplier )
  - I: 支援 [Embedded ICE](#) (built-in debugger hardware)
- E: 支援增強型 DSP 指令 (Enhanced instruction)
- J: 支援 [Jazelle](#) (JVM)
- F: 具備向量浮點單元 VFP ( Floating-point)
- -S: Synthesizable version (source code version for EDA tools)

ps : TDMI 這四項基本功能成了任何新產品的標準配備，於是就不再使用這4個後綴。但是新的後綴不斷加入，包括定義存儲器界面的，定義高速快取的，以及定義"緊耦合存儲器 (TCM)"的，於是形成了新一套命名法，這套命名法一直使用至今。比如ARM1176JZF-S，它實際上預設就支持TDMI功能，除此之外還支持JZF。

這套命名機制在 ARM Cortex-A/R/M 之後，徹底棄置。以 MMU 來說，ARM Cortex-A 系列都有 MMU，而 Cortex-M0/M0+/M3/M4 均缺乏 MMU，僅有選擇性的 MPU。[Cortex-M7](#) 開始提供 cache 和

## TCM

## ARM Classic

(在 ARM Cortex 系列出現之前)

- ARM7TDMI (armv4)
  - 3 級 pipeline (fetch/decode/execute)
  - 高密度程式/低功耗
  - One of the most used ARM-version (for low-end systems)
  - 在 ARM7TDMI 之後版本的所有 ARM cores 都具備 TDMI
- ARM9TDMI (armv4)
  - 與 ARM7
  - 5 stages (fetch/decode/execute/memory/write)
  - Separate instruction and data cache
- ARM11
  - ARMv6 架構，是 Cortex-A 的基礎

## ARM 採用 RISC

ARM architecture 從 Berkeley RISC design 合併了幾個特點，但也有些並無採用。

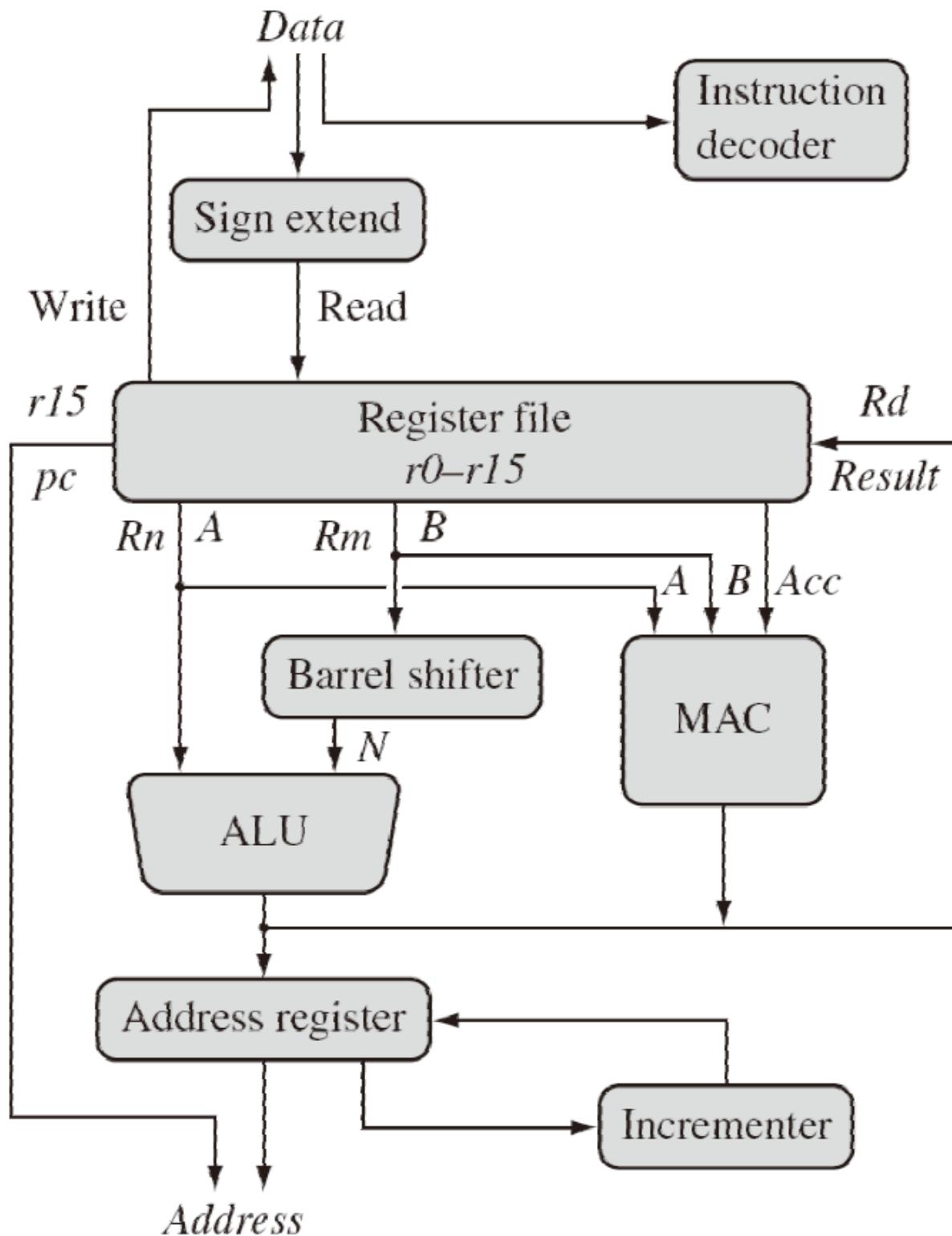
### Features used

- a load-store architecture
- 固定長度 32-bit instructions
- 3-address instruction 形式

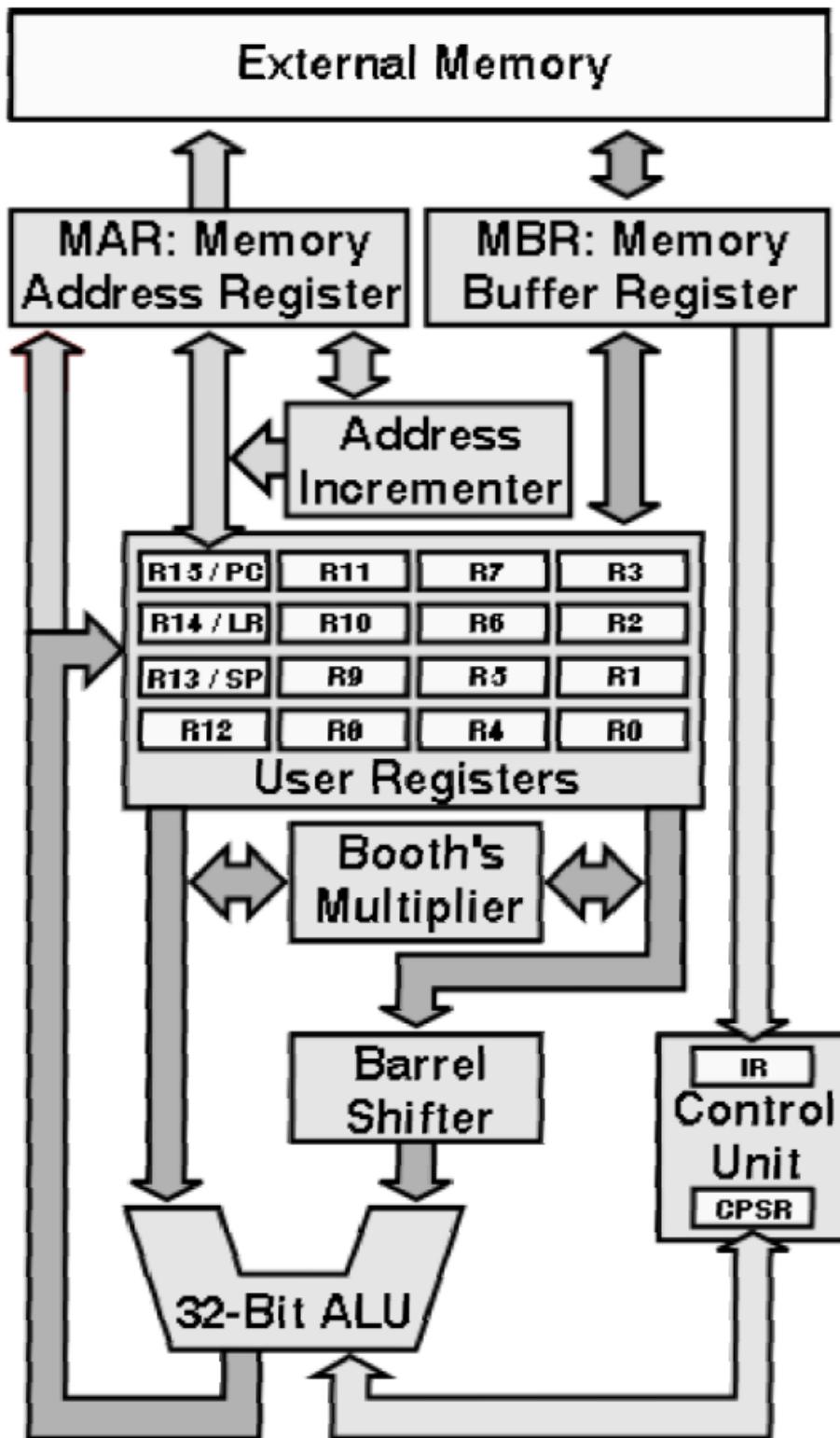
### Features rejected

- register window
  - 主要原因在於 register window 是一塊由許多暫存器所佔據的、很大的 chip 空間
- delayed branches
  - 主要原因在於 delayed branches remove the atomicity of individual instructions. 在 super-scalar 和 branch prediction mechanisms 無法好好相互配合
- single-cycle execution of all instructions
  - 還是有些指令需要超過 2 個或 2 個以上的 cycle 執行

## ARM 的架構



註: MAC = Multiply–accumulate operation



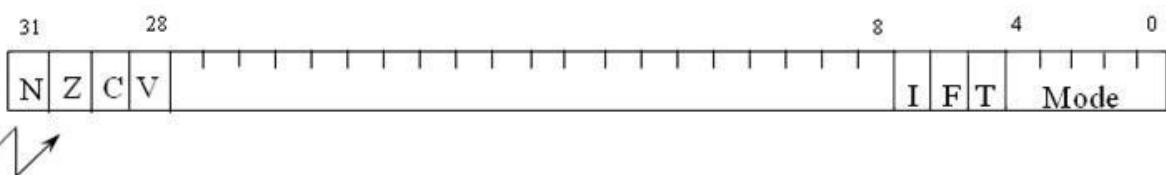
## ARM Register

以下是ARM的 register，在User/System Mode時，可以使用r0～r15，其中r13為stack pointer，r14為link register，r15為program counter

r0	
r1	
r2	
r3	
r4	
r5	
r6	
r7	
r8	r8_fiq
r9	r9_fiq
r10	r10_fiq
r11	r11_fiq
r12	r12_fiq
r13(sp)	r13_fiq
r14(lr)	r14_fiq
r15(pc)	
cpsr	
-----	
sprsr_fiq	r13_fiq
	r14_fiq
spsr_svc	r13_svc
	r14_svc
spsr_abt	r13_abt
	r14_abt
spsr_irq	r13_irq
	r14_irq
spsr_undef	r13_undef
	r14_undef

Current Program Status Register(CPSR) : 在user-level時，用於存取condition code bits

### 狀態暫存器 CPSR、SPSR 的結構



比較結果：如果指令中的 S bit 被設定，則會將 ALU 的狀態旗標放入 CPSR

\* 條件旗標

N = 負旗標

大旗標

◎ 次文化

V = 溢位旗標

\* 中斷控制位元

I = 1, 禁止中斷請求 IRQ.

F = 1, 禁止快速中斷 FIQ.

\* T位元

T = 0, 處理器處於 ARM 狀態

T = 1, 處理器處於 Thumb

\* 模式 (Mode)

M[4:0] 處理器的模式

溢位複習:

## 1、輸入的數是無符號整數，我們通過觀察C判斷是否溢出

a) C=1

i)如果是加法操作，結果不正確，結果溢出

ii)如果是減法操作，結果正確，結果未溢出

b) C=0

i)如果是加法操作，結果正確，結果未溢出

ii)如果是減法操作，結果不正確，結果未溢出。在這種情況下，結果是負數。然而，在無符號整數世界裡，負數不存在，我們認識這樣的操作是非法的。當然，如果認為答案是以有符號整數補碼的形式出現，則結果正確。

## 2、輸入的數是有符號整數，我們通過觀察V判斷是否溢出

a) V=1，結果不正確，結果溢出

b) V=0，結果正確，結果未溢出

通用暫存器有6種 data types ( signed / unsigned ) ( word / Half word / byte )，而在所有ARM的運算為32-bit，比較小的資料型態只有在資料傳送的運算中被支援。

Program Counter ( PC ) 是儲存要被執行的位址，而所有指令皆為 32-bit wide 且 word aligned

在 ARM 架構中支援了 ( 7+1 ) 種模式，如圖:

Processor mode	Description	
User	usr	Normal program execution mode
FIQ	fiq	Supports a high-speed data transfer or channel process
IRQ	irq	Used for general-purpose interrupt handling
Supervisor	svc	A protected mode for the operating system
Abort	abt	Implements virtual memory and/or memory protection
Undefined	und	Supports software emulation of hardware coprocessors
System	sys	Runs privileged operating system tasks

## Instruction sets

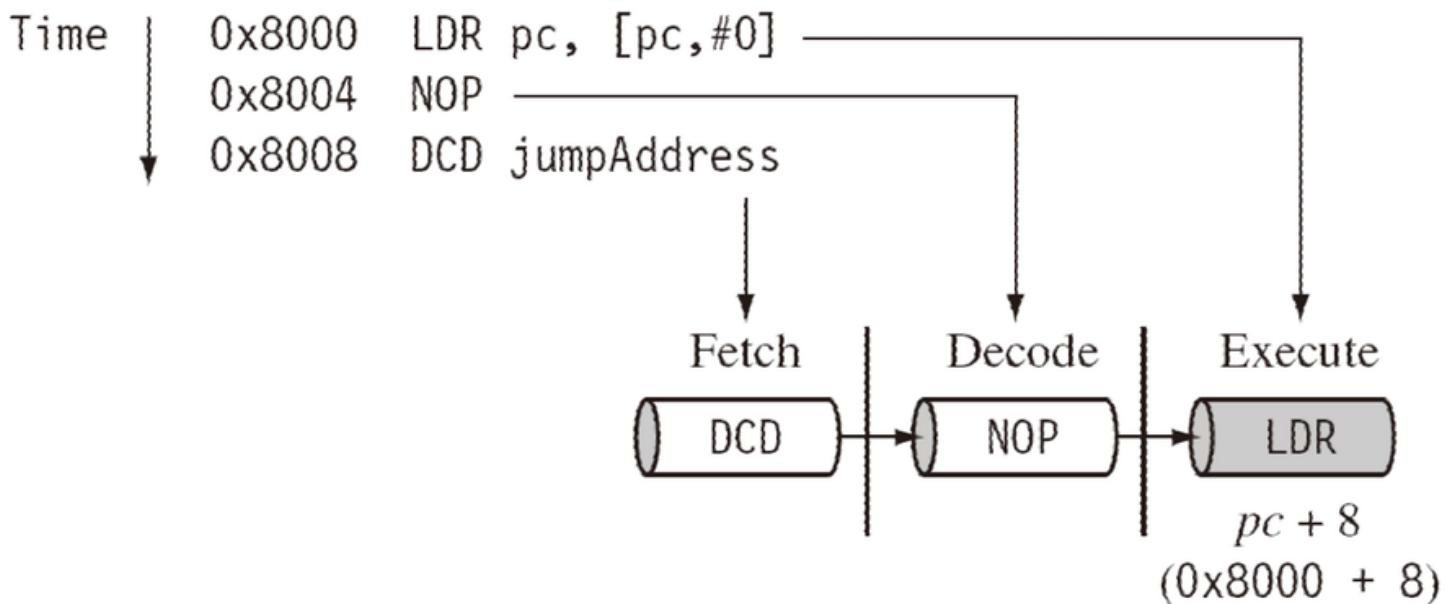
裡面有 ARM / Thumb / Jazelle，下圖為簡單介紹:

(實際上 ARM 的 extension 遠比以下列出的多)

	ARM ( <i>cpsr T = 0</i> )	Thumb ( <i>cpsr T = 1</i> )
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution <sup>a</sup>	most	only branch instructions
Data processing instructions	access to barrel shifter and ALU	separate barrel shifter and ALU instructions
Program status register	read-write in privileged mode	no direct access
Register usage	15 general-purpose registers +pc	8 general-purpose registers +7 high registers +pc
Jazelle ( <i>cpsr T = 0, J = 1</i> )		
Instruction size	8-bit	
Core instructions	Over 60% of the Java bytecodes are implemented in hardware; the rest of the codes are implemented in software.	

## Pipeline

在執行時，PC 是 8 bytes ahead，也就是說 Pipeline 準備要做 Execute 級(位址是 0x8000)時，要讀取的下一個位置是 PC + 8 的位置(從範例中可清楚看見，即 DCD 指令的位址 0x8008)

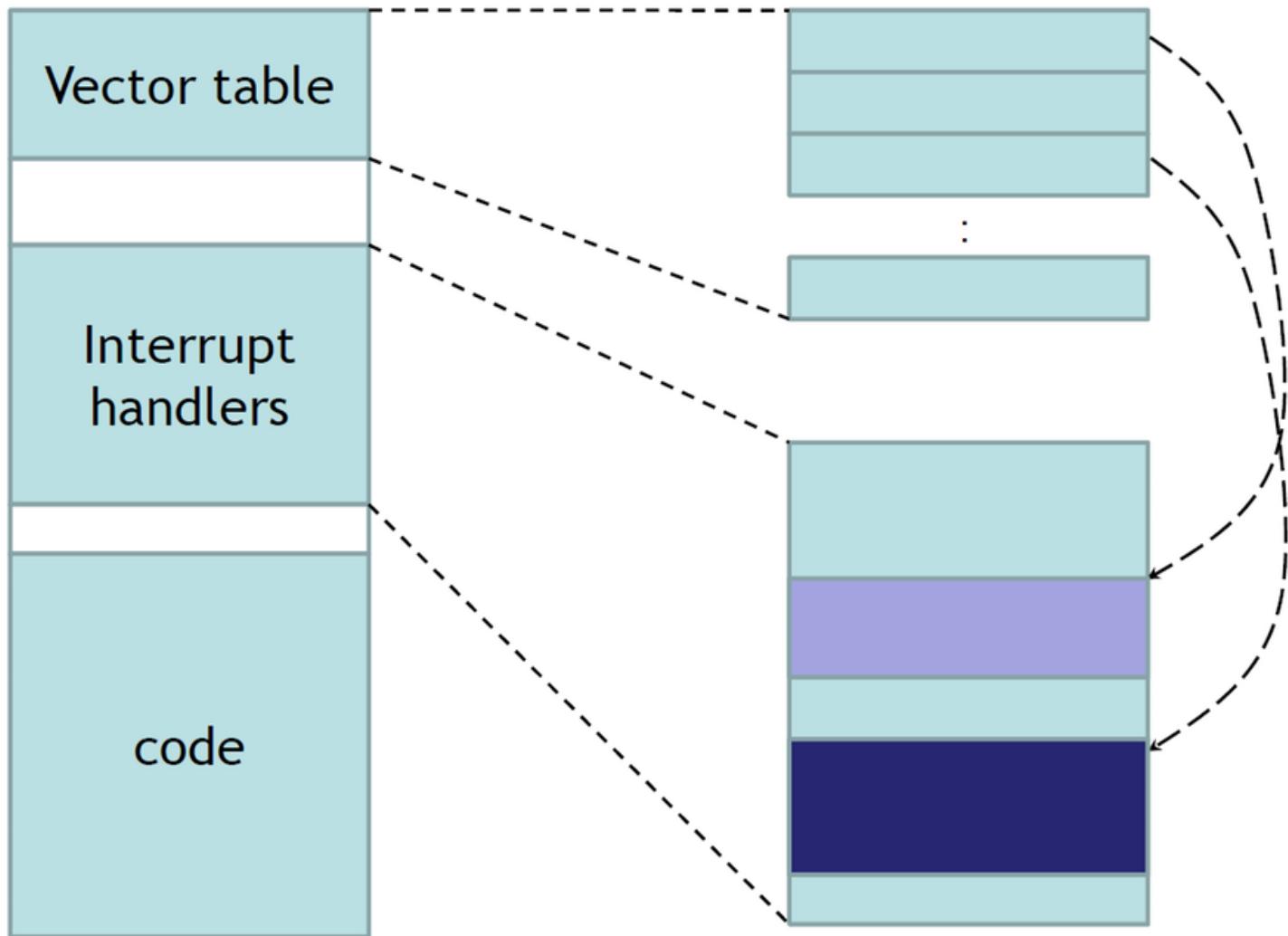


他有以下幾點特點：

- 當 Pipeline 在做 branch 或者直接修改 PC 值的話，會造成 ARM core flush 他的 pipeline
- ARM10 開始使用 branch prediction
- 即使發生中斷，也會將 Pipeline 中所有指令執行完才會去做中斷的事情

## Interrupts

當發生 exception 或 interrupt 時，會觸發 Interrupt handler，此時，他會尋找 vector table 去做中斷時所要處理的 routine.



下圖為中斷定義及其跳躍的起始位址:

Exception/interrupt	Shorthand	Address
Reset	RESET	0x00000000
Undefined instruction	UNDEF	0x00000004
Software interrupt	SWI	0x00000008
Prefetch abort	PABT	0x0000000c
Data abort	DABT	0x00000010
Reserved	—	0x00000014
Interrupt request	IRQ	0x00000018
Fast interrupt request	FIQ	0x0000001c

## □ 筆記 2 - ARM Instructions

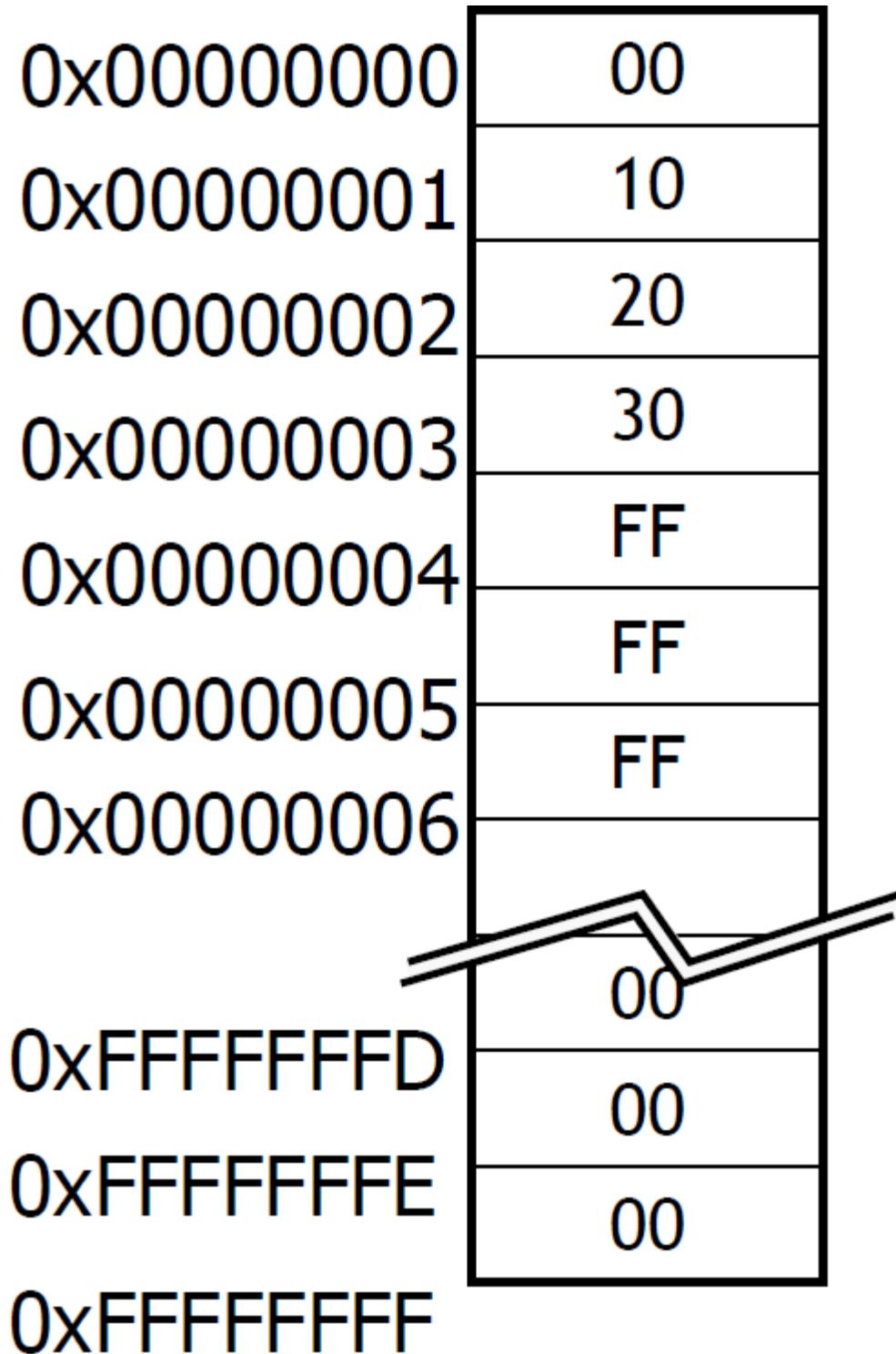
在 user-mode program 中，可以看到 15 個 32-bit general purpose registers (R0-R14), program

counter (PC) 及 CPSR，在指令集中有定義一些指令可以改變 state。

在開始深入探討前，先來看看他的 Memory system吧~

## Memory system

- memory 是一個 linear array of bytes addressed，範圍從 0 ~  $2^{32} - 1$
- 有 word, half-word, byte 三種形式
- 採取 Little Endian



剛剛有提到 Little Endian，他其實是 Byte ordering 的其中一種方式，endian 指的是當物理上的最小單元比邏輯上的最小單元還要更小時，邏輯單元對映到物理單元的排佈關係。舉例來說：如果你在文件上看到一個雙字組的data，Ex: `long MyData=0x12345678`，要寫到從0x0000開始的記憶體位址時。

## 1. 如果是Big Endian的系統：(TCP/IP)

存到記憶體會變成 0x12 0x34 0x56 0x78，最高位元組在位址最低位元，最低位元組在位址最高位元，依次排列。

## 2. 如果是Little Endian的系統：

存到記憶體會變成 0x78 0x56 0x34 0x12，最低位元組在最低位元，最高位元組在最高位元，反序排列。

比較的結果就是這樣：

	big-endian	little-endian
0x0000	0x12	0x78
0x0001	0x34	0x56
0x0002	0x56	0x34
0x0003	0x78	0x12

## Features of ARM instruction set

- Load-store architecture
- 3-address instructions
- 每個指令都可以做 Conditional execution
- 可以一次 load / store 多個暫存器
- 指令可以結合 shift 和 ALU operations (像是 add, sub)

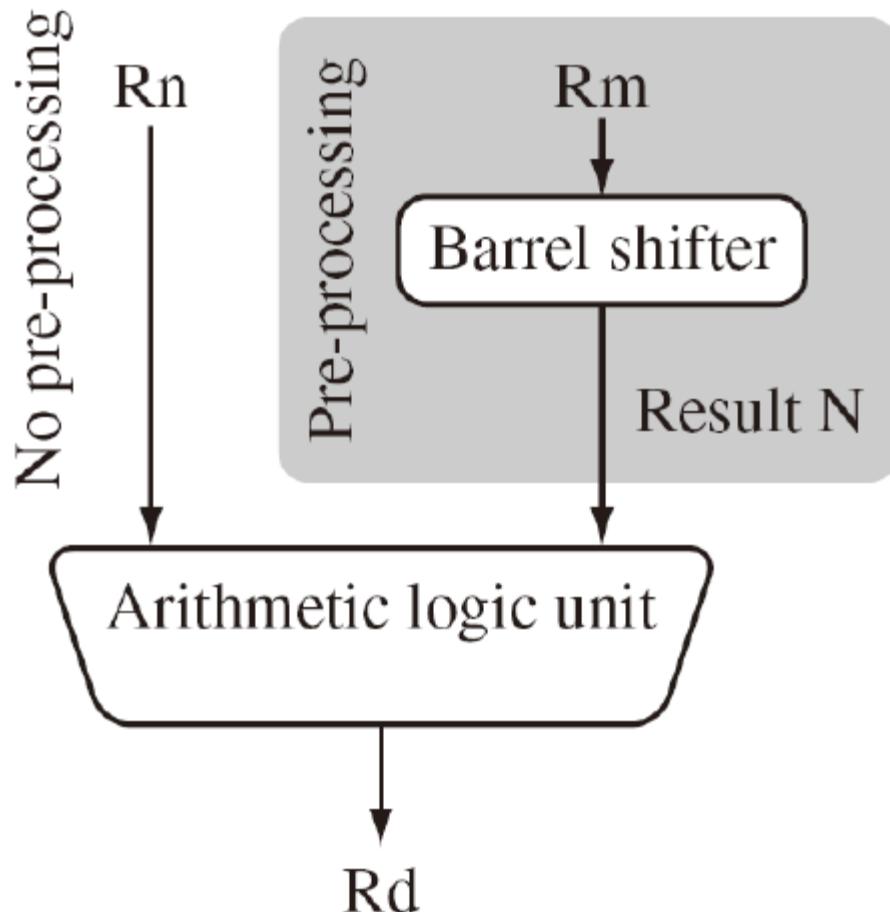
## Instruction set

可分成三大項：

- Data processing
  - move
  - arithmetic
  - logical
  - comparison
  - multiply
- Data movement (memory access)
- Flow control

## Data processing

資料處理指令包含對資料做 移動、算數、邏輯、比較、乘法 的指令，且大部分的 data processing instructions 可以對一個 operand 做 shift，如圖：



### Arithmetic operations:

- ADD : simple addition
- ADC : add with carry
- SUB : subtract
- SBC : subtract with carry
- RSB : reverse subtraction
- RSC : reverse subtraction with carry

ADD r0, r1, r2 ; r0 := r1 + r2

ADC r0, r1, r2 ; r0 := r1 + r2 + C

SUB r0, r1, r2 ; r0 := r1 - r2

SBC r0, r1, r2 ; r0 := r1 - r2 + C - 1

RSB r0, r1, r2 ; r0 := r2 - r1

RSC r0, r1, r2 ; r0 := r2 - r1 + C - 1

### Bit-wise logical operations:

- BIC : bit clear

AND r0, r1, r2 ; r0 := r1 and r2

ORR r0, r1, r2 ; r0 := r1 or r2

EOR r0, r1, r2 ; r0 := r1 xor r2

BIC r0, r1, r2 ; r0 := r1 and not r2

## Register movement operations:

```
MOV r0, r2      ; r0 := r2
MVN r0, r2      ; r0 := not r2
```

## Comparison operations:

只設定在CPSR的 condition code bits(N, Z, C and V)

- CMP : compare
- CMN : compare negated
- TST : (bit) test
- TEQ : test equal

```
CMP r1, r2      ; set CC on r1 - r2
CMN r1, r2      ; set CC on r1 + r2
TST r1, r2      ; set CC on r1 and r2
TEQ r1, r2      ; set CC on r1 xor r2
```

許元杰 這裡的CC代表CPSR的 condition code bits

## Immediate operands:

```
ADD r3, r3, #1  ; r3 := r3 + 1
AND r8, r7, #&ff ; r8 := r7[7:0]
```

許元杰 一般在#後放置數字，則表示10進位數字；而在#後放置&符號，可表示16進位數字

YANGBO L As indicated in lec09 [p.14], the & sign is assembler dependent. I wonder what it should be for GCC?

NEO J  $r8 := r7 \& 0x000000FF$

- 載入 32-bit 常數，該怎麼作？

- 如果要在 r0 暫存器中，存放 0xDEADBEEF ([Hексpeak](#))，最直接想到這樣作  
`mov r0, 0xDEADBEEF`
- 技術上來說，這是不可能的！為何？ARMv7 (含) 之前，所有的指令都是 32 bit (opcode + operand)，以 mov 來說，所有的 32 bit 自然包含 mov 指令本身、目標 register，以及目標值。因此，不可能將任意的 32-bit 編碼的值存放到 32 bit 指令中
- MIPS 的作法: [Loading a 32 bit Immediate](#)
- 直接表示值對 ARM 來說，實際是一個 8-bit 數值和一個旋轉因子，所以可表達以下 32-bit 直接數值
- 但 0x1FF 不是 8-bit 數值，而 0xF000F000 也不可，因為無法經由旋轉而變成 8-bit 數值
- 所以我們得這樣作

load\_32bit:

```
ldr r0, [pc #0] ;; 請注意: pc 位於目前位址向前 8 bytes 的地方
bx lr
.word 0xDEADBEEF
```

- 由於已經知道資料的位址，可透過 ldr 指令告訴 ARM 去取數值並且載入到暫存器。在 ARM

pipeline , PC 總是位於相對於目前指令再提前兩道指令的位址

- 在 ARMv7 後，引入兩個步驟的指令來載入數值: movw, movt

movw r0, #0xbeef ; r0 = 0x0000beef

movt r0, #0xdead ; r0 = deadbeef

- GNU as 純粹便利的寫法

```
.equ label, 0xDEADBEEF
```

```
movw r0, #:lower16:label
```

```
movt r0, #:upper16:label
```

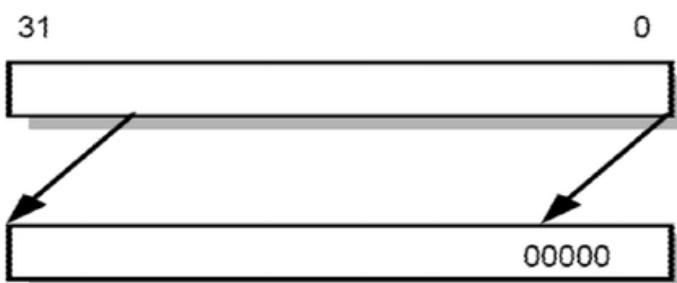
Reference: <https://sourceware.org/ml/binutils/2014-02/msg00157.html>

## Shifted register operands:

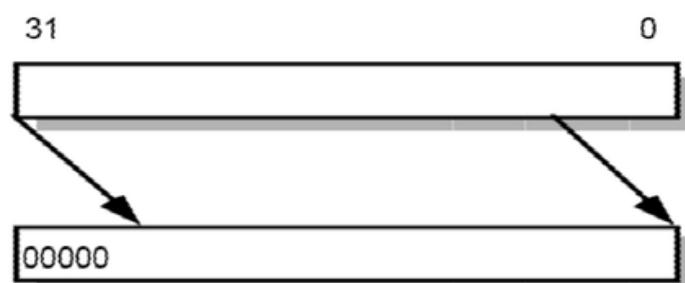
舉例來說:

```
ADD r3, r2, r1, LSL #3 ; r3 := r2 + ( r1 << 3 )
```

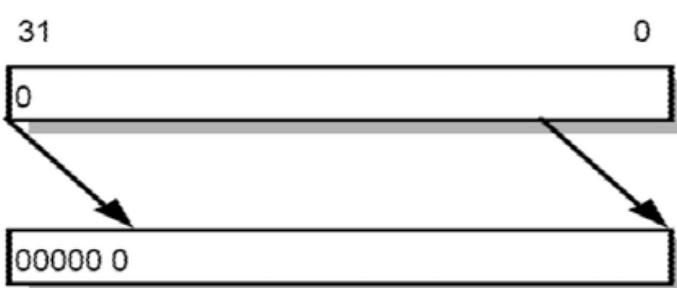
- LSL : logical shift left by 0 to 31 places; fill the vacated bits at the least significant end of the word with zeros
- LSR : logical shift right by 0 to 32 places; fill the vacated bits at the most significant end of the word with zeros
- ASL : arithmetic shift left; this is a synonym for LSL
- ASR : arithmetic shift right by 0 to 32 places; Register contents are treated as two's complement signed integers.; fill the vacated bits at the most significant end of the word with the sign bit
- ROR : rotate right. Provides the value of the contents of a register rotated by a value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.
- RRX : provides the value of the contents of a register shifted right one bit. The old carry flag is shifted into bit[31]. If the S suffix is present, the old bit[0] is placed in the carry flag.



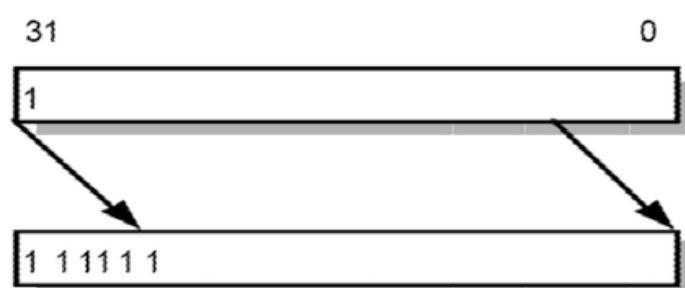
LSL #5



LSR #5



ASR #5 , positive operand



ASR #5 , negative operand

位移 (shift) 主要分兩種：

- [https://en.wikipedia.org/wiki/Logical\\_shift](https://en.wikipedia.org/wiki/Logical_shift) (邏輯移位)
- [https://en.wikipedia.org/wiki/Arithmetic\\_shift](https://en.wikipedia.org/wiki/Arithmetic_shift) (算數移位)

簡單來說，邏輯移位無論左移右移，一律都是把多出來的位數填零。算術運算左移跟邏輯移位一樣填零，右移需要考慮到 signed 的屬性，假若最高位是 1，則填補1，反之若最高位是 0，則填補0。

範例：

```
signed int a = 1234;
signed int b = 6;
signed int r = a >> b; /* 邏輯平移運算 */
```

```
unsigned int a = 1234 ;
int b = 6 ;
unsigned int r = a >> 6 ; /* 算數平移運算 */
```

注意: ARM Toolchain (包含 arm-none-eabi/linux-gnueabi) 預設將 "int" 視為 "unsigned"

- mov r0, #8000000F
  - 將數值 `0x8000000F` 放入暫存器 r0 中  
 $\langle r0 \rangle | 1\ 0\ 0\ 0 | 0\ 0\ 0\ 0 | 0\ 0\ 0\ 0 | 0\ 0\ 0\ 0 | 0\ 0\ 0\ 0 | 0\ 0\ 0\ 0 | 0\ 0\ 0\ 0 | 1\ 1\ 1\ 1 |$
- mov r1, r0, LSL #1
  - r0 左移 1 bit 後，將值放入 r1 中  
 $\langle r1 \rangle | 0\ 0\ 0\ 0 | 0\ 0\ 0\ 0 | 0\ 0\ 0\ 0 | 0\ 0\ 0\ 0 | 0\ 0\ 0\ 0 | 0\ 0\ 0\ 0 | 0\ 0\ 0\ 1 | 1\ 1\ 1\ 0 |$
  - 向左位移後，讀取 r0 的值，經過 barrel shifter 後，數值變成 0x1E
  - bit 31 向左位移後，超過 32 bit 範圍，會被捨棄，而 bit 4, 3, 2, 1 分別位移到 bit 5, 4, 3, 2
- `mov r1, r0, lsl #2` 等價於 `r1 = (int)(r0 << 2)`
  - barrel shift

CHIA-CHI H 提升指令密度

## Condition flags

If S is specified, these instructions update the condition code bits

□ 許元杰 Logical or move operation 不更新 C or V, shift operation (除RRX外)不更新 C

Ex:

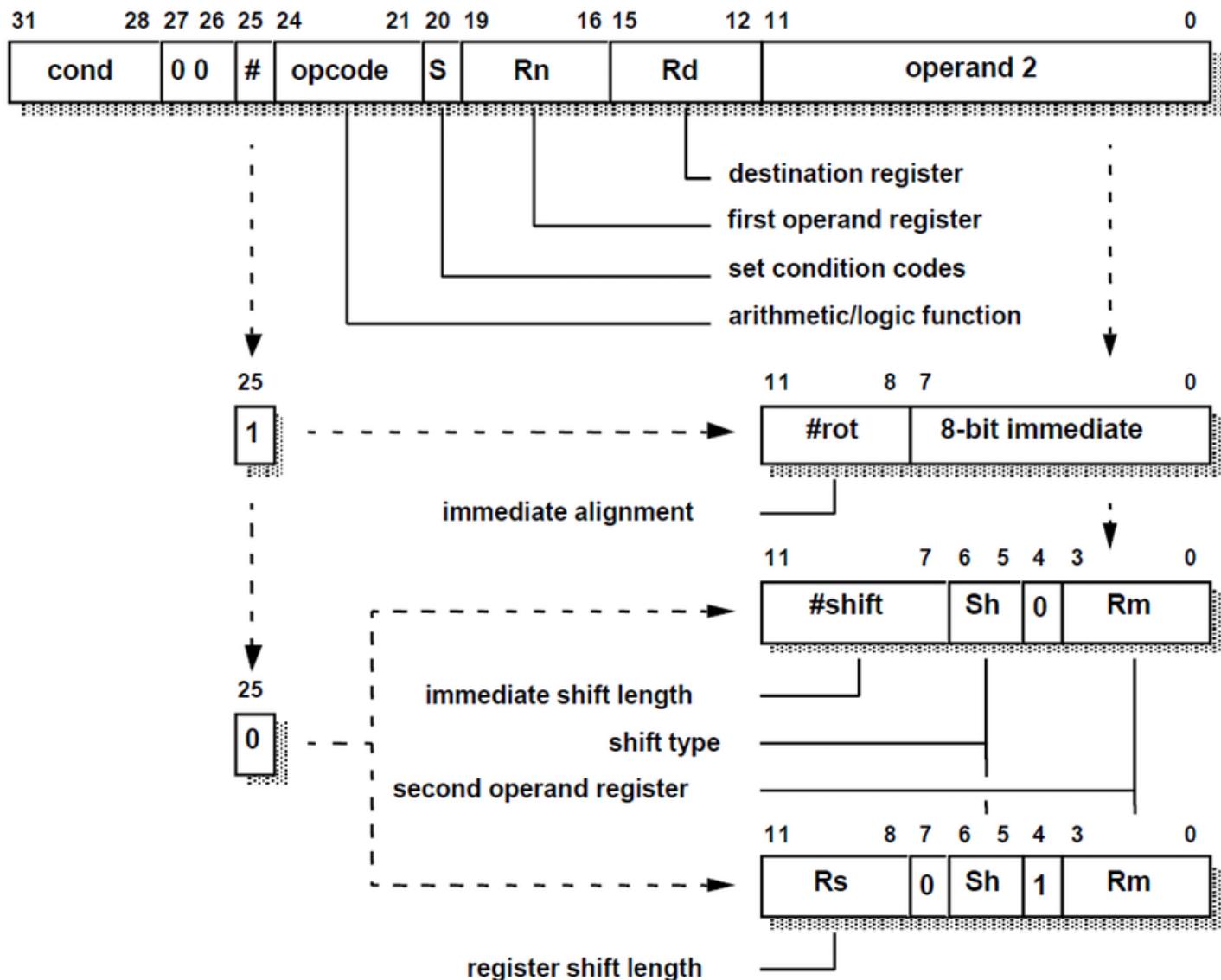
```
ADDS r2, r2, r0 ; 32-bit carry out -> C ..
```

## Multiples:

```
MUL r4, r3, r2 ; r4 := (r3 * r2)[31:0]
```

## Encoding data processing instructions

這裡可以看第 25-bit(#[#]) 決定 operand 2 是以何種格式，如果 # = 1 就如圖下只有 1 種對應格式，如果 # = 0 就需要再比對位置在第四個 bit ( 決定 2 種格式 )



## Flow Control

決定哪一條指令將被執行

- B{<conf>} label
- BL{<conf>} label
- BX{<conf>} Rm
- BLX{<conf>} label | Rm

B	branch	pc = label pc-relative offset within 32MB
BL	branch with link	pc = label
BX	branch exchange	pc = Rm & 0xffffffff, T = Rm & 1
BLX	branch exchange with link	pc = label, T = 1

Mnemonic	Name	Condition flags
EQ	equal	Z
NE	not equal	z
CS HS	carry set/unsigned higher or the same	C
CC LO	carry clear/unsigned lower	c
MI	minus/negative	N
PL	plus/positive or zero	n
VS	overflow	V
VC	no overflow	v
HI	unsigned higher	zC
LS	unsigned lower or same	Z or c
GE	signed greater than or equal	NV or nv
LT	signed less than	Nv or nV
GT	signed greater than	NzV or nzv
LE	signed less than or equal	Z or Nv or nV
AL	always (unconditional)	ignored

Branch	Interpretation	Normal uses
B BAL	Unconditional	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or zero
BLO	Lower	Unsigned comparison gave lower
BHS	or the same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater than or equal	Signed integer comparison gave greater than or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or the same	Unsigned comparison gave lower or same

## Data transfer instructions

data processing 的 mov 指令是暫存器間互相傳送資料，而 data transfer instructions 是暫存器與 memory 間互相傳遞資料，而 data transfer instructions 有三種基本形式

1. Single register load/store , 也就是對單一暫存器做 load/store
2. Multiple register load/store , 可以對多個暫存器做 load/store
3. Single register swap: SWP(B), atomic instruction for semaphore

### Syntax:

- <LDR|STR>{<cond>} {B} Rd, addressing1
- LDR{<cond>}SB|H|SH Rd, addressing2
- STR{<cond>}H Rd, addressing2

LDR	load word into a register	Rd <- mem32[address]
STR	save byte or word from a register	Rd --> mem32[address]
LDRB	load byte into a register	Rd <- mem8[address]
STRB	save byte from a register	Rd --> mem8[address]
LDRH	load halfword into a register	Rd <- mem16[address]
STRH	save halfword into a register	Rd --> mem16[address]
LDRSB	load signed byte into a register	Rd <- SignExtend
LDRSH	load signed halfword into a register	Rd <- SignExtend

ps. 沒有 STRSB/STRSH 是因為 STRB/STRH 儲存 signed/unsigned 到記憶體位置(存進去就不管他是有號無號，讀出來才要判別)

Memory 定址可以透過暫存器和 offset，例：

LDR R0, [R1] @ mem[R1]

// 3 ways to specify offsets:

// 1. Immediate

LDR R0, [R1, #4] @ mem[R1+4]

// 2. Register

LDR R0, [R1, R2] @ mem[R1 + R2]

// 3. Scaled register

LDR R0, [R1, R2, LSL #2] @ mem[ R1 + 4 \* R2 ]

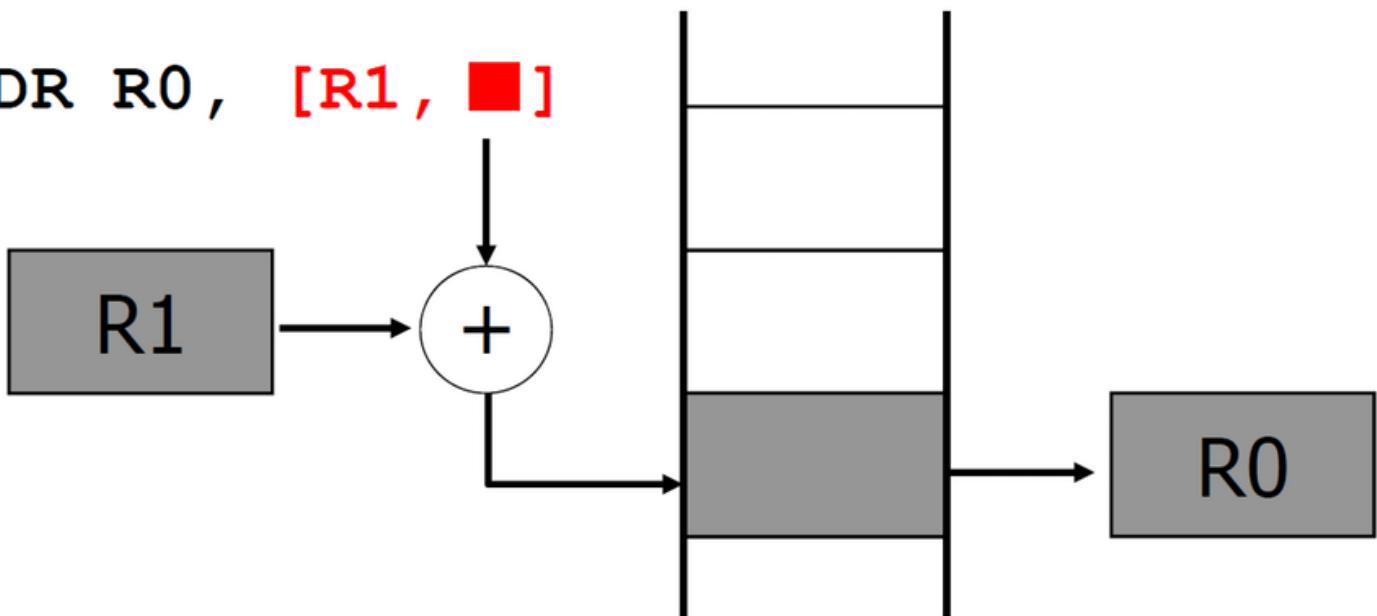
他有三種 Addressing modes

Index method	Data	Base address register	Example
Preindex with writeback	mem[ base + offset ]	base + offset	LDR r0,[r1, #4]!
Preindex	mem[ base + offset ]	not updated	LDR r0, [r1, #4]
Postindex	mem[ base ]	base + offset	LDR r0, [r1], #4

## 圖解 Addressing modes

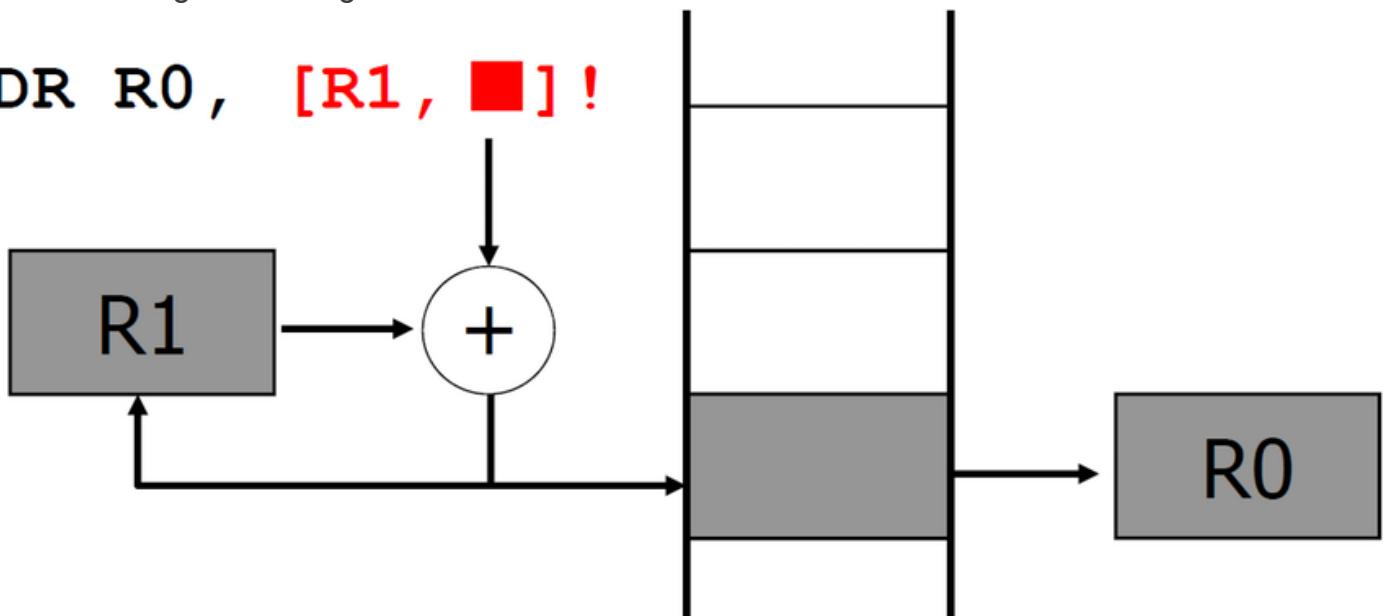
- Pre-index addressing

**LDR R0, [R1, ■]**



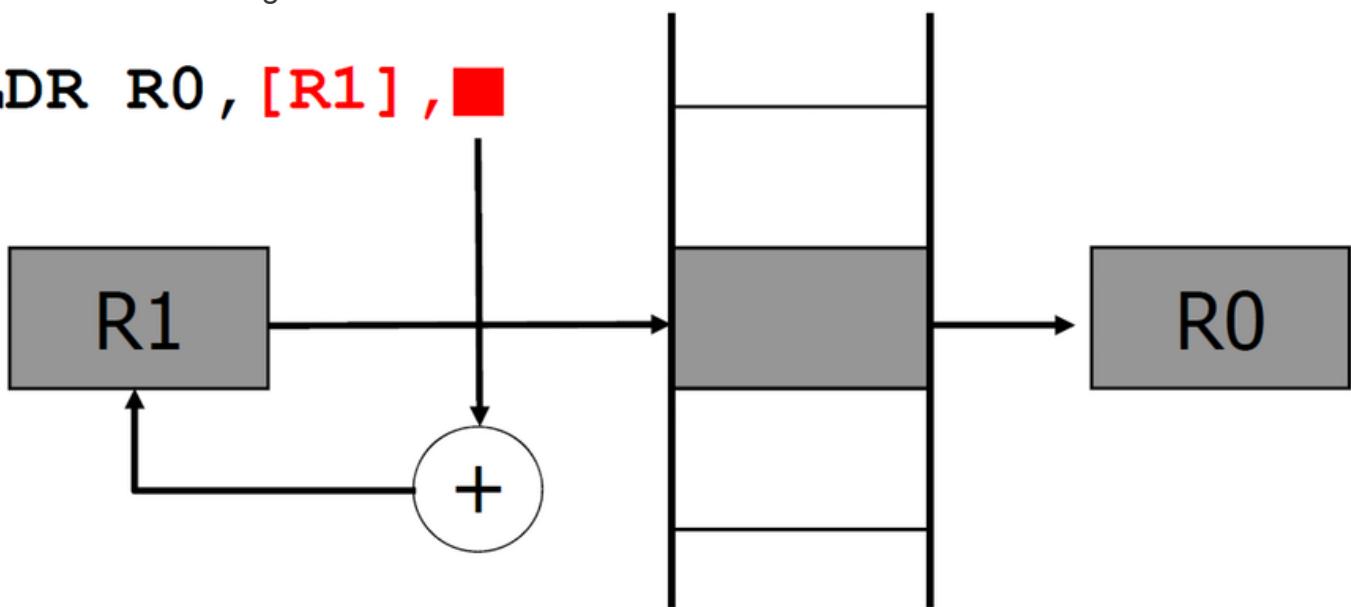
- Auto-indexing addressing

**LDR R0, [R1, ■]!**



- Post-index addressing

**LDR R0, [R1], ■**



以上是有 offset 的三種 addressing modes，接下來來看看 Register 的 addressing modes

- Pre-indexed addressing

LDR R0, [R1, R2] @ R0 = mem[ R1 + R2 ]

@ R1 unchanged

- Auto-indexed addressing

LDR R0, [R1, R2]! @ R0 = mem[ R1 + R2 ]

@ R1=R1+R2

- Post-indexed addressing

LDR R0, [R1], R2 @ R0=mem[ R1 ]

@ R1 = R1 + R2

ps.有一個 pseudo instruction **ADR** 可以 load and address 到一個暫存器裡

□ JIM H 為何要有 pseudo-instruction 呢？

-> <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204j/Babbfdih.html>

Google:// unified syntax arm

Google:// interworking arm

## Multiple register load/store

意思就是一次可以存取很多個暫存器的值

簡單的範例:

LDM R0, {R1-R3}

// R0 := R1

// R0 + 4 := R2

// R0 + 8 := R3

### Syntax:

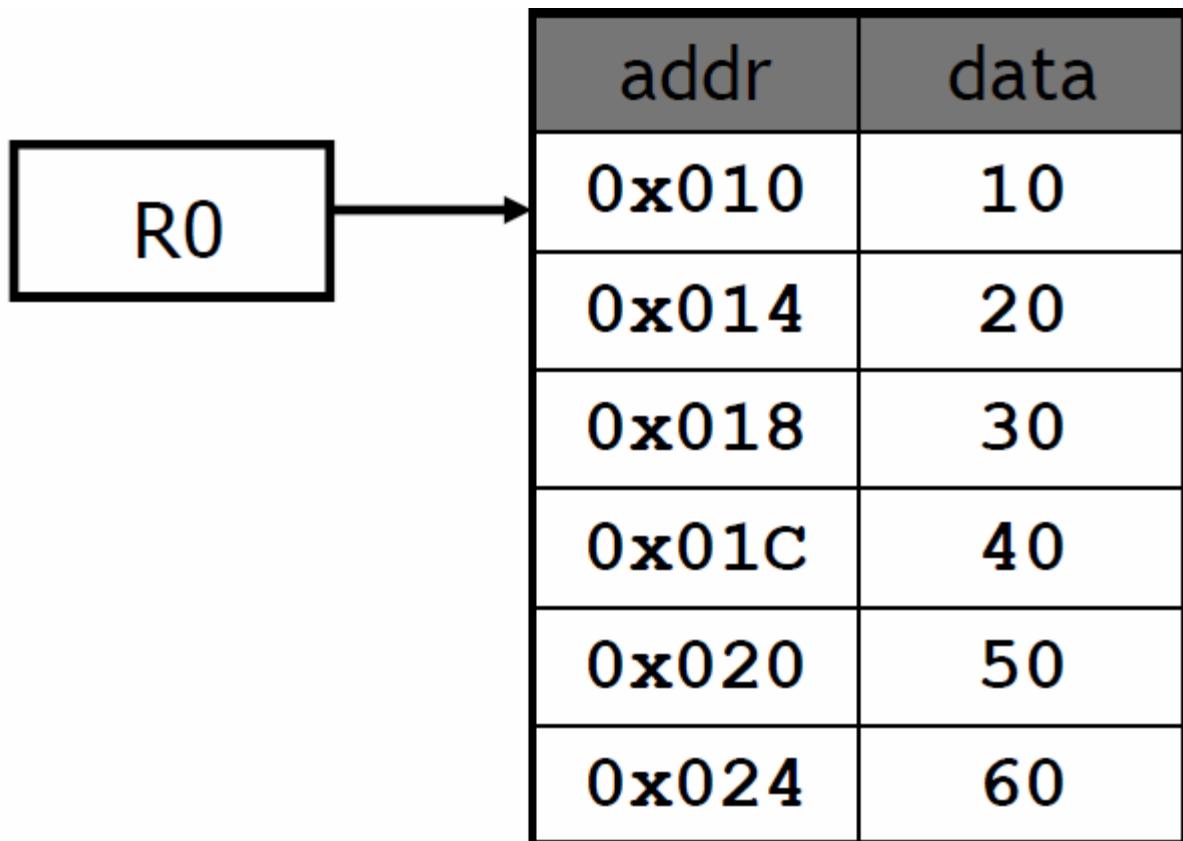
- <LDM|STM>{<conf>}<addressing mode> Rn{!},<registers>{^}
- 驚嘆號(!)代表要存取後會 register 進行更新(後面有範例')

Addressing mode	Description	Start address	End address	Rn!
IA	increase after	Rn	Rn + 4*N - 4	Rn + 4*N
IB	increase before	Rn + 4	Rn + 4*N	Rn + 4*N
DA	decrease after	Rn - 4*N +4	Rn	Rn - 4*N
DB	decrease before	Rn -4*N	Rn - 4	Rn - 4*N

用上表有點難看出變化，以下用圖來表示

- LDMIA R0!, {R1, R2, R3} == LDMIA R0!, {R1-R3}

其中 R1 = 10, R2 = 20, R3 = 30, 更新後的 R0 = 0x01c

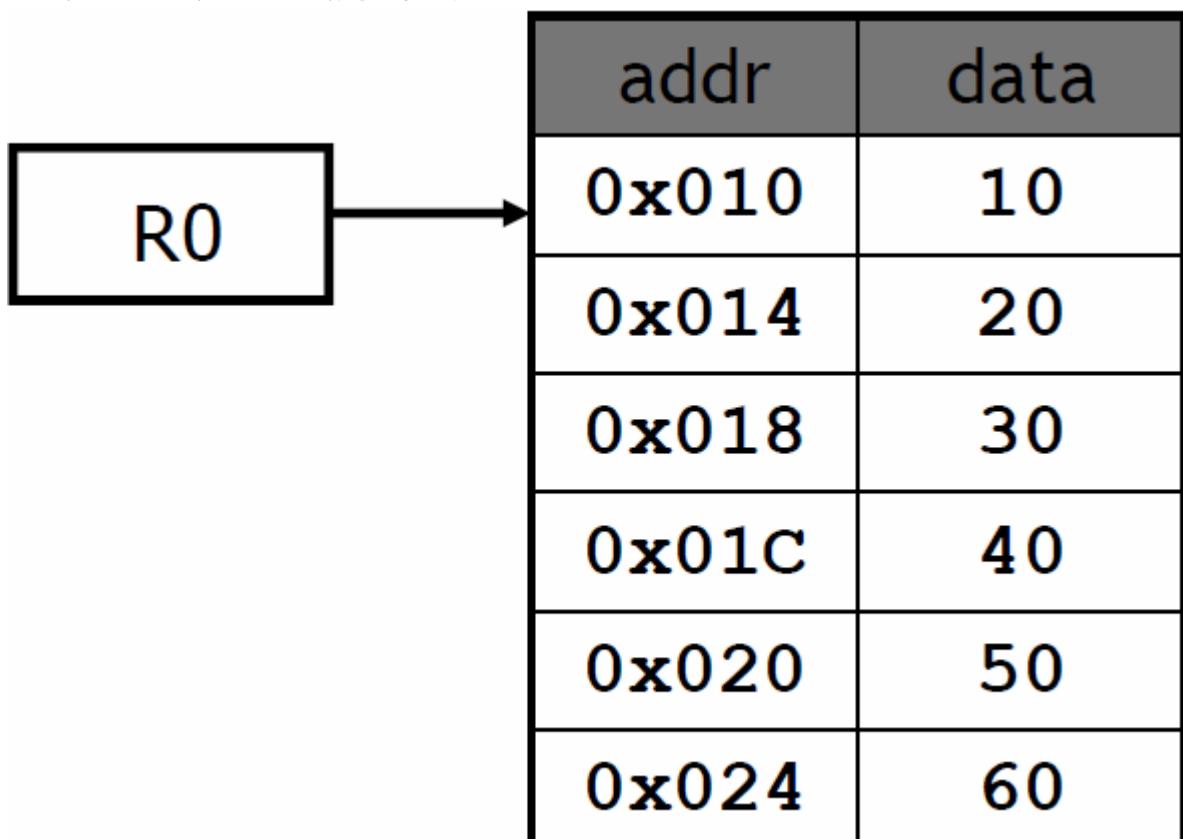


從這裡可以看到依序從最低位開始存，存完才將  $R0 + 4$ ，最後，當存完 R3 的值之後，R0 更新成 0x01c，因為上面指令有 ！」，所以將最後值存在 R0 中

ps.: 如果沒有！」，則最後  $R0 = 0x010$ ，後面的範例也是如此

- LDMIB R0!, {R1, R2, R3} == LDMIB R0!, {R1-R3}

其中  $R1 = 20, R2 = 30, R3 = 40$ , 更新後的  $R0 = 0x01c$

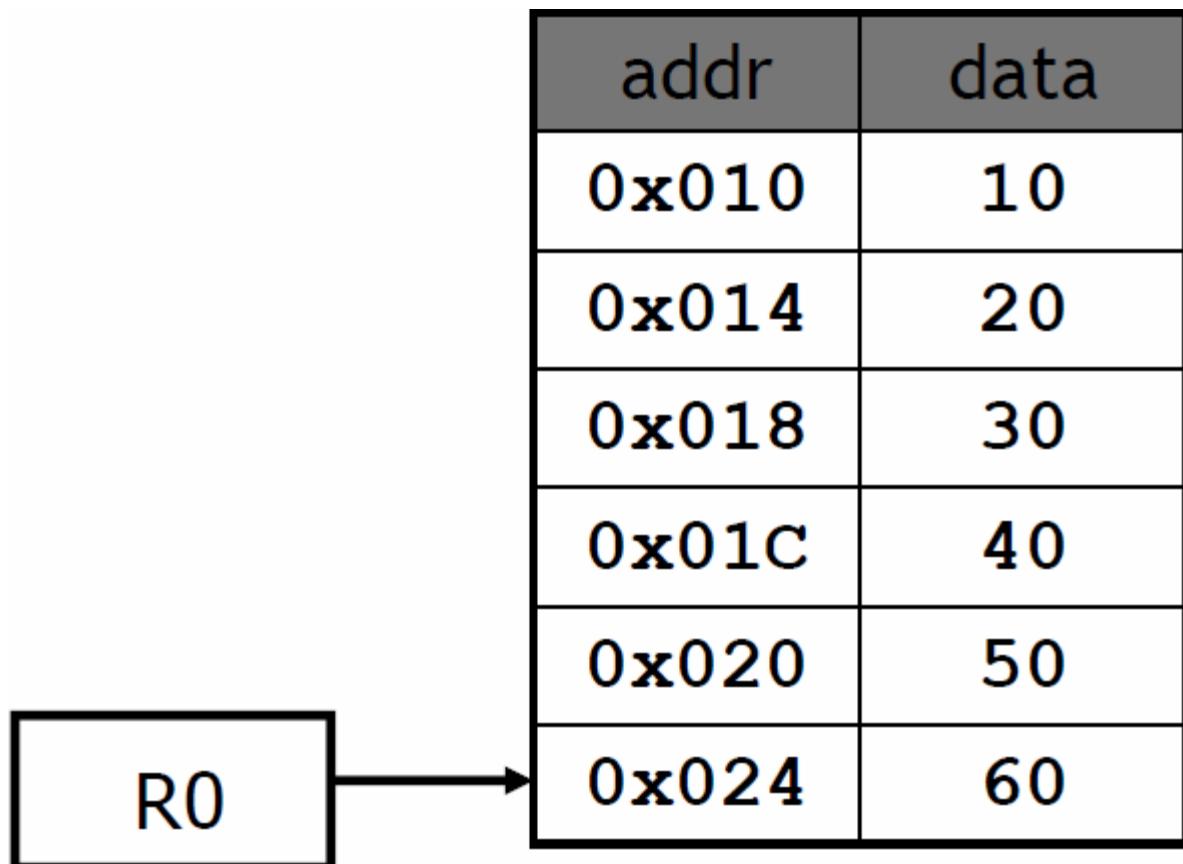


從這裡可以看到先將  $R0 + 4$  後才開始存，然後依序往上加，最後，當 R0 更新成 0x01c 也存完 R3 的值

之後，因為上面指令有 ！」，所以將最後值存在 R0中

- LDMDA R0!, {R1, R2, R3} == LDMDA R0!, {R1-R3}

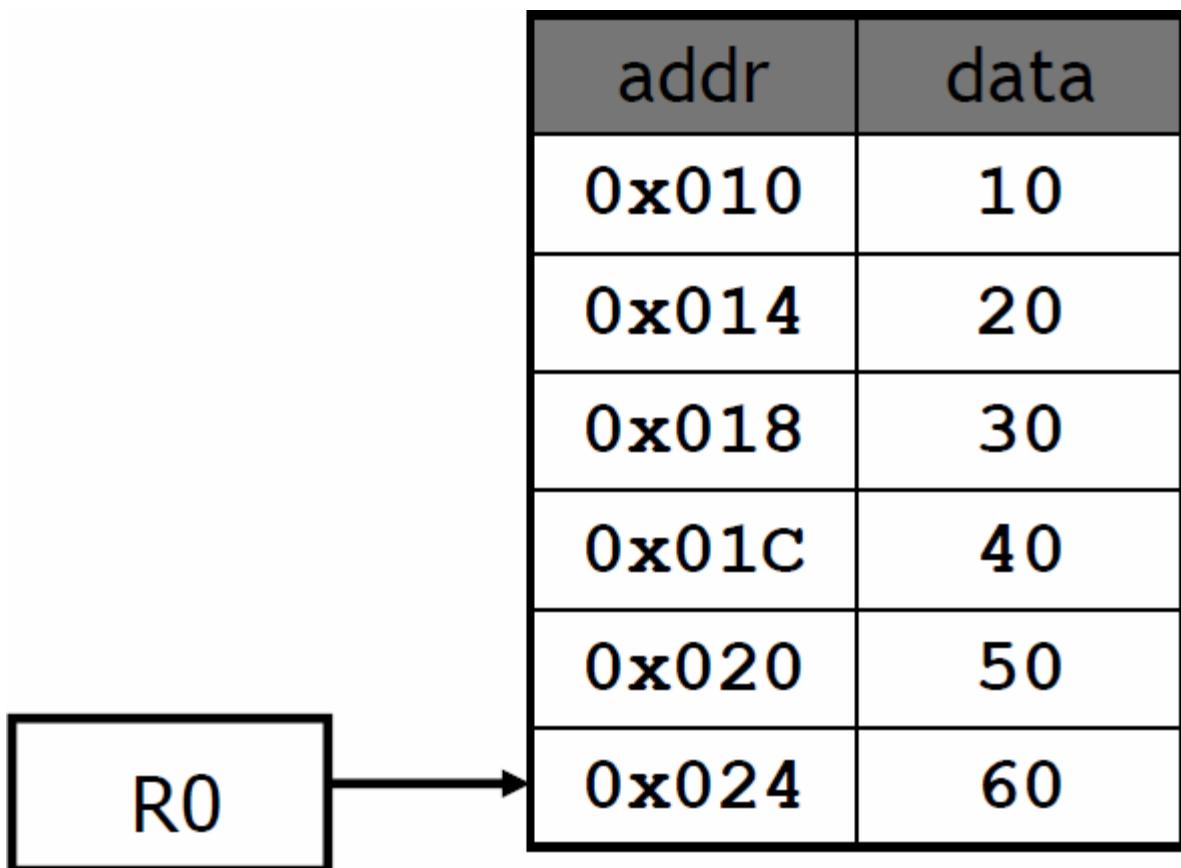
其中 R1 = 40, R2 = 50, R3 = 60, 更新後的 R0 = 0x018



從這裡可以看到依序從最高位開始存，存完才將 R0 - 4，最後，當存完 R1 的值之後，R0 更新成 0x018，因為上面指令有 ！」，所以將最後值存在 R0 中

- LDMDB R0!, {R1, R2, R3} == LDMDB R0!, {R1-R3}

其中 R1 = 30, R2 = 40, R3 = 50, 更新後的 R0 = 0x018



從這裡可以看到先將R0 - 4 後才開始存，最後，當R0更新成 0x018也存完R1的值之後，因為上面指令有 !r，所以將最後值存在 R0中

### Multiple load/store registers 的應用

- 可用來複製一塊 block 的 memoryj3
  - R9: address of the source
  - R10: address of the destination
  - R11: end address of the source

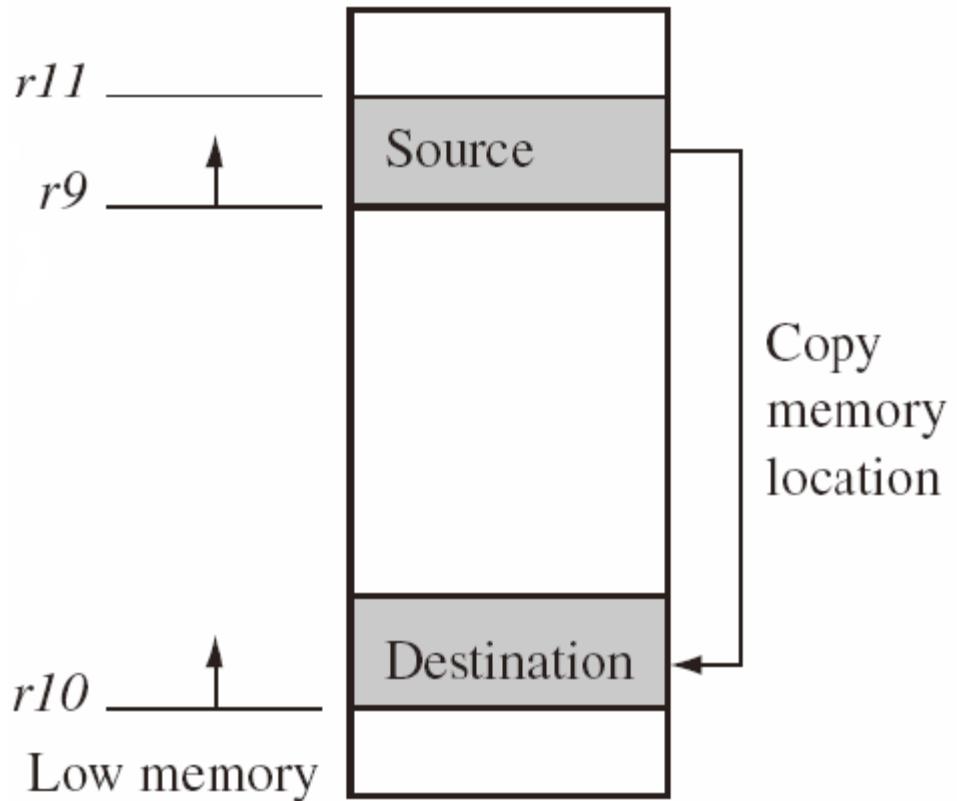
loop: LDMIA R9!, {R0-R7}

STMIA R10!, {R0-R7}

CMP R9, R11

BNE loop

## High memory



- Stack
  - full: pointing to the last used
  - ascending: grow towards increasing memory addresses)

mode	POP	=LDM	PUSH	=STM
Full ascending (FA)	LDMFA	LDMDA	STMFA	STMIB
Full descending (FD)	LDMFD	LDMIA	STMD	STMDB
Empty ascending (EA)	LDMEA	LDMDB	STMEA	STMIA
Empty descending (ED)	LDMED	LDMIB	STMED	STMDA

## PRE Address Data

*sp* →

0x80018	0x00000001
0x80014	0x00000002
0x80010	<i>Empty</i>
0x8000c	<i>Empty</i>

STMFD sp!, {r1, r4}

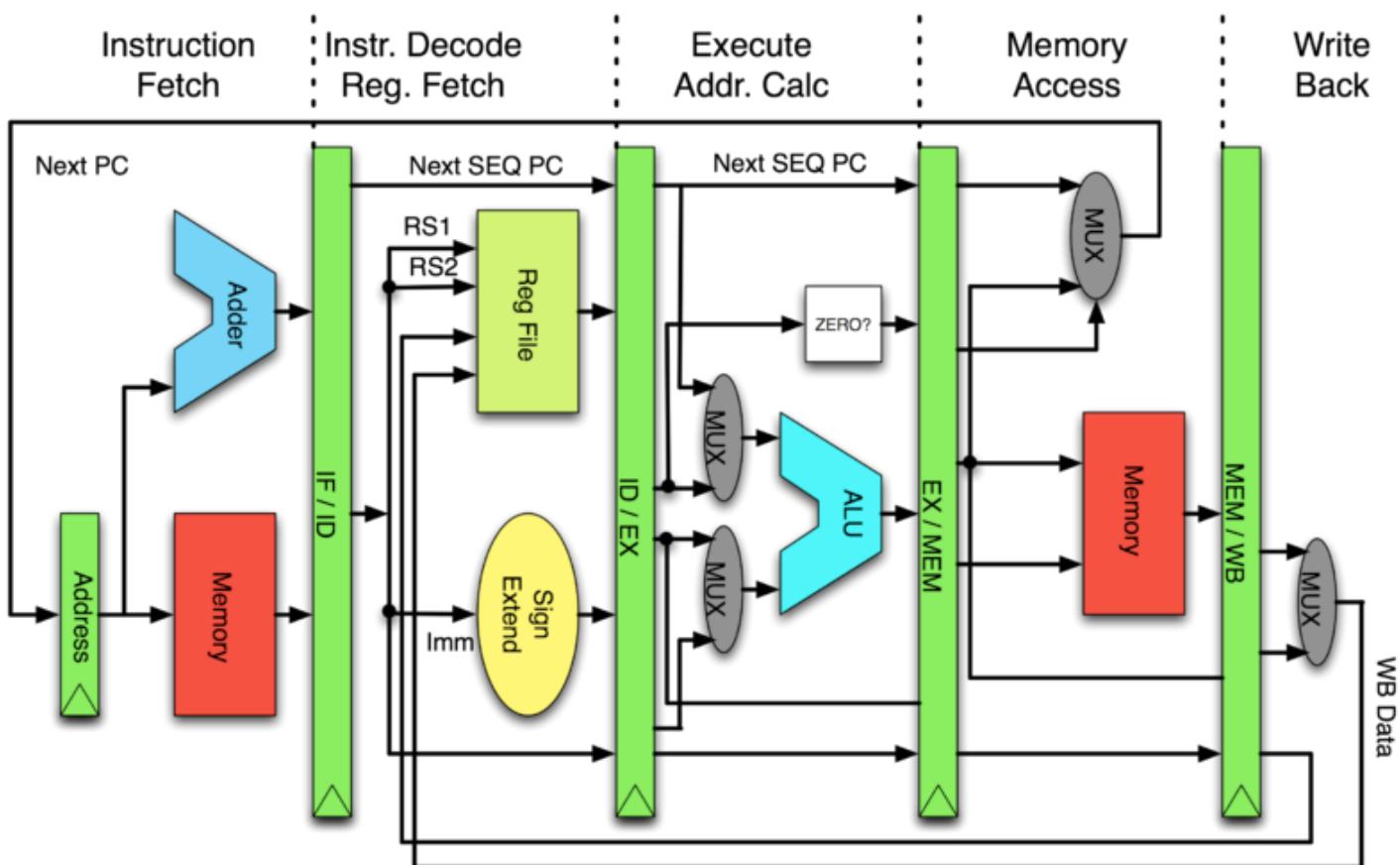
## POST Address Data

*sp* →

0x80018	0x00000001
0x80014	0x00000002
0x80010	0x00000003
0x8000c	0x00000002

## ARM 架構

- 為何ARM的 `PC` 是指向 下兩條指令？
    - pipeline stages: 從 EXE stage 往回看 FETCH stage : 正在 fetch 下兩條指令
- 參考: 舊的 MIPS 架構: fetch 之後,  $PC = PC + 4$ , 且一路往下傳遞到接下來的 stages



[https://commons.wikimedia.org/wiki/File:Pipeline\\_MIPS.png](https://commons.wikimedia.org/wiki/File:Pipeline_MIPS.png)

- 如上面所說 PC 指向正在執行的後 2 條指令 (+8)，但若某道正在執行指令遇到中斷，這時候的 PC 會如何變化？
  - 這個問題要分兩個面向來答覆，一個是 interrupt 的處理機制，另一個則是 PC 的計算方式
  - 引述《[The Definitive Guide to the ARM Cortex-M3](#)》Page 287:
 

"Interrupt handler and interrupt return: In the ARM7, the first instruction of the interrupt handler is in the vector table, which normally contains a branch instruction to the actual interrupt handler. In the Cortex-M3, this step is no longer needed. For interrupt returns, the ARM7 relies on manual adjustment of the return program counter. In the Cortex-M3, the correctly adjusted program counter is saved into the stack and the interrupt return is triggered by loading EXC\_RETURN into the program counter. Instructions, such as MOVS and SUBS, should not be used as interrupt returns on the Cortex-M3. Because of these differences, interrupt handlers and interrupt return codes need modification during porting."
- 以 ARM Cortex-M3/M4 來說，不再需要像 ARM7 那樣手動調整返回的 program counter 值，而是以 Cortex-M 硬體給定 `EXC_RETURN` 作為新的 program counter，過程中原有的 pc 值會由硬體重新計算，一旦返回到原有程式時，仍以 +4/+8 (ARM) 作為位移量
- 學組語的目的，不見得是為了改善效能，而是：
- 判斷 optimizing compiler 產生的機械碼是否正確
  - gcc 和 clang/llvm 引入大量的最佳化技術，已很難光看原始程式碼，去推知最終生成的機械碼
- 從 Google 搜尋偷到的程式是否有效益 (千萬不要人云亦云，要有判斷能力)

( 11:00-12:00 )

## [ Introduction to ARM Architecture ]

- p. 4

- objdump
  - 可以顯示所有的object file的資訊
  - 支援格式非常多: ELF, a.out
  - 是 GNU Toolchain 一部分, 可以反組譯
  - 使用時要指定架構, 如: *arm-linux-gnueabihf-objdump*)
  - objdump 和 readelf 相輔相成, 可以做到類似功能, 但做法不同(底層library實作不同): BFD
    - 可以用兩個不同工具產生的結果來互相對照
- <main> -> C語言的進入點
  - 但要有一個人跳進去執行他, 所以是從 ELF <start>開始在跳到 <main> => crt (C runtime)
- 不知如何用 objdump 得到類似結果? 試了幾種參數感覺跟投影片上都有落差
  - 使用 objdump -D 可以得到部份相似內容
    - -D = disassembler
    - 因為可能有最佳化
    - -j 看 text section

- p. 5

- Versatile Express board with Cortex-A9 (ARMv7) core will be “emulated” using Linaro builds.

WEN H     *Versatile Express board* 是?

JIM H     *Versatile Express* 是 ARM 提供的開發硬體(讓客戶廠商的參考硬體), 價格較高(19萬~3X萬 NTD), 但ARM官方正式支援, 而且 QEMU 也支援  
*"Fast Model", "Foundation Model"* (ARM Ltd.) -> cycle-accurate simulation

WEN H     Linaro 有維護 QEMU, 專門開發 Open Source 的公司

- ARM 不是純粹的RISC
  - 會因為客戶去改變設計, 如針對 Nokia 設計出 Thumb 指令集, 透過 16 位元的編碼, 仍是 32 位元的處理器
- QEMU模擬的時間不是 Cycle accurate
  - Cycle accurate: Load、Store、Add、Sub 執行的Cycle數會跟理論上的不一樣
  - 所以 Benchmark 會不精確, 只能看到趨勢變化
    - 需要真正硬體才能得知正確效能
  - *"Fast Model", "Foundation Model"* (ARM Ltd.) -> cycle-accurate simulation

- p. 7

- 為什麼Variable cycle instructions (LD/STR multiple)可以提高效能?
- –Auto-increment/decrement addressing modes

WEN H     這個的意思是?

DATE H     後面會提到 STM 跟 LDM 的東西, 那時候會有介紹

JIM H     ARMv8 (64-bit) 移除 LD/STR mutiple

- 有 LD/STR 跟 3 stage pipeline 變成 5 stage pipeline 有關
  - 增加了 MEM、WB
- 因為記憶體開銷實在太大
  - 一般 LD/STR 2個 Clock cycle
  - LD/STR multiple 4個 Clock cycle

- 一次做很多個 LD/STR
- 提升程式碼密度
- 但花的時間可能差不多
- pipeline 設計的好，multiple LD/STR 就能發揮效益
- ARMv8 (64-bit) 移除 LD/STR multiple，因為要考量到 Prediction，一a次失敗會浪費掉太多資源

• p. 8

- 在 [ARM系統開發者指南](#) 中將 exception modes 翻譯成「處理器模式」，感覺跟 exception 這個字的意思差很多，不知怎麼翻譯成處理器模式的？
- CPU modes, exception modes, processor modes 對 ARM 來講都是一模一樣

□ WEN H [ARM wiki](#) 是寫 CPU modes 還是這邊的 exception modes 不是指 CPU modes？

JIM H 以 ARM 來說，會導致 CPU 變更 execution mode 改變，就是 exception 使然，所以等價，當然，如果我們可用同樣的術語，對簡報陳述較好

WEN H ARM 切換不同模式會 exception

- 5 SPSR
  - spsr\_fiq, spsr\_irq, spsr\_svc, spsr\_undef, spsr\_abt
- Several exception modes
  - User mode, FIQ mode, IRQ mode, Supervisor (svc) mode, Abort mode, Undefined mode, System mode
  - Monitor mode, Hypervisor mode (ARMv6, ARMv7 之後)

□ WEN H 模式數量會隨著版本改變

ARMv6 之後支援多核心

簡報是 ARMv4, ARMv5

• p. 9

- context switch -> reserve/restore registers (GPR)

□ WEN H context switch 要保存暫存器

為了降低 context switch 成本，ARM 降低通用暫存器的數量

- AAPCS, ATPCS
- Function vs. Proc
  - g(x), f(x) (g of f)(x); Proc 是退化的 func，對應到 ISA 的 branch/jump
    - e.g. C 語言只能 return 一個值

□ WEN H procedure 輸出域很窄

- function 有輸入域和輸出域

- R12 or IP is not instruction pointer, it is the intra procedural call scratch register

▪ 在 [ARM系統開發者指南](#) 中 r0~r12 似乎沒有特殊功能，是書中沒寫到還是後來 ARM 新增了？

□ WEN H EX : R10 (SL), r11 (FP), r12 (IP)

- 所以 R12: intra procedural call scratch register 的功用是？(待查)
  - 保存要跳躍的範圍
- <http://blog.csdn.net/gooleman/article/details/3529413>

- 返回值會存在 R0

• p. 11 ARM vs. x86

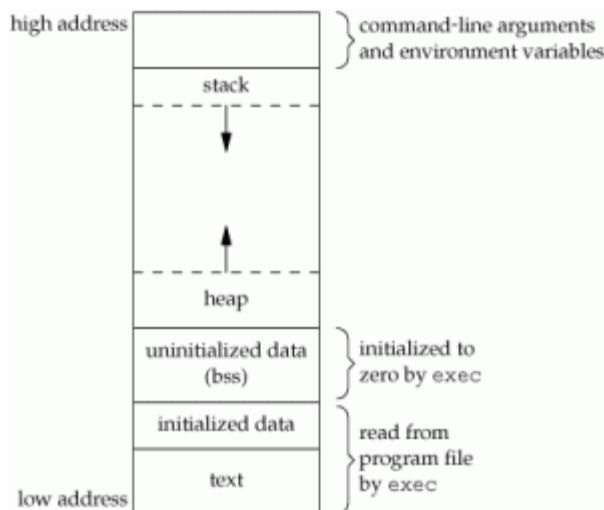
- 補了一下計組 Endianness 的知識... Endianness = 一個資料(32-bit)放在記憶體中的方式

- ARM Instructions are little endian (except on the -R profile for ARMv7 where it is implementation defined)
- p. 26 Conditional Flags
  - V 和 Q 差在哪裡??
    - V – Overflow flag
    - Q – Sticky overflow

DATE H <http://stackoverflow.com/questions/19557338/importance-of-qsaturation-flag-in-arm>

簡而言之就是飽和操作如果遇到 overflow 則會讓 Q = 1

- p. 29 PUSH operation



- stack pointer 是紀錄堆疊最後的位置

- p. 32

- sbc r0, r0, r1
  - means:  $r0 = r0 - r1 - \text{NOT}(C)$

DATE H [`- NOT\(C\)` 怎麼做? 為什麼 After Operation CPSR 沒有變化?](#)

DATE H [`- NOT\(C\)` 怎麼做`是什麼意思?](#)

ARM 需要在指令後面加上 s 才會更新 Flag

例如 : sub r1, 0

跟 subs r1, 0 <--這個才會修改 flag

COLDNEW 請參考: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0068b/CIHCJFJG.html>

- p. 37 的重點在? 看不出來上下兩個指令有甚麼差別在這頁有甚麼差別

DATE H 有號無號對於乘除是個大問題，不能像加減法用 2's complement 跟 Flag 輕鬆解決

- p. 39

- \$ objdump -d helloworld | less
  - 可改為 \$arm-linux-gnueabihf-objdump -d helloworld | less，即可順利跑出結果

- p. 43 last bit off right is Carry 這句話是最後一個向右移的位元是進位？

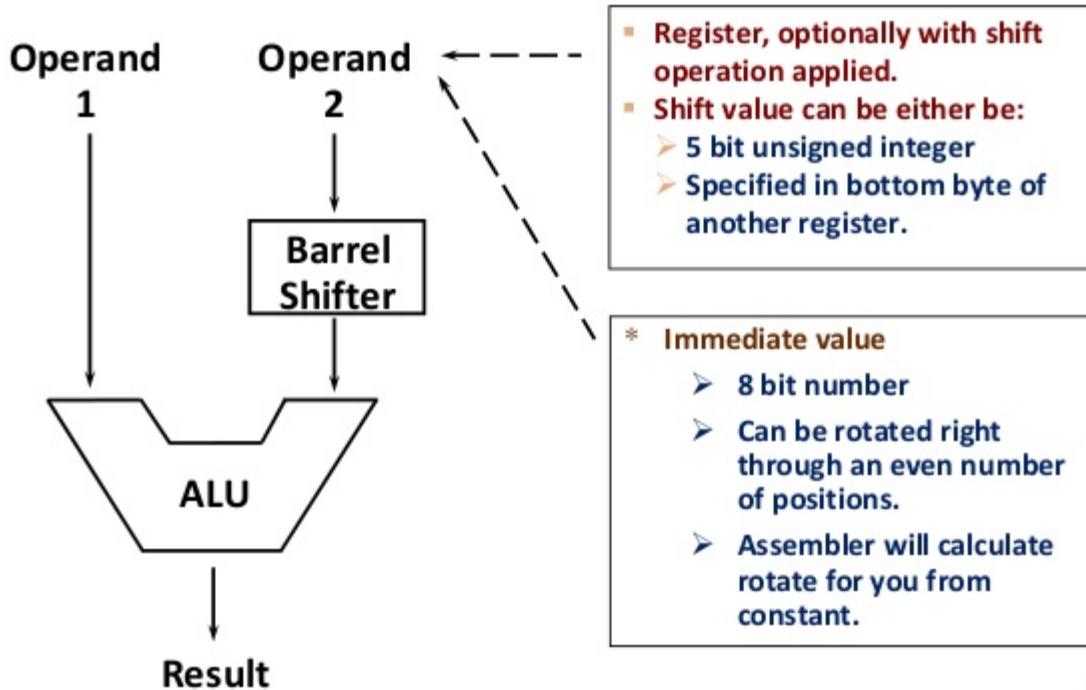
[ 16bit ] [c]  
0101010101010101 0

ROR 1

1010101010101010 1

最後一個 bit 會被送到 carry flag，也會放到最左邊的 bit 上

## Using the Barrel Shifter: The Second Operand



- Barrel Shifter 可提升程式碼密度 (空間複雜度)
  - 如: 某數 \* 3 => ADD R0, R1, R1, LSL #1 means  $R0=R1+(R1<<1)$
  - ARM 的最佳化可能會對齊到 2 的 N 次方 +1
- p. 44 RTFM = Read The F\*cking Manual
  - [Jserv's blog: 對自己好一些：談技術手冊閱讀](#)
    - 要努力找到第一手資料，以確保資料的正確性
- p. 47 ASR 如何運作
- LDR -> <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0041c/Babbfdih.html>
  - asrs r0, r0, #1
  - p. 43 ASR – Arithmetic Shift Right (MSB copied at left, last bit off right is Carry)
- WEN H MSB 是?
  - DATE H most significant bit 最高位的 bit
  - ASR, MSB 保持不變，其他 bit 向右 shift
- WEN H 可以用 QEMU + GDB 進行單步執行
  - 用 objdump 去看 machine code
  - 對 compiler 可以設定參數，進行最佳化
- 莊彥宣 asr 可以當成是有號的右移，等於是 2^n 除法
  - 在右移的時候，左邊補進來的數字也會是有號的，於是正數還是正，負依舊是負

而在此圖的案例當中,必須要用 2's complement 來探討

```
add.w r0,r0,r0,lsl #31
```

```
asrs r0,r0,#1
```

莊彥宣 為什麼要先做 add.w r0,r0,r0,lsl #31 ?

意思是指先把自己跟自己的 sign 相加,也就是如果是負數,就加一,正數則不變  
這要從二的補數來看,當你右移 #1 之後,相當於除以二

此時最右邊的那個 bit 是 "直接消失" 對於正數不會有任何影響

但對於負數來說等於是多進了一位,假設以下例子

$-3 / 2 ==> 1101 >> 1 (\text{asr}), \text{answer} = 1110 = -2$

因為二補數表示法的負數相當於"與 0 的距離"

所以對於負的奇數來說

消失的最右邊那個 bit,等於是間接幫他進位(與 0 的距離變大)ex.  $-3 ==> -4$

因此必須要先做 add.w r0,r0,r0,lsl #31 來消除掉這個進位

$-3 / 2 ==> (-3+1)/2 ==> 1110 >> 1 (\text{asr}), \text{answer} = 1111 = -1$

結論就是compiler太聰明了!

- 為什麼Lab38中 example 3 執行完會是0 ?

- HINT:因為 -6 和 8 是往右位移,不是真的除2

- 可以到Makefile中把 -O3 改成 -O0 , objdump出來的結果會比較看得懂(因為關掉最佳化)

- O3 會在編譯階段就知道值(答案)了

- 最佳化關掉和打開都會得到 0

- p. 48 RRX 如何運作

- mvn r0, r0, RRX

- p. 43 RRX – Rotate Right with Extend (bits popped off the right end first go into Carry, Carry is shifted in to left, last bit off right is Carry)

[c] [ 8bit ]

1 01010100

RRX 1

[c] [ 8bit ]

0 10101010

把 c 加入,上圖所示,把 8bit 用 c 延長成 9bit 再去 rotate

- p. 50 movw ??

OLDNEW `movw` followed by a `movt` is a common way to load a 32-bit value into a register.

WEN H 比較新的指令集才有,跟ARM版本有關(ARMv7)

JIM H -> <http://community.arm.com/groups/processors/blog/2010/07/27/how-to-load-constants-in-assembly-for-arm-architecture>

- p. 93 STMDB equals to push operation

p. 94 LDMIA equals to pop operation

- ex. in fib.s

- use `stmdb sp!, {r4, r5, lr}` instead of `push {r4, r5, lr}`

- and `ldmia sp!, {r4, r5, lr}` instead of `pop {r4, r5, lr}`

p.121 gdb arm 的 objectfile 有需要注意的事項嗎?

TSE-JEN C ``gdb example1``, ``b main``, ``r`` then I get warning: Unable to find dynamic linker breakpoint function.

GDB will be unable to debug shared library initializers

`gdb gcc`出來的code倒是可以正常 break

COLDNEW 如果你的 `example1` 是 arm binary, 你應該無法這樣執行才對 (除非你的 Linux Distro 有對 binfmt 作些設定)

TSE-JEN C 謝謝! It works like a charm!! 我在step2用 `gdb-multiarch` 也行的樣子(不知為何沒有裝到 `arm-linux-gnueabihf-gdb`)

COLDNEW 剛看一下 `gdb` 編譯參數，他可以加上 `--enable-targets=all` 這個選項，我想你用的 linux distro 應該是編譯 `gdb` 時有打開這個參數，所以就不需要 `arm-linux-gnueabihf-gdb` 了

TSE-JEN C 感謝! so 上有說到這個, 跟我用的distro有關(lubuntu 15.04)

- 針對 qemu-user 執行的程式使用 gdb

- step1: `qemu-arm -L /usr/arm-linux-gnueabihf -g 1234 ./example1`
- step2: open another terminal B, enter `arm-linux-gnueabihf-gdb` or `gdb-multiarch`
- step3: In terminal B, enter `file example1`
- step4: In terminal B, enter `target remote :1234`
- step5: enter `c` to continue program or add your breakpoint, e.g. `b main`
- 

## 回顧 ARM 架構

投影片: <https://drive.google.com/file/d/0B5GW0alORHIBUjVvUXJ2NVhPckU/view?ths=true>

## Evolution of ARM

[ [source](#) ] ARM 緣起於 1978 年底，物理學家 [Hermann Hauser](#) 與工程師 [Christopher Curry](#)，在英國劍橋創辦了 CPU (Cambridge Processing Unit) Ltd. 公司，後者主要業務是為當地市場供應電子設備。1983 年，CPU 公司更名為 Acorn Computer 公司。

不久後，Acorn Computer 推出了第一款產品是 Acorn System 1。這款產品主要瞄準大學市場，售價 80 磅，只配備了一個小的 LED 螢幕、一個按鍵和盒式磁帶介面。在 Acorn System 1 後，陸續生產了 System 2, 3 和 4，還有消費者導向的 Acorn Atom。

1981 年跟隨英國廣播公司 BBC 聯合推廣——他們與 Acorn 合作推出了 BBC Micro 電腦。BBC Micro 採用的處理器架構採用的是效能相對較低但也省電、低成本的「精簡指令集」(RISC) 架構。儘管這款產品引熱了英國的電腦市場，但好景並沒有持續太長：除了作為 BBC Micro 的產品持續輸出到廣播局和學校外，這些產品在英國以外幾乎毫無市場。這家公司在接下來的幾年也並沒找到更好的策略，瀕臨破產。

隨後 Acorn 公司一度被當時的 IT 巨頭 Olivetti 收購，爾後又被 Andy Hopper 提議分拆單獨成立商業運作公司。

這時候 Apple Computer 出現了，原因正是因為 Apple Computer 正在開發新的 PDA 裝置 Newton，Acorn/ARM 晶片符合高效能、低功耗的特性，ARM 為蘋果開發了專門設計的 ARM 610 晶片。

1990 年 11 月 27 日，Acorn 公司正式改組為 ARM (Advanced Risc Machine) 公司：蘋果出資了 150 萬英鎊，Acorn 本身則以 150 萬英鎊的知識產權和 12 名晶片工程師技術入股，另一個晶片廠商 VLSI 出資 25 萬英鎊。

正是在這個時期團隊初期缺乏資金，ARM 做出了影響時代的決定：ARM 不製造晶片，只將晶片的設計方案授權給其他公司，由它們來生產。自此，ARM 的公司在商業上開啟了設計授權之路。

這之後就是 ARM 的一些成績了：一年後，ARM 最先將產品授權給英國 GEC Plessey。兩年後，另一家知名公司德州儀器也加入合作，再後來半導體產業的巨頭三星隨後也加入了 ARM 授權行列。

ARM 創辦人 Hermann Hauser 在接受 BBC 採訪表示，ARM 被軟銀收購的當天將成為「英國科技產業最令人難過的一天」。Hermann 還表示，當他聽說軟銀將斥資240 億英鎊收購ARM 時，感覺「非常難過」。



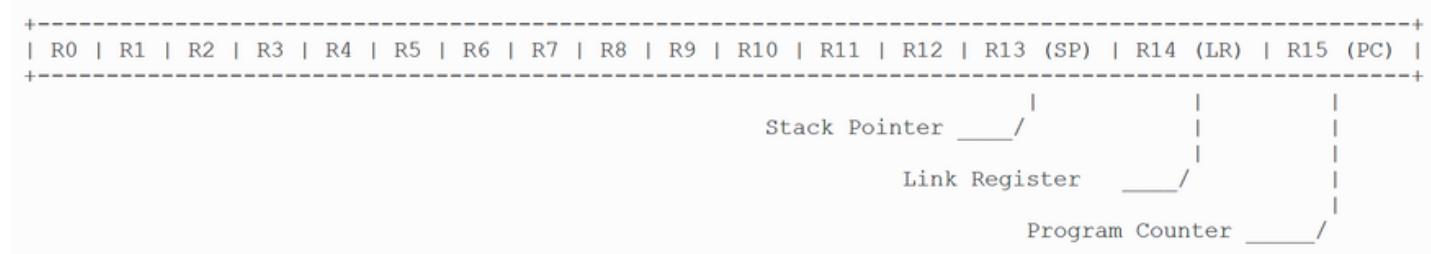
可見 ARM 被收購一案確實令人矚目，因為這家公司也代表了英國的科技技術。

介紹 ARM 的起源、發展以及獲利模式 (賣 IP，矽智財)，以及 Linaro 公司，後者致力加速 ARM 平台相關的開放原始碼專案，由 ARM、Freescale 也就是現在的 NXP、TI 等公司出錢出力發起。

## Evolution of the ARM ISA

到目前為止，ARM 共 8 種 ISA 版本，也就是 ARMv1 ~ ARMv8。其中 ARMv1 和 ARMv2 位址範圍只到 26 bits，自 ARMv3 開始則採用了 32 bits 位址範圍，也因此在這份投影片中，將以 ARMv3 作為 ARM 基本 ISA (ARM's basic ISA) 來進行演化的討論。

基本上 ARM 處理器具有 16 個 32 bit 長度的暫存器，其中有 13 個為 **通用暫存器 (General Purpose Registers, GPRs)**，R13-R15 則有其他用途。



R13 通常會被用來當作堆疊指標 (Stack Pointer, SP)，在實際使用中，一般會在記憶體分配一些空間作為堆疊，系統初始化時將這一塊堆疊的底部位址儲存到 R13。

R14 為 連結暫存器 (Link register, LR)，用來存放副程式的返回地址，比如我們在組語中呼叫到了 BL、BLX 等指令時，會將 PC 的數值複製到 R14 中，作為返回 (return) 的位址，具體範例如下。

```

1
2     ...
3     BL calc      ; Jump to calc
4     ...          ; Execute here after return
5     ...
6
7     calc:         ; function body
8     ADD r0, r1, r2 ; do some calculate here
9     MOV pc, r14    ; PC = R14 to return
10
11;; This example is taken from: http://www.davespace.co.uk/arm/introduction-to-arm/branch.htm

```

**bl.asm** hosted with ❤ by GitHub [view raw](#)

R15 則是程式計數器(Program Counter, PC)，用來存放下一道指令的位址，根據 [ARM7TDMI Technical Reference Manual](#)

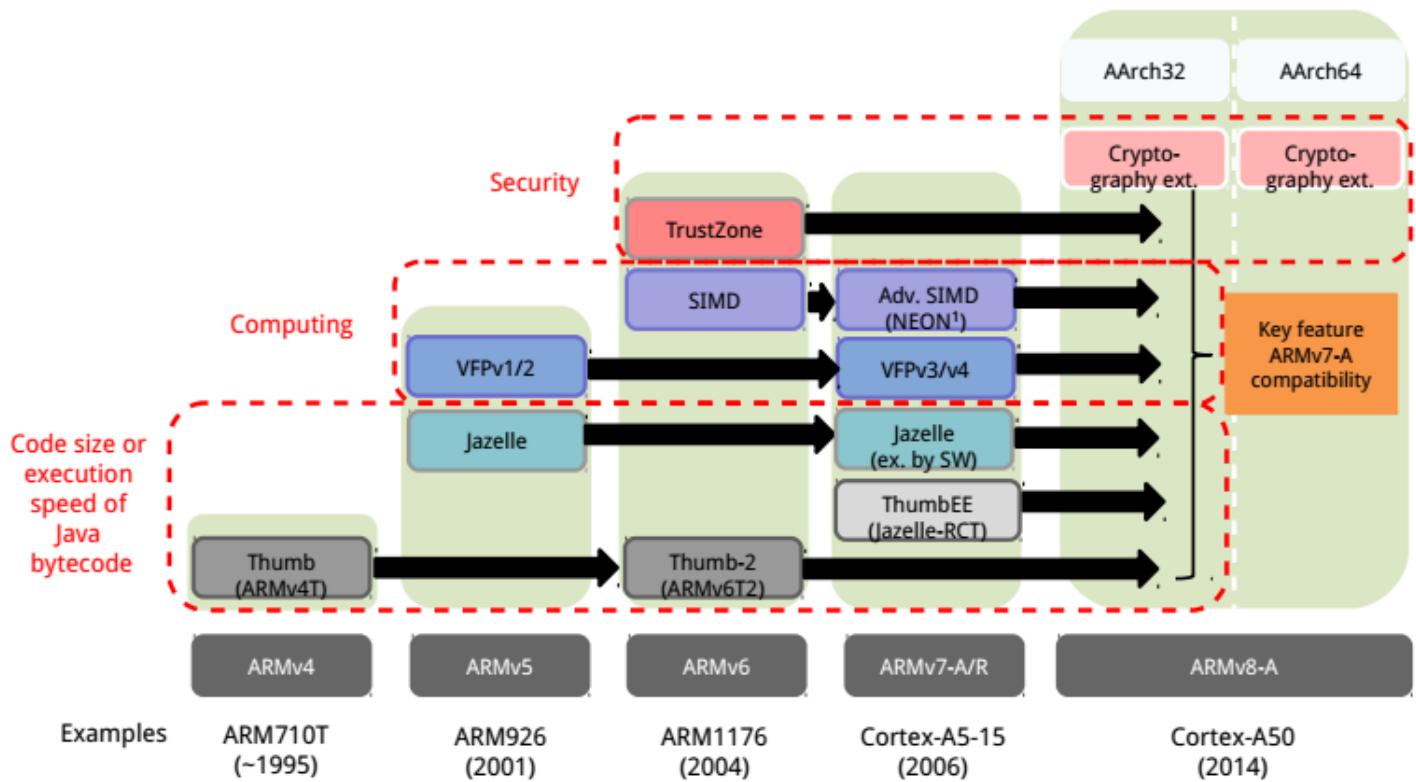
，R15 在 ARM 或是 Thumb 模式下狀況不同

- ARM 模式 (ARM state)
  - bits [1:0] 未定義且會被忽略，bits [31:2] 保留了 PC 數值
- Thumb 模式 (Thumb state)
  - bit [0] 未定義且被忽略，bits [31:1] 保留了 PC 數值

除了上面的基礎 ISA 外，ARM 根據不同的狀況增加了許多種 ISA 的擴充，比如在 Java 很火紅的年代，為了提昇 JVM 執行的效率，而引入了 [Jazelle 指令集](#)，用來協助增強 Java bytecode 運算的速度/佔用空間 (不過由於 Android 的 dalvik 並未將 Java 程式轉換成 bytecode 的形式，因此在 Android 平

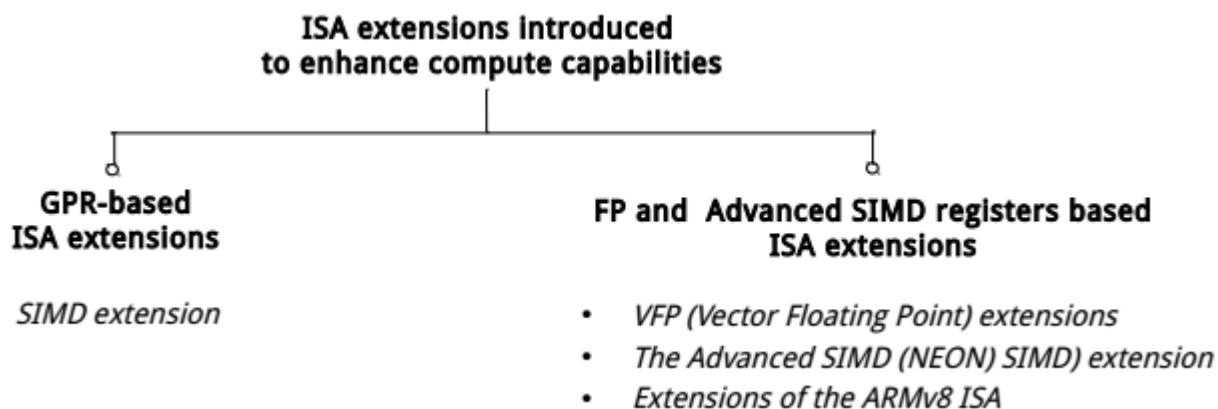
台上 Jazelle 指令集並未有任何顯著的效能提昇)

各個不同的 ARM 版本對應的擴充指令集架構資訊如下：



而這些 ISA 擴充架構則又可以分為兩組：

- 通用暫存器 (General Purpose Registers, GPRs)
- FP (Floating Point) 和 Advanced SIMD (NEON)



## SIMD extension

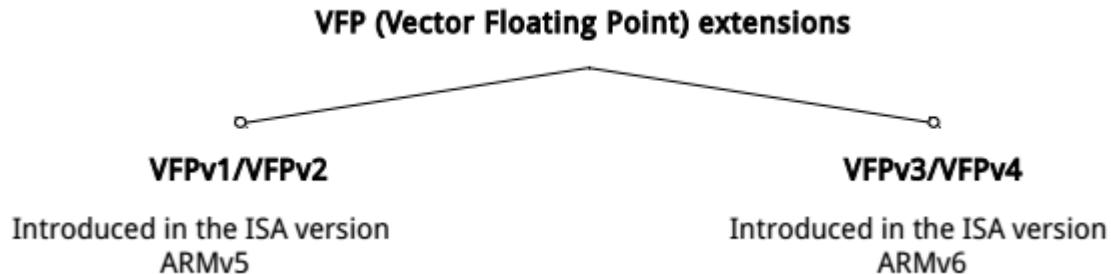
我們先來看 SIMD (Single Instruction Multiple Data) extension, 他其實是透過通用暫存器 (General Purpose Registers, GPRs)，也就是 R0 ~ R12 這 13 個 32-bit 暫存器所組成，這項擴展自 ARMv6 引入，但是由於效能提昇有限，自 ARMv7 後被 Advanced SIMD, 也就是我們說的 NEON 所替代掉。

此外，早期的 ARM 處理器並沒有負責處理浮點數運算的功能，因此浮點數的運算就必須透過 CPU 來進行處理，對於越來越多的浮點數要求 (影像處理、音訊、遊戲 ... etc) 若沒有額外的計算輔助，則 CPU 會耗費非常多的時間進行浮點數的運算，為了解決這個問題，ARM 加入了 VFP (Vector Floating

Point) 這種透過協同處理器來輔助計算浮點數的應用。

## VFP extensions

VFP (Vector Floating Point) 指令集擴充可以分兩個部份來討論，一個是自 ARMv5 引入的 VFPv1/VFPv2，另外一個則是自 ARMv6 引入的 VFPv3/VFPv4。



## VFPv1/v2

VFPv1/v2 自 ARMv5 引入，具有 32 個 VFP 暫存器，並可分成四個暫存器庫區(Register Banks)，每一區具有 8 個 VFP 暫存器，如下圖

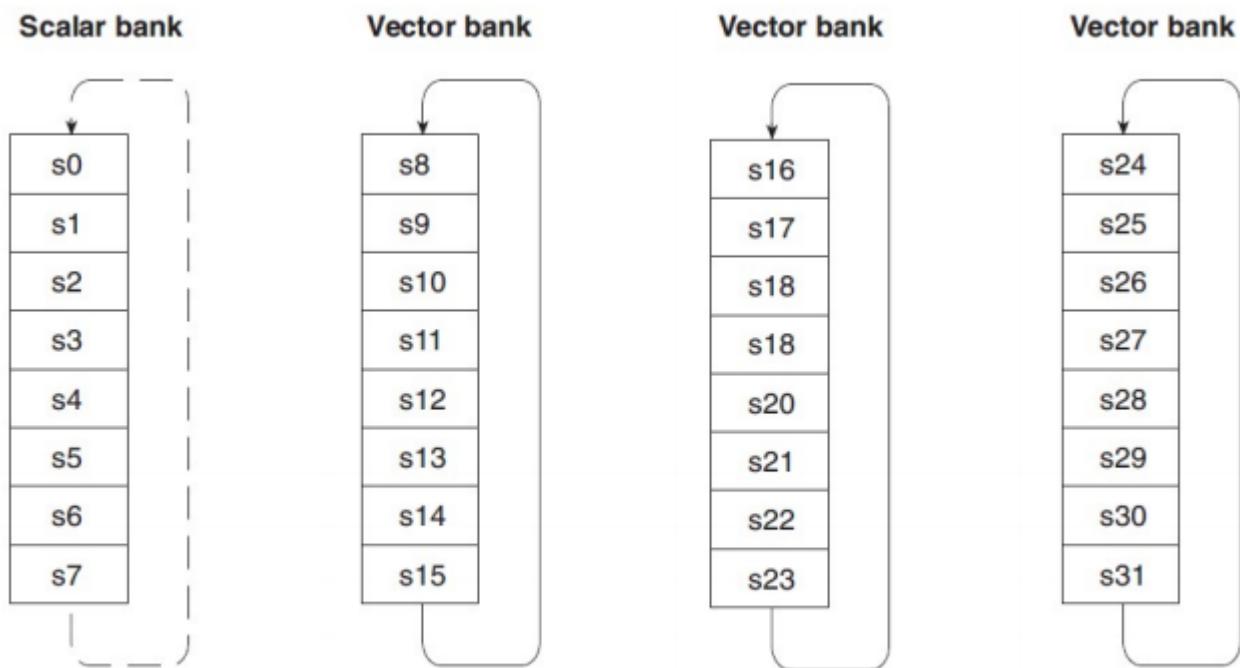


Figure: Register banks of the VFP extensions [65]

從上圖我們可以看到，在 VFPv1/v2 中，第一個暫存器庫區 (Register Banks) 存放了純量 (Scalar) 運算元，剩下的三區則是存放向量 (Vector) 運算元。和 SIMD (Single Instruction Multiple Data) 不同的是，向量是依序 (sequential) 處理，而不是像 SIMD 那樣同步進行。

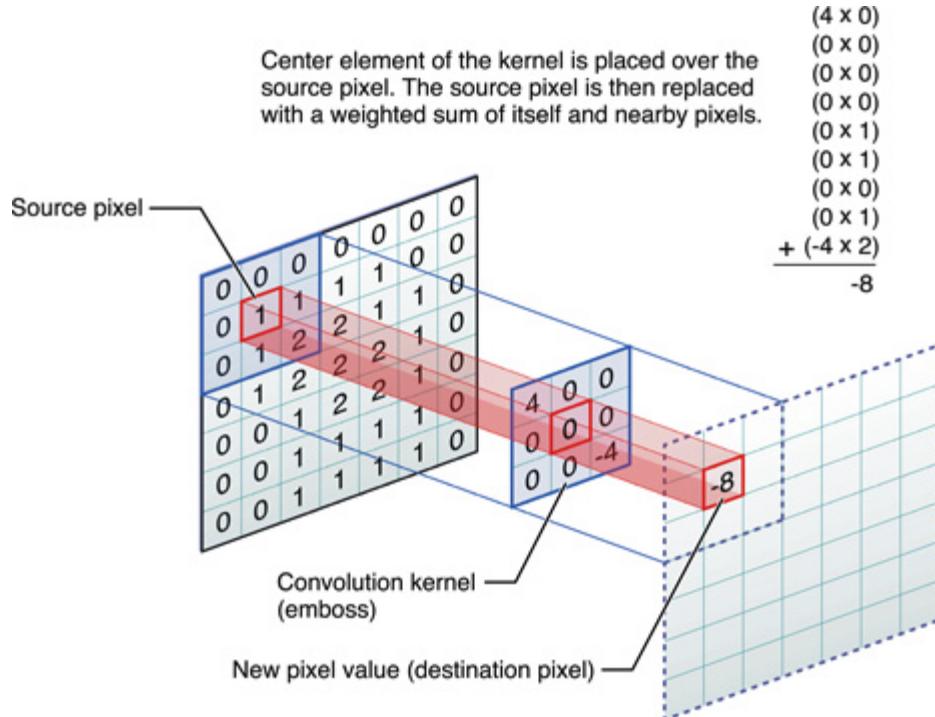
## VFPv3/v4

如同名稱一樣，VFPv3/v4 是 VFPv1/v2 的延伸，自 ARMv6 開始引入。和 VFPv1/v2 不同的是，VFPv3/v4 的 VFP 暫存器變成 64 bit 暫存器，並增加了一些指令協助 FX (Fixed Point) 與 FP (Floating Point) 之間的轉換。

## The Advanced SIMD (NEON)

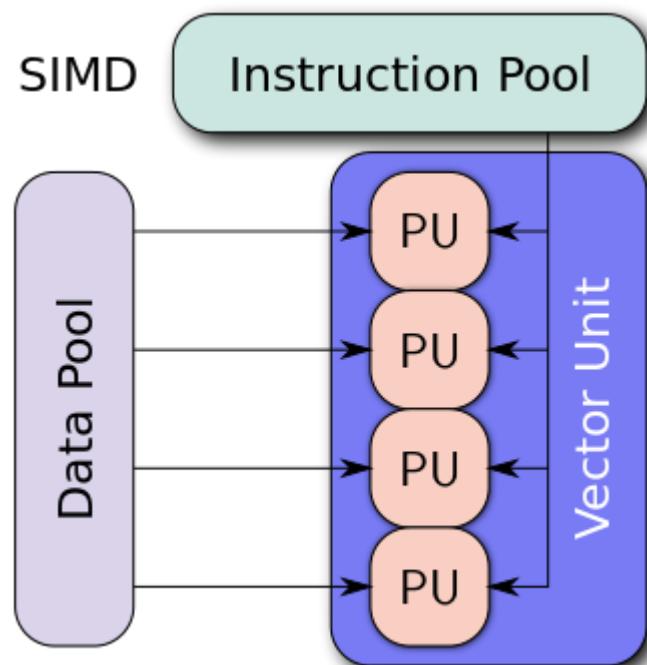
NEON 指令集自 ARMv7 引入，為 64/128-bit SIMD (Single Instruction Multiple Data) extension。NEON 指令集被設計用來補足日益興盛的影像編碼/解碼、2D/3D 圖像處理、遊戲、影像處理等功能。

為何這類用途需要額外的指令集去處理？以影像處理為例，影像的處理其實就是透過遮罩(mask)去對2維影像陣列進行捲積(convolution)的運算，就像這樣 ([圖片來源](#))



對於這種運算，我們是可以同時對陣列(vector)的各個元素進行處理的，也就是這些運算可以平行處理(parallel)，可以加快運算速度。

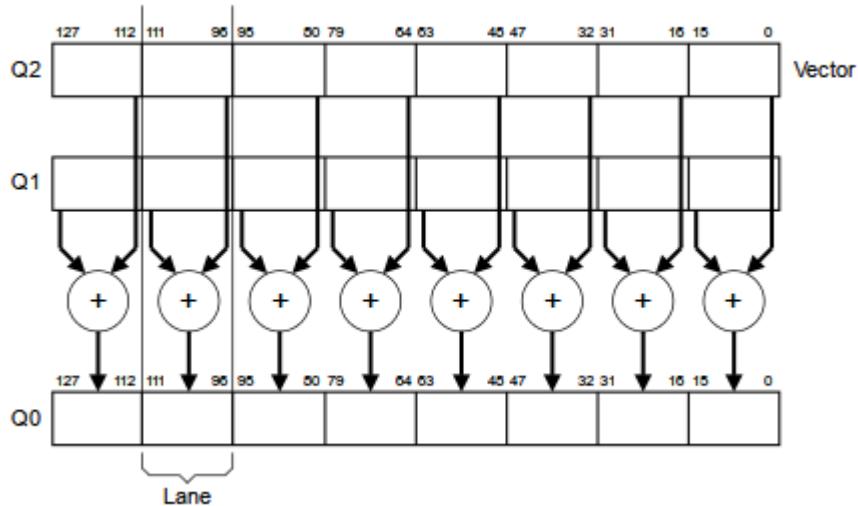
順道一題，SIMD (Single Instruction Multiple Data) 這種運算模式同時也是費林分類法([Flynn's Taxonomy](#))中的一種運算結構。



若以 NEON 指令集的命令來看，**VADD.I16 Q0, Q1, Q2** 這樣的指令，會執行一個平行的陣列加法，將

Q1 以及 Q2 各元素的運算結果存放到 Q0 中。([圖片來源](#))

**Figure 1.2. 8-way 16-bit integer add operation**



## ARM NEON 案例分析

給定每個 pixel 為 32-bit 的 RGBA 的 bitmap，其轉換為黑白影像的函式為：

```
void rgba_to_bw(uint32_t *bitmap, int width, int height, long stride) {
    int row, col;
    uint32_t pixel, r, g, b, a, bw;
    for (row = 0; row < height; row++) {
        for (col = 0; col < width; col++) {
            pixel = bitmap[col + row * stride / 4];
            a = (pixel >> 24) & 0xff;
            r = (pixel >> 16) & 0xff;
            g = (pixel >> 8) & 0xff;
            b = pixel & 0xff;
            bw = (uint32_t)(r * 0.299 + g * 0.587 + b * 0.114);
            bitmap[col + row * stride / 4] = (a << 24) + (bw << 16) + (bw << 8) + (bw);
        }
    }
}
```

人眼吸收綠色比其他顏色敏感，所以當影像變成灰階時，僅僅將紅色、綠色、藍色加總取平均，這是不夠的，常見的方法是將 red \* 77, green \* 151, blue \* 28，這三個除數的總和為 256，可使除法變簡單

請提出效能改善的方案：

- 建立表格加速浮點數操作 (L1 cache?)
- 減少位移數量

Hint: 考慮以下寫法

```
bwPixel = table[rgbPixel & 0x00ffffff] + rgbPixel & 0xff000000;
```

==> 16 MB; 表格太大

Hint: 如果先計算針對「乘上 0.299」一類的運算，先行計算後建立表格呢？

```
bw = (uint32_t) mul_299[r] + (uint32_t) mul_587[g] + (uint32_t) mul_144[b];
```

```
bitmap[col + row * strike / 4] = (a << 24) + (bw << 16) + (bw << 8) + bw;
```

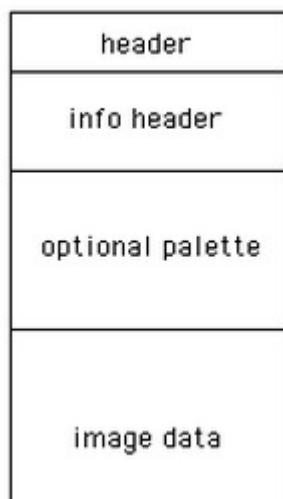
==> 降到 32 KB 以內; cache friendly

目前實作的程式碼：[embedded-summer2015 / RGBAtobW](#) (Github)

BMP (BitMap) 檔是是很早以前微軟所開發並使用在 Windows 系統上的圖型格式，通常不壓縮，不像 JPG、GIF、PNG 會有破壞性或非破壞性的壓縮。雖然 BMP 缺點是檔案非常大，不過因為沒有壓縮，即使不借助 OpenCV、ImageMagick 或 .NET Framework 等等，也可以很容易地直接用 Standard C Library 作影像處理。

BMP 主要有四個部份組成

1. Bitmap File Header : Magic Number ('BM')、file size、Offset to image data
2. Bitmap Info Header : image width and height、the number of bits per pixel、Compression type
3. Color Table (Palette)
4. Image data



未優化版本：

以下是我使用一張 1920x1080 的 BMP 圖片所印出來的資訊

==== Header =====

Signature = 4D42

FileSize = 8294456

DataOffset = 54

===== Info =====

Info size = 40

Width = 1920

Height = 1080

BitsPerPixel = 32

Compression = 0

=====

RGBAtobW is in progress....

Save the picture successfully!

Execution time of `rgbaToBw()` : 0.034494



=>執行時間 : 0.034494 sec

## 優化版本：

### Version 1

RGB 分別都是 8 bit，可以建立三個大小為 256 bytes 的 table，這樣就不用在每次轉 bw 過程中進行浮點數運算。

原本 : `bw = (uint32_t) (r * 0.299 + g * 0.587 + b * 0.114);`

查表 : `bw = (uint32_t) (table_R[r] + table_G[g] + table_B[b]);`

=>執行時間 : 0.028148 sec

### Version 2

參考了 [LIU TIM](#) 的實作，使用 pointer 的 offset 取代原本的繁雜的 bitwise operation。

```
uint32_t *pixel = bmp->data;
```

```
r = (BYTE *) pixel + 2;
```

```
g = (BYTE *) pixel + 1;
```

```
b = (BYTE *) pixel;
```

=>執行時間 : 0.020379 sec

### Version 3

將上述兩種優化方法合併在一起

=>執行時間 : 0.018061 sec

<https://github.com/charles620016/embedded-summer2015>

### Version 4 and Version 5

在 [Facebook 嵌入式系統社團](#)上有位朋友 [Bi-Ruei Chiu](#) 以上述程式為基礎，使用 NEON instruction set 來加速，執行環境是 Raspberry Pi 2 (Cortex-A7 x4)，這是他實作的程式碼 ([GitHub](#))。

用 Raspberry Pi 2 測試：

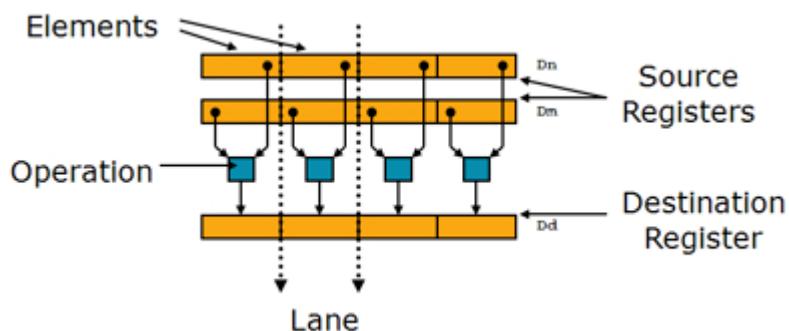
```
CC = gcc-4.8
CFLAGS = -O0 -Wall -fno-tree-vectorize -mcpu=cortex-a7 -mfpu=neon-vfpv4 -mfloat-abi=hard
```

Execution time of rgbaToBw() : 0.353600  
[Version 1 : using RGB table]  
Execution time of rgbaToBw() : 0.319600  
[Version 2 : using pointer]  
Execution time of rgbaToBw() : 0.251800  
[Version 3] : versoin1 + versoin2  
Execution time of rgbaToBw() : 0.226800  
[Version 4] : NEON  
Execution time of rgbaToBw() : 0.016000  
[Version 5] : NEON (unroll loop + PLD)  
Execution time of rgbaToBw() : 0.013200

什麼是 NEON？看官方解釋：

NEON technology is an advanced SIMD (Single Instruction, Multiple Data) architecture for the ARM Cortex-A series processors.

- Registers are considered as **vectors of elements** of the same **data type**
- Data types can be: signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit, single precision [floating point](#)
- Instructions perform the same **operation** in all **lanes**



Register :

- 16 x 32-bit general purpose ARM registers (R0-R15).
- 32 x 64-bit NEON registers (D0-D31) OR viewed as 16x128-bit registers (Q0-Q15).

簡言之，有了 NEON instruction set，就可以「同時」操作許多個 8, 16 或 32-bit 的資料，在訊號處理、影像處理、視訊解碼等有很高的應用價值。

首先先看 Version 4：

將 RGB三色的 weight 丟進 r3 - r5。

`vdup.8` (Vector Duplicate)，分別複製到大小為 8 bit 的 NEON register d0 - d2

```
mov r3, #77
mov r4, #151
```

```
mov r5, #28
vdup.8 d0, r3
vdup.8 d1, r4
vdup.8 d2, r5
```

vld4.8 (Vector Load) , 載入 pixel 的資料到 4 個 8-bit 的 NEON register d4-d7 , 其中那個 4 是 interleave , 因為我們有 ARGB , 所以 gap = 4 , 從下面兩張圖就可以看出 vld 的用法。

再來就是計算 weighted average 啦。Vector Multiply 和 Vector Multiply Accumulate

```
@ (alpha,R,G,B) = (d7,d6,d5,d4)
vld4.8 {d4-d7}, [r0]!
vmull.u8 q10, d6, d0
vmlal.u8 q10, d5, d1
vmlal.u8 q10, d4, d2
```

將值除以256就是我們要的灰階值了。

vrshrn (Vector Shift Right by immediate value)

```
vrshrn.u16 d4, q10, #8
```

最後將結果儲存。

vst (Vector Store)

```
vst4.8 {d4-d7}, [r3]!
```

=>執行時間 : 0.016000 sec

從上面瀏覽過一遍用到的 NEON instruction set , 就可以發現我們都是一次對多個 NEON register 作操作 , 下面討論分析有比較圖就可以看出效能差距。

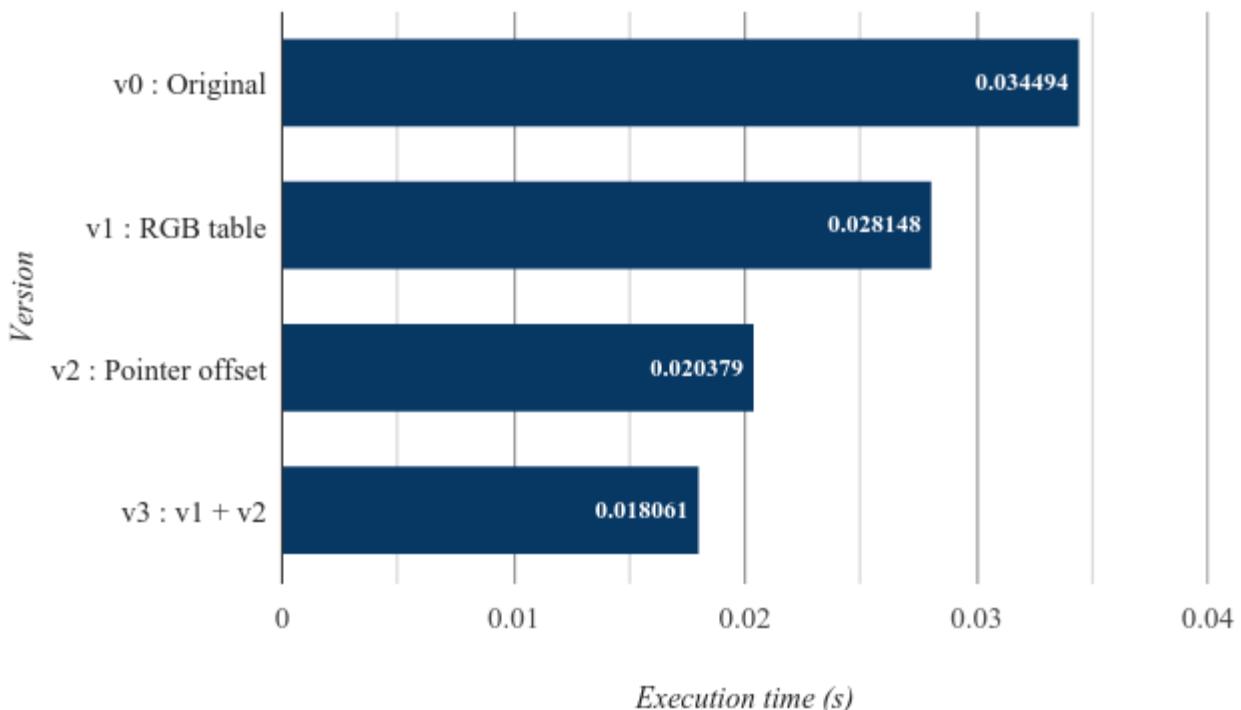
再來看 Version 5

CHARLES L (Cycle Counter for Cortex A8 這網站好酷阿) ~我會在仔細研究一下。

=>執行時間 : 0.013200 sec

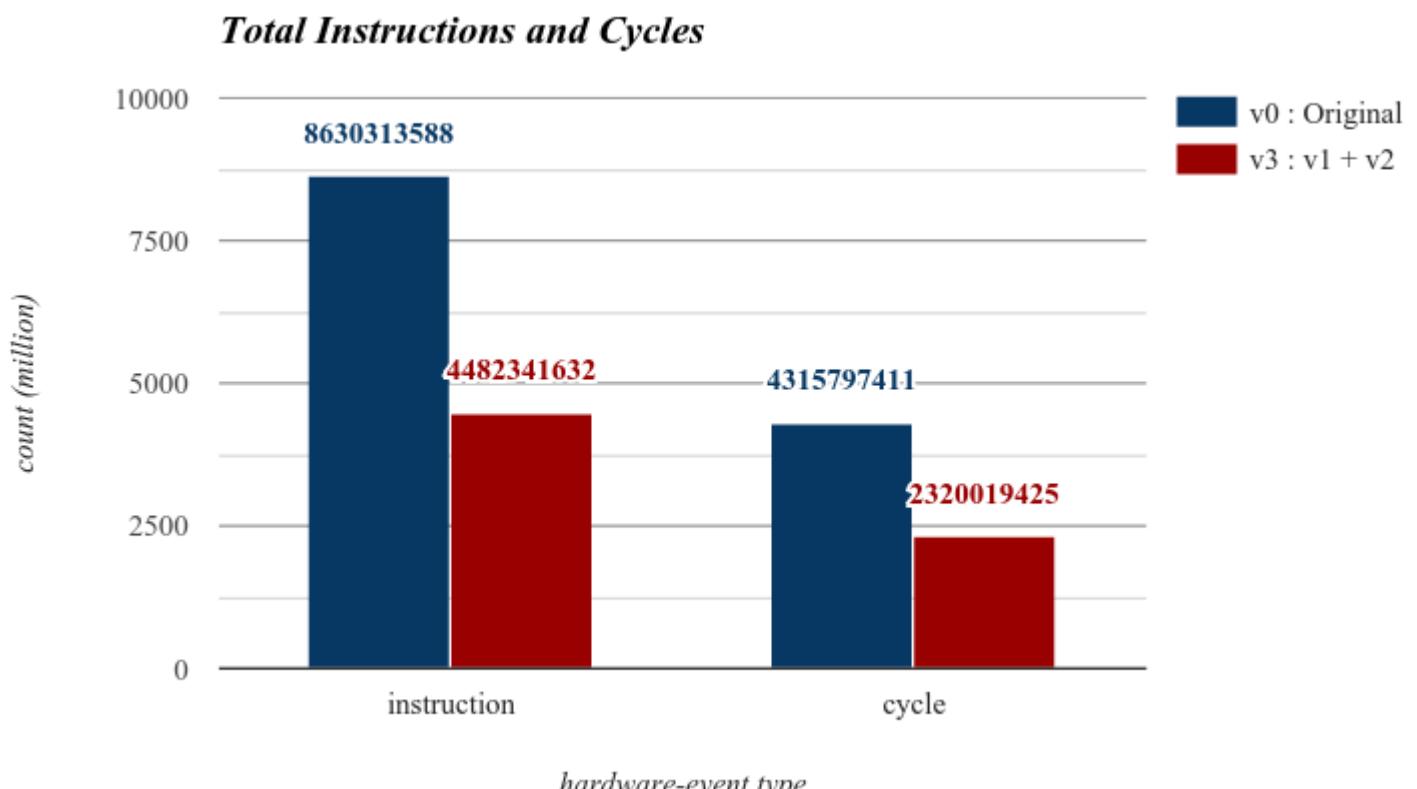
討論與分析 :

### ***Execution time of RGB to Grayscale Conversion***



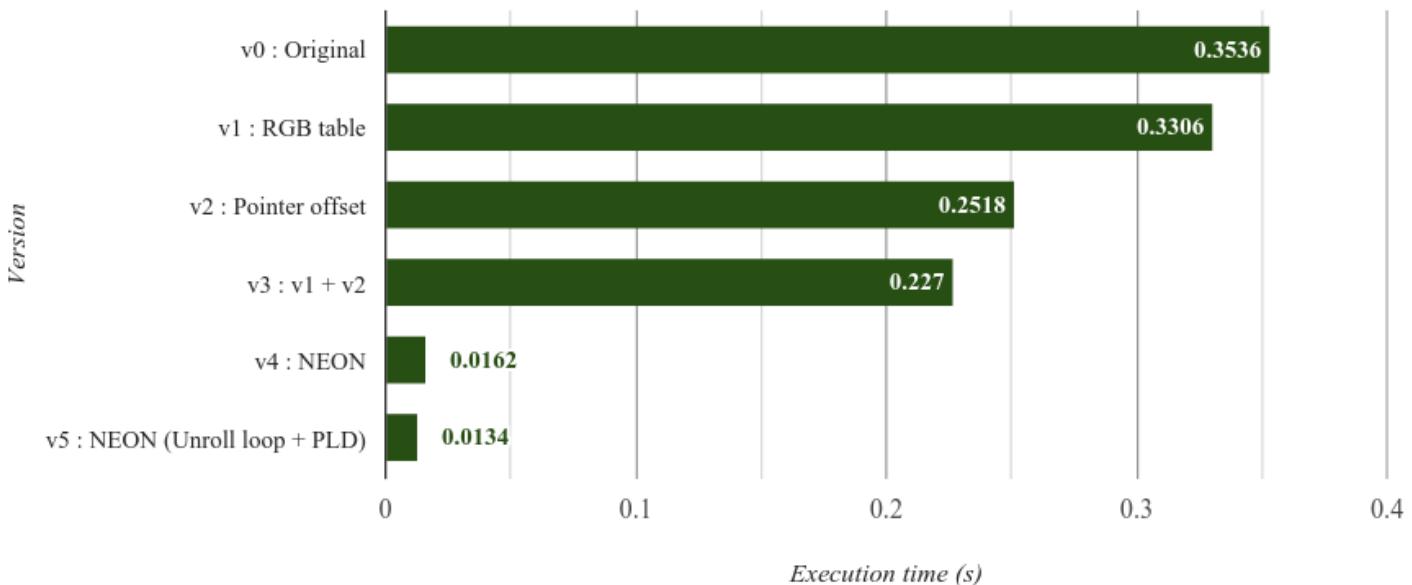
以上各版本執行時間都是 50 次迴圈平均下來的結果，測試檔為 1920x1080 32bit bmp 圖片。可以明顯看到 v2 效能表現比起 v1 來說 還好上許多，可見原始程式中「多次」的 bitwise operation 結果所帶來損耗比起浮點數運算還更多一些。若我們再將浮點數運算改成查表的話，最後時間能進步到 0.018061 secs，幾乎是原來的一半。

另外我也使用 perf 效能分析工具來觀察原始版本和 version 3 中 instruction 和 cycle 數量。



可以看到 version 3 的 instruction 和 cycle 大約都只有原來的一半，這結果也正好反應在上面的執行時間上。

### *Execution time of rgbaToBw() on Raspberry Pi 2*



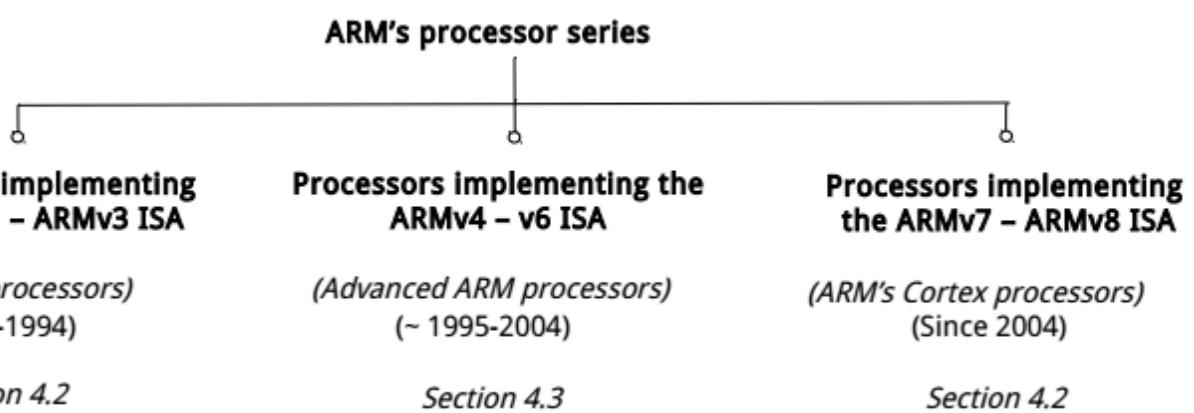
從這張表就可以很清楚瞭解使用 NEON 指令集加速後所得到的效能增長，Version 4 只花了原本 4.6 % 的時間就完成彩色轉灰階處理。

## 邁向 ARMv8

- [ARMv8-M architecture: what's new for developers](#) (video)
- [ARM64 vs ARM32 -- What's different for Linux programmers](#)

## Overview of ARM's processor series

到目前為止(投影片寫的年代)，ARM 系列大致上可以分為以下幾類，這將會在投影片的不同章節來提及



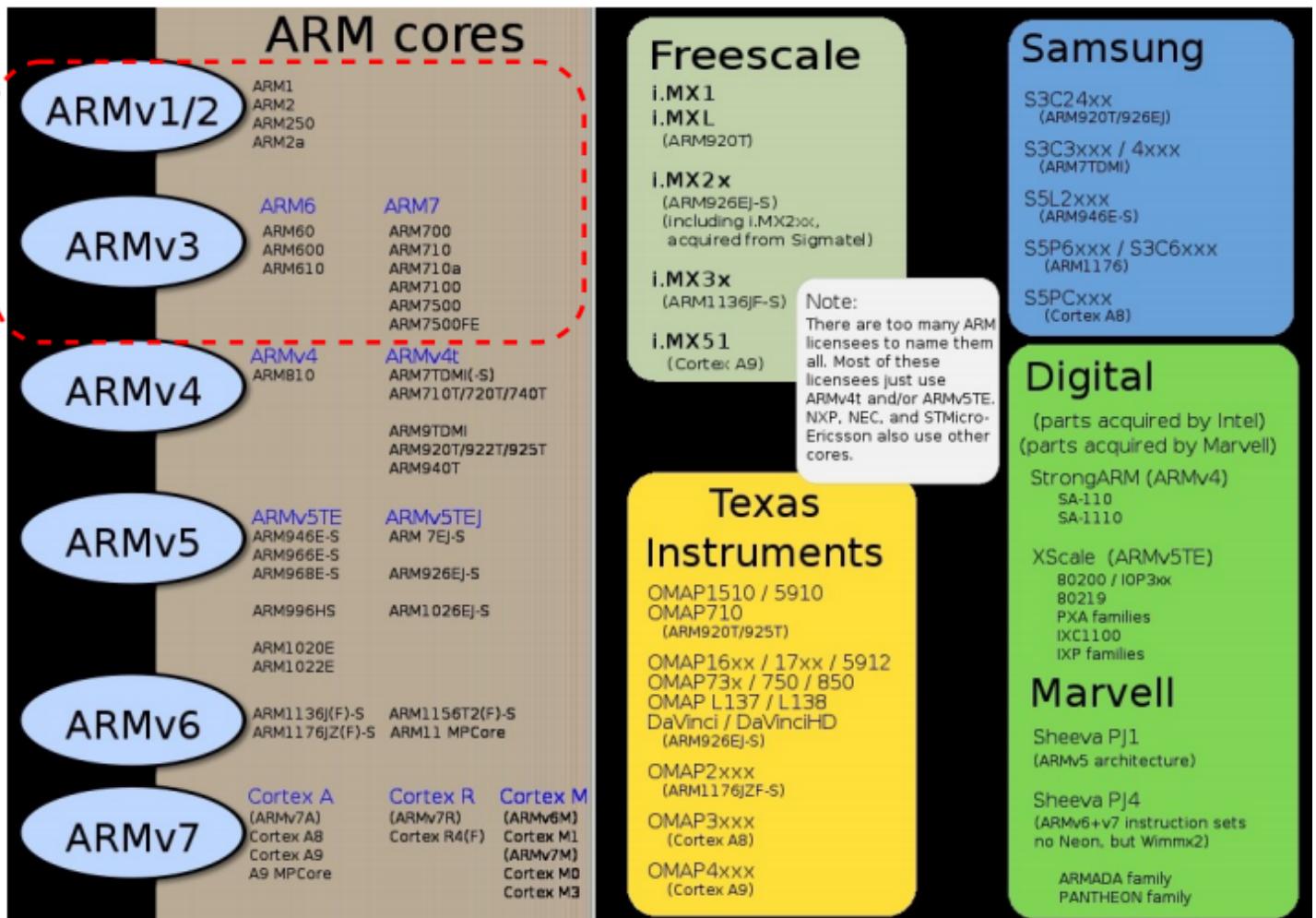
### Processors implementing the ARM v1 - ARM v3 ISA

如同 [Overview](#) 裡面提及的那樣，ARM v1 以及 ARM v2 只有實作 26-bit 位址匯流排(address bus)以及 32-bit 的資料匯流排(data bus)，也因此，ARM v1 以及 ARM v2 皆屬於 26-bit 的 CPU 結構。

到了 ARMv3 時候狀況就些許不同了，ARM v3 採用了 32-bit 位址匯流排(address bus)以及 32-bit 的資

料匯流排(data bus)，這種 32-bit CPU 架構一直延續到 ARMv7。

ARMv8 開始，CPU 架構則更改為 64-bit。



## Processors implementing the ARM v4 - ARM v6 ISA

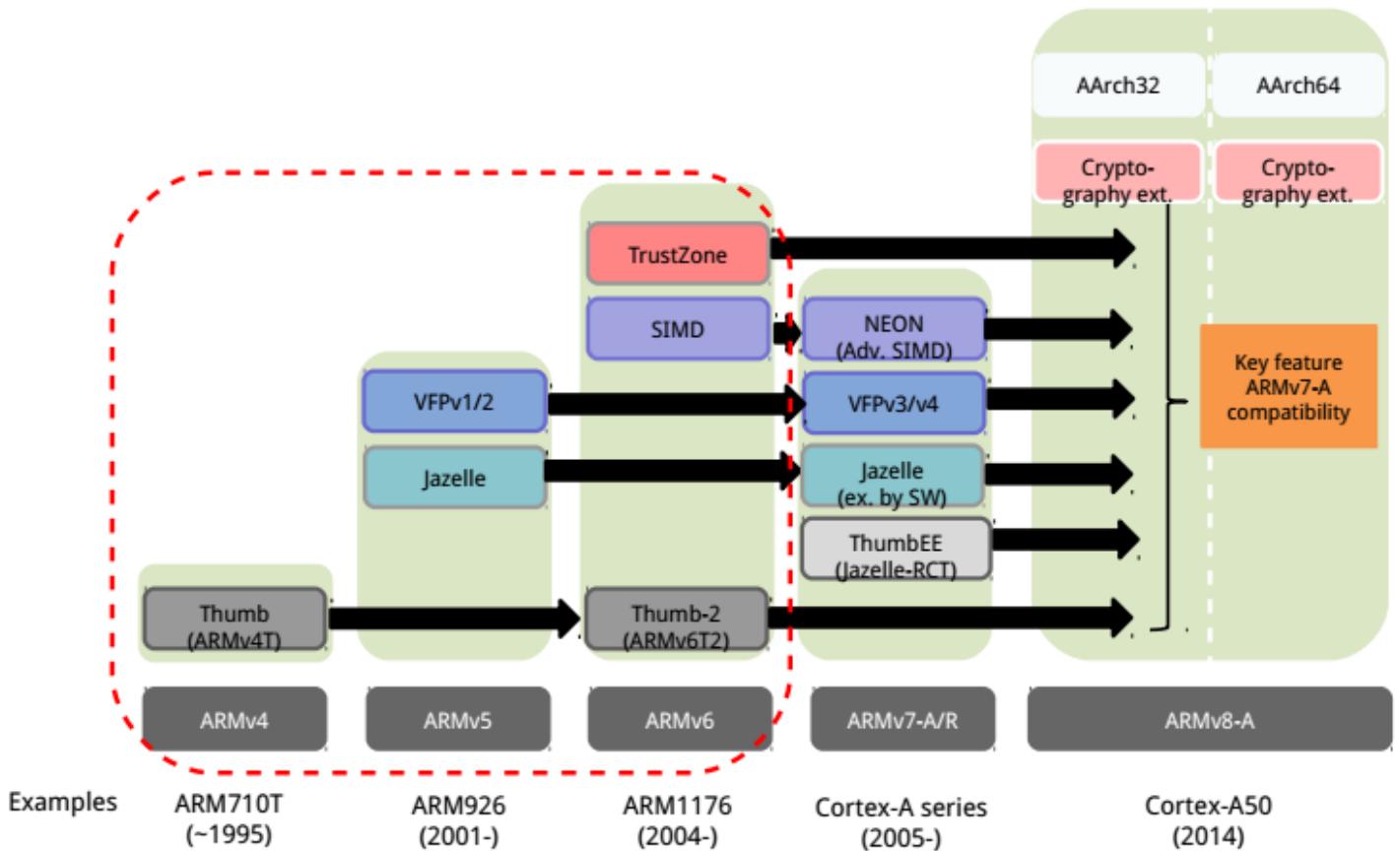


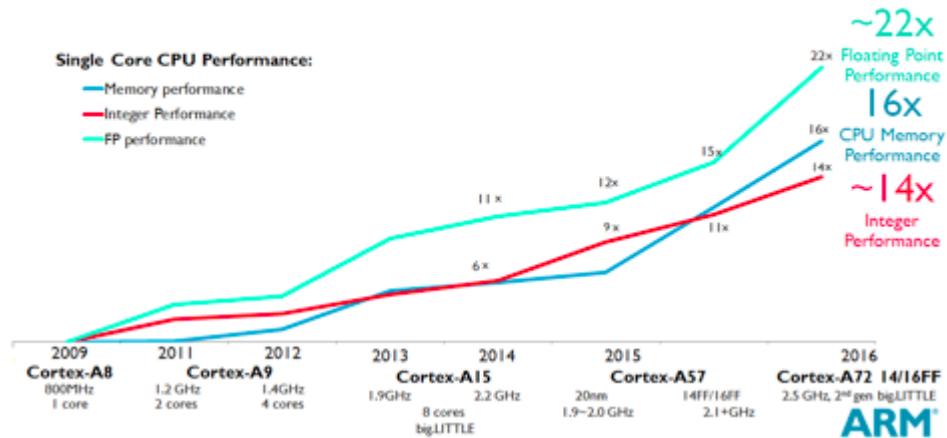
Figure: Enhancements of the ARMv6 ISA

自 ARMv4 開始，開始有針對特定需求而增加的延伸指令集。以 ARMv4 增加的 Thumb 指令為例，當時 Nokia 決定採用 ARM 的 IP 後，Nokia 為了減少程式碼密度(code density)，便派遣了不少工程師協助 ARM 建立了 Thumb 指令集 ([故事來源](#))。

Initially Nokia worried about ARM's code density, which was quite poor. This translated to more memory and hence higher cost. ARM engineers listened to Nokia and got thinking. This was how the more memory efficient THUMB architecture was born. Likewise, TI wanted the ARM core on a smaller die size. ARM engineers initially believed this to be impossible. TI engineers knew that the goal was realistic. When ARM engineers achieved what they had thought was impossible, they got a fat bonus.

而其他的 ISA 也是有相應的理由才發展出來的，比如前文提到的 [Jazelle 指令集](#) 就是用來協助增強 Java bytecode 運算的速度/佔用空間，畢竟在當時的手機，也就是我們稱為傻瓜手機(傳統手機)的時代，Java ME (J2ME) 的遊戲是非常盛行的。

## CPU Performance is Accelerating



### A walk through of the Microarchitectural improvements in Cortex-A72

(2015) 依據 [ARM](#) 指出，過去 5 年，ARM 處理器效能提升了 50 倍，具體反映在 Cortex-A72 的實做，此外，功耗的改善也是關鍵。以手機應用程式的使用情境來說：

"2.5GHz Cortex-A72 CPU consumes 30~35% less power than the 28nm Cortex-A15 processor, still delivering more than 2x the peak performance."

改動部份：

- \* Pipeline front end
- \* Decode/Rename block
- \* Dispatch/Retire
- \* FPU & SIMD
- \* Load/Store unit

## ARM big.LITTLE technology

投影片: <https://drive.google.com/file/d/0B5GW0alORHIBLW0ycldfZHhieHc/view?ths=true>

- [ARM big.LITTLE Technology Explained](#) (video)
- [多核心處理器有前途嗎？](#)
- <http://www.linaro.org/blog/core-dump/energy-aware-scheduling-eas-project/>

### Introduction to the big.LITTLE technology

#### Global task scheduling (GTS)

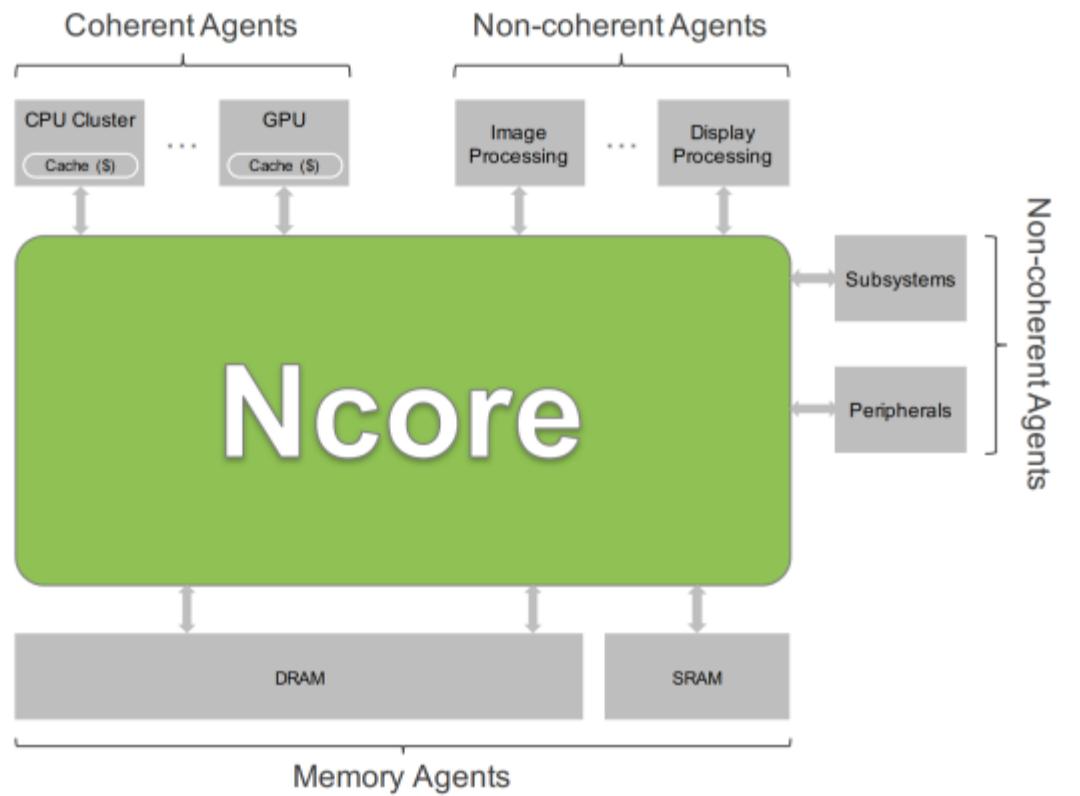
#### Supporting GTS in OS kernels (partly discussed)

##### bug不貴嗎

- 在2013年，[Samsung Exynos 5410](#) 上因為 CCI400 本身的 bug，造成產品 fail
- 5410 是世界上第一顆 4+4 核產品，用在 Galaxy S4 上，跑在最前面的自然就需要承擔風險
- CCI400 用來作不同 cluster 間的 cache coherence。
- Exynos 5410 只能作到 cluster migration，但無法兩個 cluster 都開啟，而這種 mode 的 overhead 是最大的，不能針對不同 task 精細分配執行的 cpu。CCI400 應該是 disable 掉 coherence function，使得資料要交換都要經過 dram，一次要搬移整個 cluster 的 data 量很大，data path 又很長，使得耗電和溫度表現很差。

- 六次 tape out 都抓不出問題，一次 tape out 多貴啊，後來只能再推出 5420 去取代原本的產品，而且造成一些 project 被取消掉。
- [Samsung Exynos 5 Octa 5420 looks to correct past mistakes, shoddy graphics](#)
  - Part of the reason is that the Exynos is hard to manufacture, but there was also a troublesome bug in the CCI-400 coherent bus interface. Developers noticed the bizarre behavior caused by this issue, and Samsung was eventually forced to admit that **coherence between the two CPU islands was disabled on the 5410. Basically, switching between the A15 and A7 cores caused all caches to be flushed from memory.** That's trouble for performance and battery life — both things big.LITTLE is supposed to improve.
- [魅族 MX3 深入拆解 與 GS4 不一樣的 5410?](#)
- 根據 XDA 論壇上的一種說法，Exynos 5410 的 CCI-400 有不可修復的 bug，因此在實際產品中是不工作的，由於這樣一個問題，如果系統同時啟用兩個 Cluster，它們之間將無法維持緩存一致，自然也就無法共同工作。否則不同處理器讀取到的同一個內存地址的數據不同，系統不崩潰就怪了。
- 當然，如果拋棄所有的緩存，這八個處理器還是可以同時工作的，但是那樣就得不償失了——八核的目的是為了快，而不是慢。另外一個網站，SAMMOBILE 對這個問題寫了一篇很長的文章，進一步闡述了這個問題：由於 CCI-400 存在問題，這個元件在 Exynos 5410 里甚至根本沒有供電。因此事實上它是不存在的。
- **Exynos 5410 前後經過了六次流片，最終依然讓這個 bug 成為了漏網之魚**，所以三星才這麼匆忙推出 Exynos 5420，終於用上了修復版的 CCI400——顯然，Mediatek 也是，這也是 Meditek 可以嘲笑三星的資本
- [Samsung's processor chaos affects Mobile divisions processor choice](#)
  - We have information from several sources that Exynos's CCI is inherently crippled in silicon. It is not functional or even powered on in the shipping product (i9500). In fact, this has been such of an issue, that as a result, **the chip was almost cancelled.**
  - Internally at SLSI, **as many as three projects were cancelled late last year.**
- [Samsung Updates Exynos 5 Octa \(5420\), Switches Back to ARM GPU](#)
  - Part of the problem with the design was a broken implementation of the CCI-400 coherent bus interface that connect the two CPU islands to the rest of the SoC. In the case of the 5410, the bus was functional but **coherency was broken and manually disabled** on the Galaxy S 4.
- [More Exynos 5420 Octa details revealed by Samsung](#)
  - In the Exynos 5410 implementation, the CCI-400 coherent bus interface is broken such that every time when **a application switches between the Cortex A15 and the Cortex A7, the information in the cache is flushed to the memory.** The wasted cycles to transfer data between memory and cache every time during a switch represent a huge power overhead.
- [\[Info\] Exynos Octa and why you need to stop the drama about the 8 cores \[Upd 25/05\]](#)
  - Reality is the Exynos 5410 has some serious issues with its cache coherent interconnect / CCI which cripples the chip to only cluster migration, effectively making the major parts of the big.LITTLE operating scheme useless. This is an issue in silicon which cannot be solved.
- [『S4 根本是個未完成就趕上市的產品』，XDA 的專家開發者 AndreiLux 如是說.....](#)

# Ncore Cache Coherent Interconnect IP



Copyright © 2016 Arteris

12



## Arteris Announces Ncore Cache-Coherent Interconnect

隨著半導體製程逼近物理極限，即便 CPU 本身運算能力提昇，很大比例的效能障礙卻來自記憶體，cache coherence 就是箇中關鍵議題。為了解決 ARM 多核心效能議題，Arteris 公司提出 Ncore Cache Coherent Interconnect (CCI)，允許 Mediatek 和 Samsung 一類的晶片實做者延展 Arteris 提出的 FlexNoC 架構，用更少的線路達到更多實體連接。

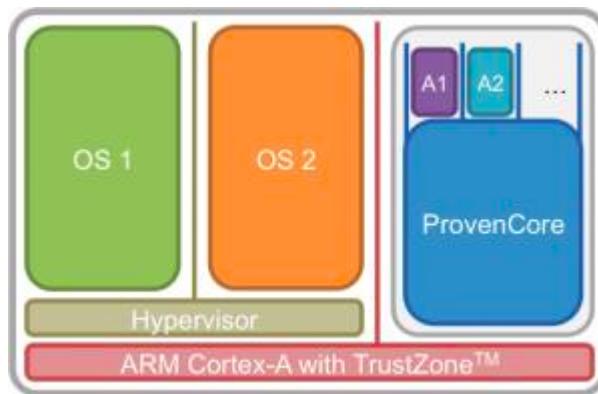
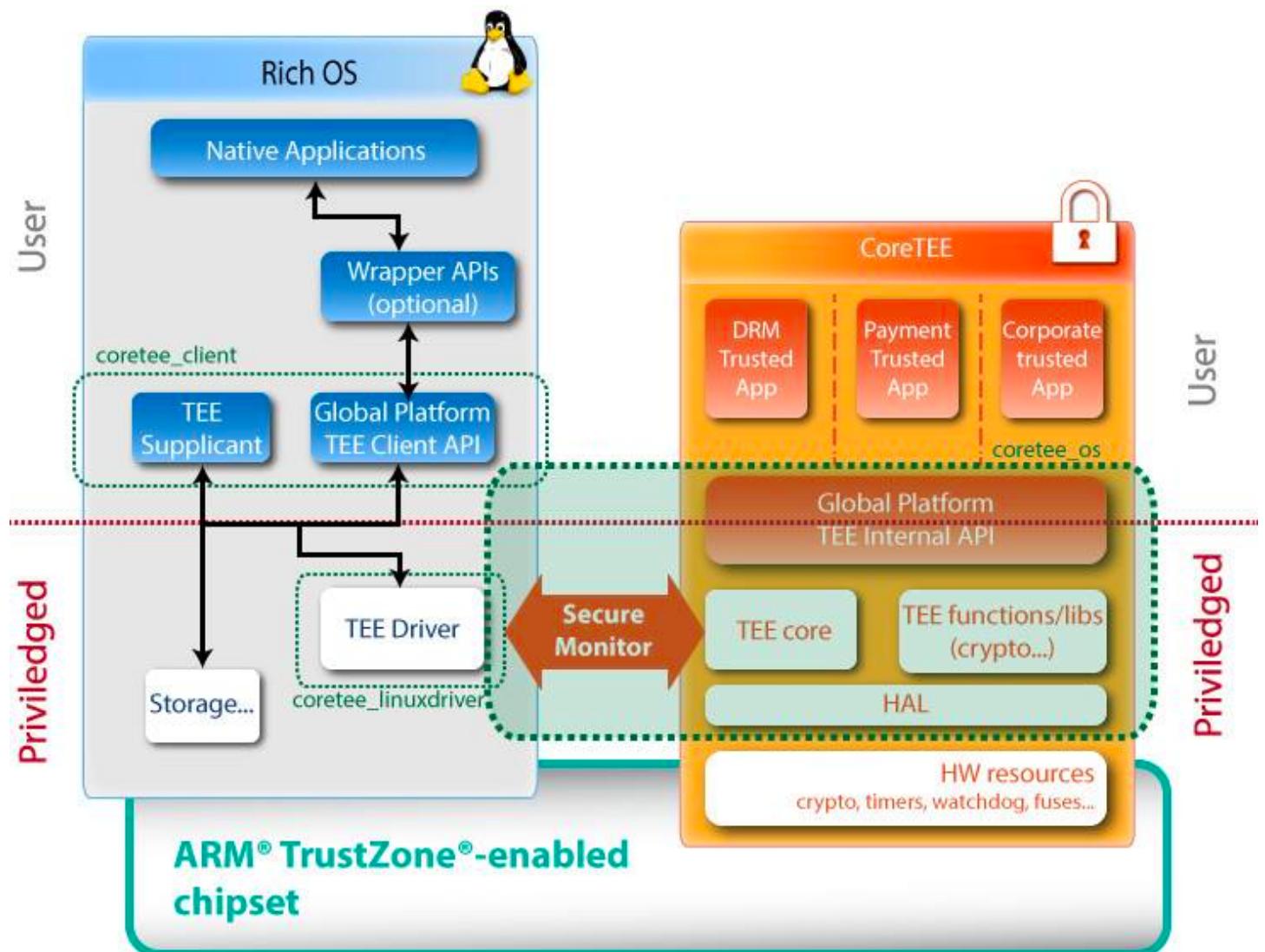
NoC (Network-on-Chip) 主要是用以改善現今主流的 crossbar + bus 作為硬體 IP 間的 interconnection 方式，優點當然是因 network 架構大幅減少的接線，因此降低了接線的 routing congestion (可減少 place and route 的時間成本)，進而可提高 area utilization(降低生產成本)，特別是晶片系統日驅複雜的今日，這些問題變逐漸被重視，但是 NoC 並非沒有缺點，像是 latency control 的問題。

若單以 IP 間 interconnection 的觀點看來，NoC 對於軟體的影響不大，就像是使用功能相同但實作不同的 software library. 但是若以系統發展來看，這就未必了. 做軟硬體系統整合，在接觸 SoC 有過一些特別應用情境經驗的人應該會知道，在一些系統頻寬比較吃緊狀況，偶有軟體需要針對不同應用去調整 bus arbiter 的 IP priority 的情況.

因此以這觀點來看在 NoC 平台上，為了進一步優化系統，可能必須要知道硬體 IP 提供的 Network topology，而針對應用提供應用上較佳的 NoC routing 設定或甚至提供 routing policy .

## Extension: TrustZone

### DAC 2016: Just how much security is enough?



### TrustZone 「TEE」 tech ported to Raspberry Pi 3

Sequitur Labs Inc. 將 Linaro OP-TEE 移植到 Raspberry Pi 3，這樣就可透過硬體的 TrustZone 機制，來隔離程式碼，從而確保安全的執行環境。

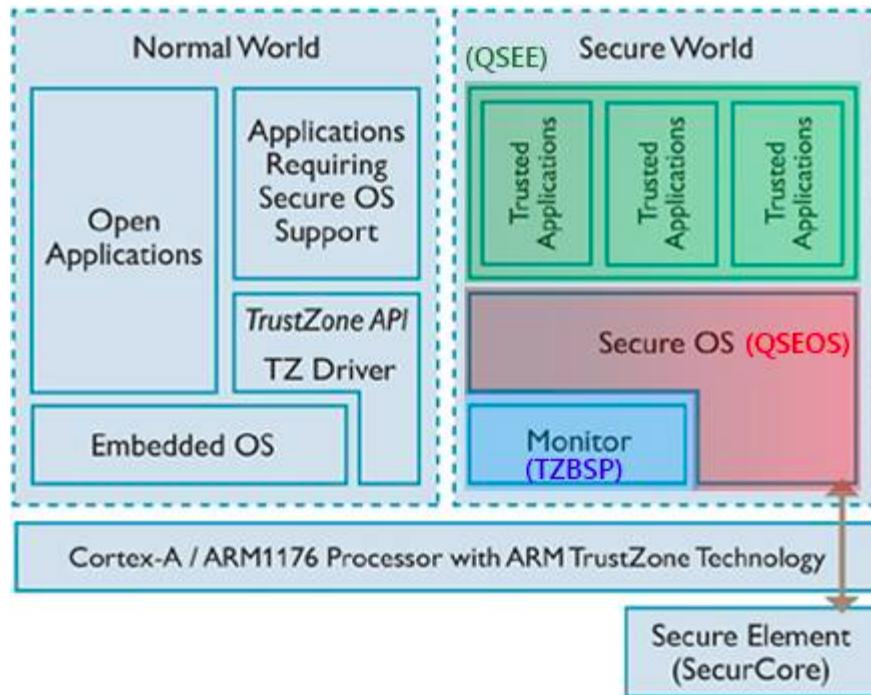
七月份這些程式碼會由 Linaro Security Working Group (SWG) 來維護。除了作到 software isolation 外，real-time system 也能透過 TrustZone 隔離特定高可靠要求的即時任務，從而達到 hard-RT。

法國的 Prove & Run 公司提出針對 ARM Cortex-A (VE + TrustZone) 和 Cortex-M3/M4/M7 的系統隔離解決方案，今年初在該公司的產品白皮書中，清楚提及若干應用情境：

<http://www.provenrun.com/about/proven-security-for-the-iot/>

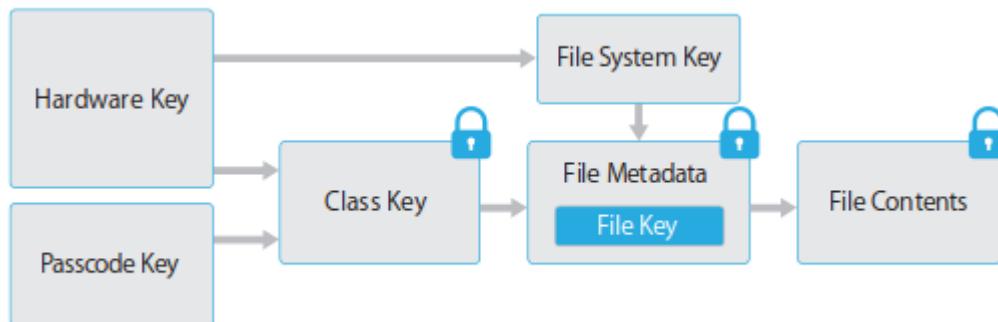
隨著車載電子陸續連接上網際網路後，安全議題則是走入各式實體裝置中，其中一個經驗的案例是 "BMW attack"，藉由對 BMW 的 ConnectedDrive 進行攻擊，證實這樣新型態的漏洞實際存在。

## ARM virtualization extension + TrustZone 在 ARMv8.1+ 是新架構更新的重要特徵



### Exploring Qualcomm's Secure Execution Environment

Qualcomm 的通訊晶片提供名為 "Secure Execution Environment" (Secure Core) 的執行單元，是 ARM TrustZone 技術的應用，在 Android 原始程式碼 hardware/qcom/keymaster 有相關界面。



### Extracting Qualcomm's KeyMaster Keys - Breaking Android Full Disk Encryption

Qualcomm 平台 TrustZone kernel 安全漏洞被揭露後，另一個深入的衝擊就是透過 TrustZone kernel 來打破 Android's Full Disk Encryption (FDE)！

Android FDE 建構在 Linux 核心的 dm-crypt 子系統，並透過 KeyMaster 模組來保護應用程式產生的 cryptographic keys，為了避免過程中被破解，KeyMaster 模組執行於和 Android 作業系統完全獨立的 Trusted Execution Environment (TEE)。這聽起來很理想，但是，一旦 TEE 出問題後，就會釀成一系列的悲劇。

十年前，ARM 併購位於挪威的繪圖晶片公司 Falanx，催生了 Mali GPU。

今日 Mali 已經是世界上輸出最多的 GPU，單在 2015 年即有 7.5 億個以 Mali 為架構的 SoC 被運送到全球各地。

## Extension: Virtualization

- An in-depth look into the ARM virtualization extensions

- ARM 推出 VE，是受到 Intel VT 的啟發，但設計得更細緻，並延續 TrustZone 的機制，在 VE 啟用後，主要分成三種運作模式：PL0 (user mode), PL1 (kernel mode), PL2 (HYP mode)

- Embedded Virtualization applied in Mobile Devices

### OKL4 Microvisor - Freescale Vybrid VF610

繼承 OKL4 Microvisor 資產 (Open Kernel Labs 賣給 General Dynamics 後，部分來自 UNSW/NICTA 的團隊分出來) 的新創公司 Cog Systems 展示在 NXP Vybrid 平台 (混合 ARM Cortex-A5 與 Cortex-M4，強調 security extension) 上執行 7 個虛擬化的 Linux guest。

### Boeing video

seL4 受到 DARPA 贊助並已用於波音公司開發的 Unmanned Little Bird (ULB) 戰鬥直昇機，這份影片展示建構在 eChronos RTOS 和 seL4 的高度安全飛行控制系統，體驗 21 世紀的技術吧！

Mission board 由 ARM Cortex-A15 硬體組成，執行 seL4 + 虛擬化的 Linux，而 Flight controller 由 ARM Cortex-M4F 硬體組成，運作 eChronos RTOS，詳情可

見：<http://ssrg.nicta.com.au/projects/TS/SMACCM/>

### Cybersecure flight systems

seL4 的重要應用領域是軍火工業，在美國 DARPA 的 High-Assurance Cyber Military Systems (HACMS) 中，Rockwell Collins 是主要的其中重要的軍火商，而 seL4 就充分使用。



### **Conditional execution**

結合 conditional execution 和設定 flags，可以不使用任何 branch 實現簡單的 if statements，由於少了 branches 所造成較多 cycles，且不會增加 code size，因此，使用這種方法可以避免 branch 帶來的 overhead 因而提升效率。

舉例來說：

```

if ( i < 10){
    c = i +'0';
}
else{
    c = i + 'A' -10;
}

```

用 conditional execution 和設定 flags 的方式寫成的組語如下：

```

CMP    i, #10
ADDLO  c, i, #'0'
ADDHS  c, i, #'A'-10

```

再舉另外一個例子：

```

if ( c=='a' || c=='e' || c=='i' || c=='o' || c=='u' ){
    vowel++;
}

```

其組語可寫成：

```

TEQ    c, #'a'
TEQNE  c, #'e'
TEQNE  c, #'i'
TEQNE  c, #'o'
TEQNE  c, #'u'
ADDEQ  vowel, vowel, #1

```

再看看另一個例子：

```

if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')){
    letter++;
}

```

其組語可寫成：

```

SUB    temp, c, #'A'
CMP    temp, c, #'Z'-#'A'
SUBHI  temp, c, #'a'
CMPIHI temp, #'z'-#'a'
ADDLS  letter, letter, #1

```

以 **Block copy** 為範例：

```

void bcopy(char *to, char *from, int n){
    while (n--)
        *to++ = *from++;
}

```

一般會用到很多個 branch 的方式撰寫，如下：

@ arguments: R0: to, R1: from, R2: n

bcopy:

TEQ R2, #0

BEQ end

loop:

SUB R2, R2, #1

LDRB R3, [R1], #1

STRB R3, [R0], #1

B bcopy

end:

MOV PC, LR

用conditional execution 和設定 flags 的方式的如下(可以明顯看到 branch 數還有code size減少):

@ arguments: R0: to, R1: from, R2: n

@ rewrite 「n--」 as 「--n>=0」

bcopy:

SUBS R2, R2, #1

LDRPLB R3, [R1], #1

STRPLB R3, [R0], #1

BPL bcopy

MOV PC, LR

再加上 unrolling 會變成(假設 n 是 4 的倍數):

@ arguments: R0: to, R1: from, R2: n

@ assume n is a multiple of 4; loop unrolling

bcopy:

SUBS R2, R2, #4

LDRPLB R3, [R1], #1

STRPLB R3, [R0], #1

BPL bcopy

MOV PC, LR

如果 n 是 16 的倍數呢:

@ arguments: R0: to, R1: from, R2: n

@ n is a multiple of 16;

bcopy:

SUBS R2, R2, #16

LDRPL R3, [R1], #4

STRPL R3, [R0], #4

```

LDRPL R3, [R1], #4
STRPL R3, [R0], #4
LDRPL R3, [R1], #4
STRPL R3, [R0], #4
LDRPL R3, [R1], #4
STRPL R3, [R0], #4
BPL bcopy
MOV PC, LR

```

@ arguments: R0: to, R1: from, R2: n

@ n is a multiple of 16;

bcopy:

```

SUBS R2, R2, #16
LDMPL R1!, {R3-R6}
STMPL R0!, {R3-R6}
BPL bcopy
MOV PC, LR

```

@ could be extend to copy 40 byte at a time

@ if not multiple of 40, add a copy\_rest loop

以 Search 為範例:

```

int main(void)
{
    int a[10]={7,6,4,5,5,1,3,2,9,8};
    int i;
    int s=4;
    for (i=0; i<10; i++)
        if (s==a[i]) break;
    if (i>=10) return -1;
    else return i;
}

```

其組語:

```

.section .rodata
.LC0:
.word 7
.word 6
.word 4
.word 5
.word 5
.word 1
.word 3
.word 2

```

```
.word 9
.word 8
.text
.global main
.type main, %function
```

main:

```
sub sp, sp, #48
adr r4, L9 @ =.LC0
add r5, sp, #8
ldmia r4!, {r0, r1, r2, r3}
stmia r5!, {r0, r1, r2, r3}
ldmia r4!, {r0, r1, r2, r3}
stmia r5!, {r0, r1, r2, r3}
ldmia r4!, {r0, r1}
stmia r5!, {r0, r1}
mov r3, #4
str r3, [sp, #0] @ s=4
mov r3, #0
str r3, [ sp, #4] @ i=0
```

loop:

```
ldr r0, [sp, #4] @ r0=i
cmp r0, #10 @ i<10?
bge end
ldr r1, [sp, #0] @ r1=s
mov r2, #4
mul r3, r0, r2
add r3, r3, #8
ldr r4, [sp, r3] @ r4=a[i]
teq r1, r4 @ test if s==a[i]
beq end
add r0, r0, #1 @ i++
str r0, [sp, #4] @ update i
b loop
```

end:

```
str r0, [ sp, #4]
cmp r0, #10
movge r0, #-1
add sp, sp, #48
mov pc, lr
```

我們可以拿掉不必要的 load/store、loop invariant，使用 addressing mode、conditional execution來做最佳化：

```
.section .rodata
.LC0:
.word 7
.word 6
.word 4
.word 5
.word 5
.word 1
.word 3
.word 2
.word 9
.word 8
.text
.global main
.type main, %function
```

main:

```
sub sp, sp, #48
adr r4, L9 @ =.LC0
add r5, sp, #8
ldmia r4!, {r0, r1, r2, r3}
stmia r5!, {r0, r1, r2, r3}
ldmia r4!, {r0, r1, r2, r3}
stmia r5!, {r0, r1, r2, r3}
ldmia r4!, {r0, r1}
stmia r5!, {r0, r1}
mov r1, #4
mov r0, #0
add r2, sp, #8
```

loop:

```
cmp r0, #10 @ i<10?
bge end
ldr r4, [r2, r0, LSL #2]
teq r1, r4 @ test if s==a[i]
addeq r0, r0, #1 @ i++
beq loop
```

end:

```
cmp r0, #10
movge r0, #-1
add sp, sp, #48
mov pc, lr
```

經過最佳化後，從 code size 從 22 words 變成 13 words。

NOTE: Day 1 結束前，提醒學員 Day 2 一早要隨堂測驗，給定不全的 bubble sort in ARM，然後補完對應的實作，Day 1 回家趕快學習 ARM 指令！

```
.code
.thumb

/*
Function Name : BubbleSort
Sorts an array using the bubble sort algorithm

C Prototype :
void BubbleSort(int A[], int cnt, bool asc)
: Where int A[] is the array to initialize
: cnt is the size of A, and asc is a flag to set the order.
Parameters : R0: Address of A
             : R1: cnt, number of elements
             : R2: asc, ascending order flag
Return value : None
*/

```

### BubbleSort:

```
PUSH {R3-R7, LR} ; Save Context in the registers we will use.
MOV R3,#+1 ; set swapped register as true
```

### OuterLoop:

```
CMP R3,#+1 ; check if swapped register is true
BNE FINISH ; if (swapped == false) -> finish
MOV R3,#+0 ; set swapped register as false
MOV R4,#+1 ; initialize for loop to start at i = 1
B InnerLoop ; go to the inner loop
```

### InnerLoop:

```
CMP R4,R1 ; check for more elements r4 -> i
           ; index of array
BNE Compare_Array ; if not at the end of the foor
                   ; loop compare elements in array
B OuterLoop ; once inner loop completes go
            ; back to outer loop
```

### Compare\_Array:

```

SUB R5,R4,#+1      ;; r5 <- i - 1
LDR R6,[R0,R5,LSL #+2]   ;; r6 <- A[r5]
LDR R7,[R0,R4,LSL #+2]   ;; r7 <- A[r4]
??                  ;; compare to see if
                     ;; ascending flag is set
BEQ Ascending
BNE Descending

```

Ascending:

```

CMP R6,R7      ;; compare A[i-1] > A[i]
B Swap_and_Increment

```

Descending:

```

CMP R7,R6      ;; compare A[i-1] < A[i]
B Swap_and_Increment

```

Swap\_and\_Increment:

```

BGT Swap    ;; if flags are set from
               ;; either Ascending or Descending labels
B Increment   ;; increment array index

```

Swap:

```

STR R6,[R0, R4, LSL #+2] ;; A[r6 ]-> @(r0 [r4 << 2]
                     ;; word shift
??                ;; A[r7] -> @(r0 [r5 << 2]
                     ;; word shift
MOV R3,#+1      ;; set swapped register as true
B Increment     ;; increment array index

```

Increment:

```

ADD R4,R4,#+1      ;; Next element number
B InnerLoop     ;; repeat

```

FINISH:

```

POP {R3-R7}      ;; restore context
MOV pc,lr       ;; return
END

```