探索嵌入式 ARM 平台與 SoC

Part II – 定址與組合語言瀏覽．硬體啓動程序．中斷與例外處理

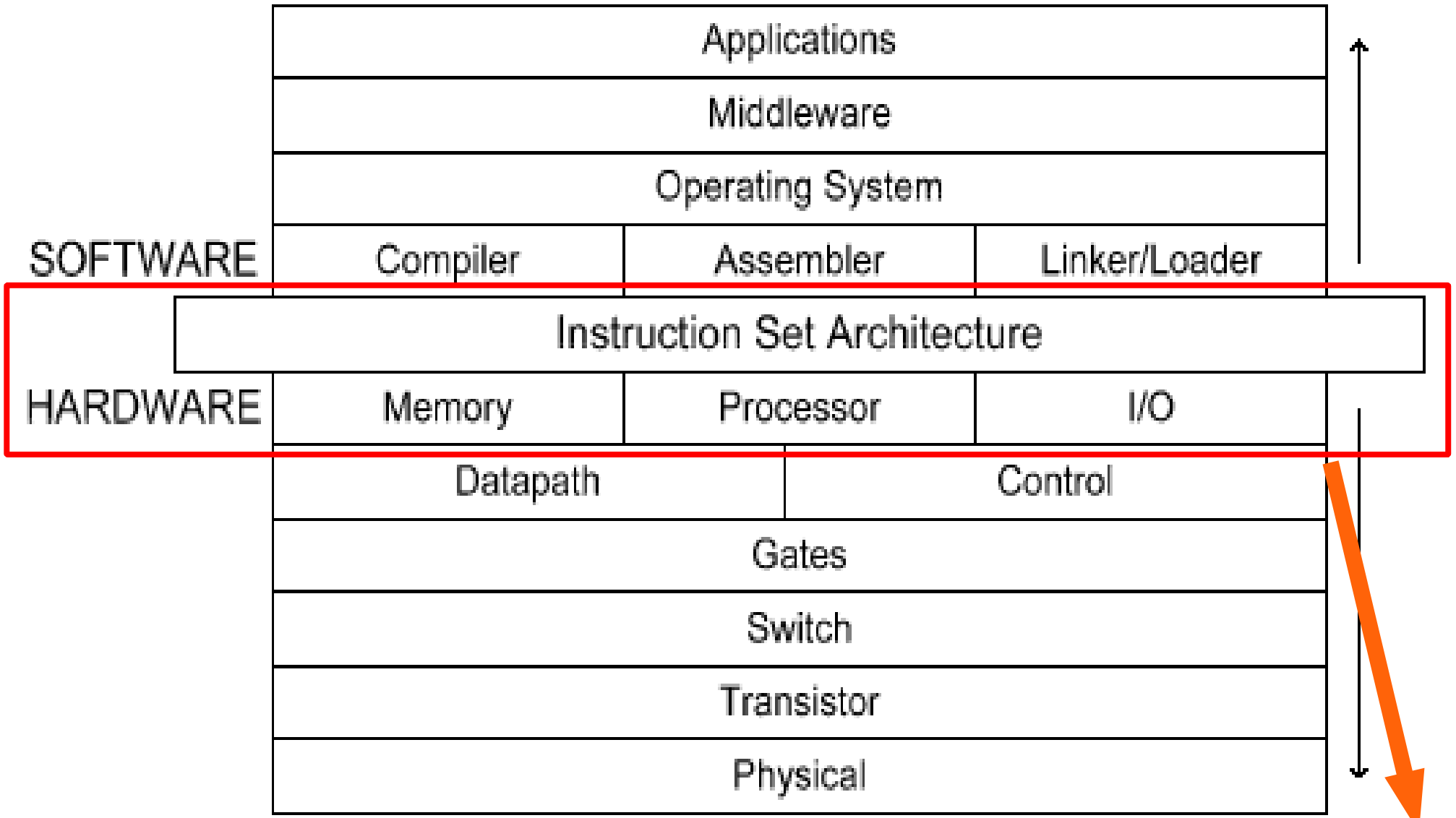
Jim Huang (**jserv**)
from **0xlab**

June 20, 2010

Rights to copy

contact@0xlab.org

Corrections, suggestions, contributions and translations are welcome!

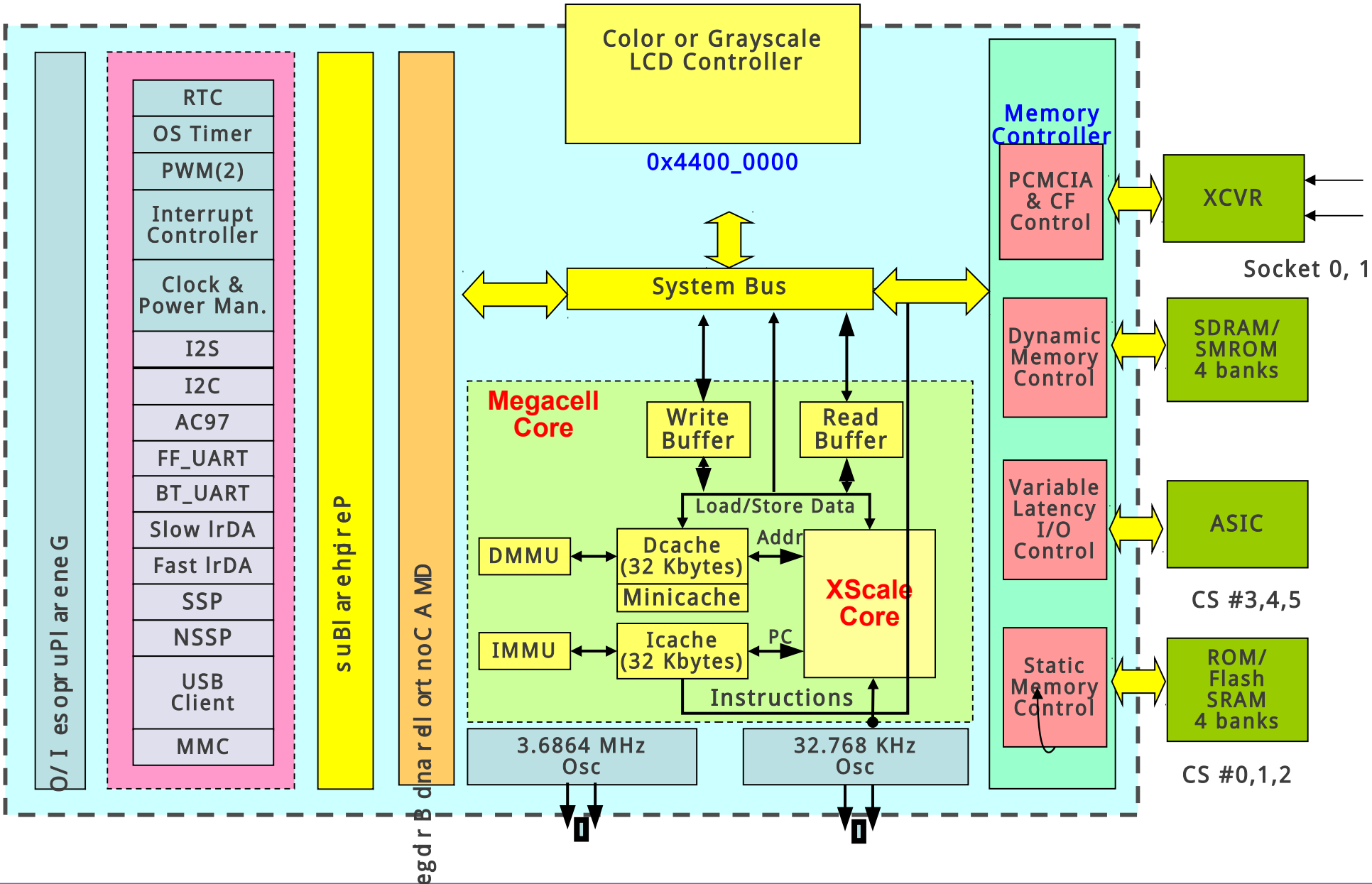

Applications
Middleware
Operating System
SOFTWARE
Compiler
Assembler
Linker/Loader
Instruction Set Architecture
HARDWARE
Memory
Processor
I/O
Datapath
Control
Gates
Switch
Transistor
Physical
Part II 涵蓋範圍

Agenda

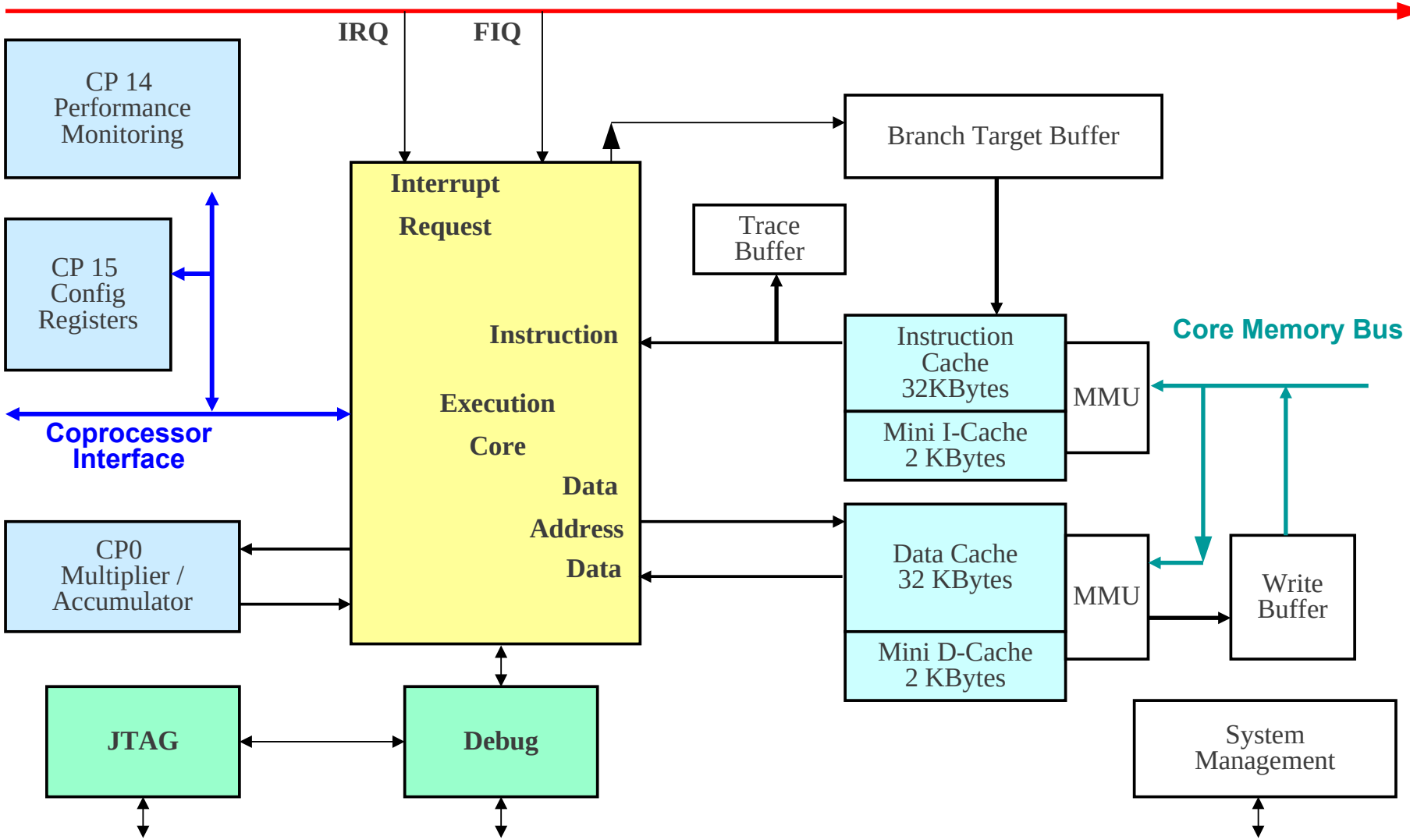
- PXA255 SoC 與 CuRT 的硬體啓動程序
- ARM Interrupt, ISR, Exception 的處理
- ARM 定址與組合語言概況

- PXA255 SoC 與 CuRT 的硬體啓動程序
- ARM Interrupt, ISR, Exception 的處理
- ARM 定址與組合語言概況

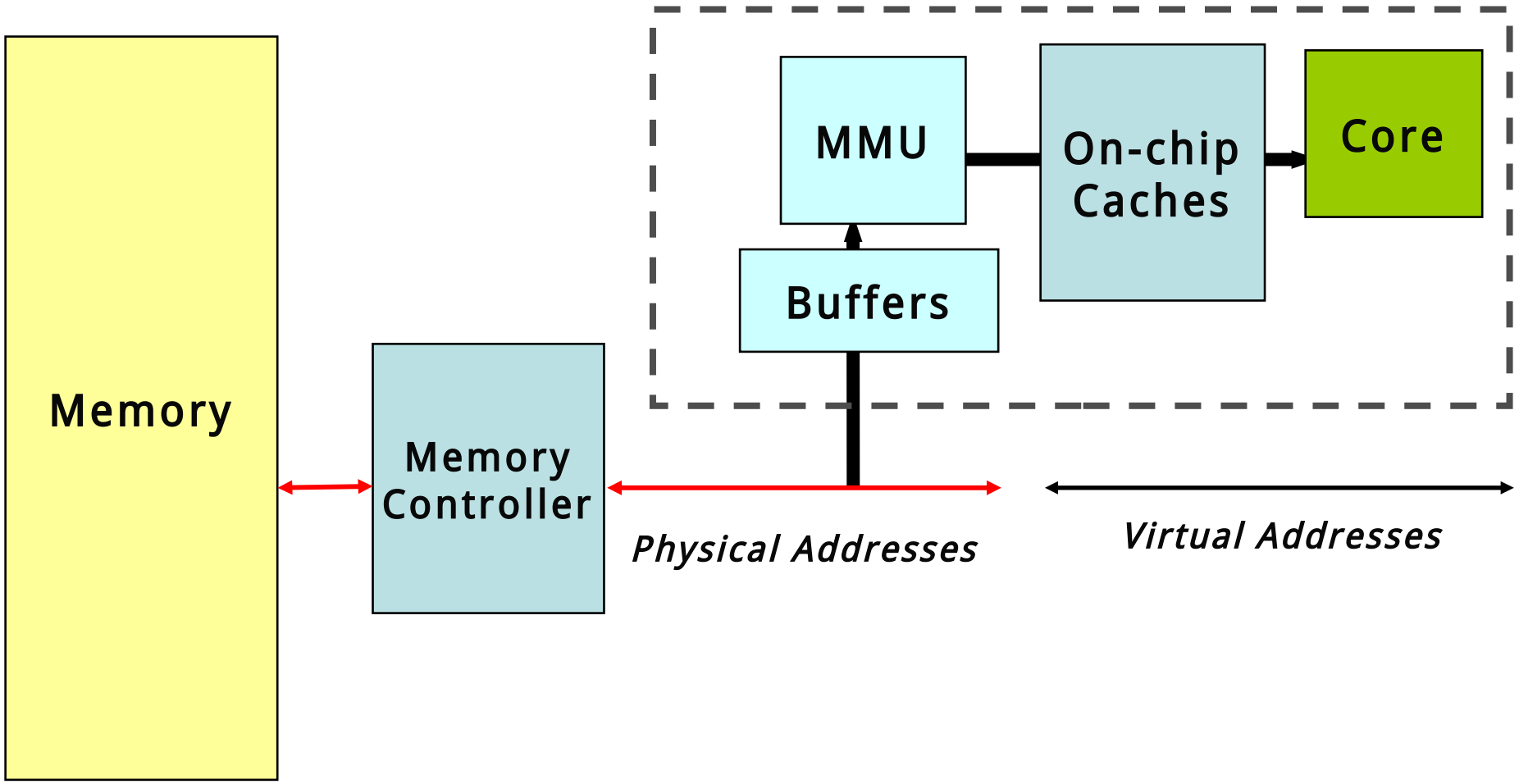
PXA255 Function Block



PXA255 系統架構

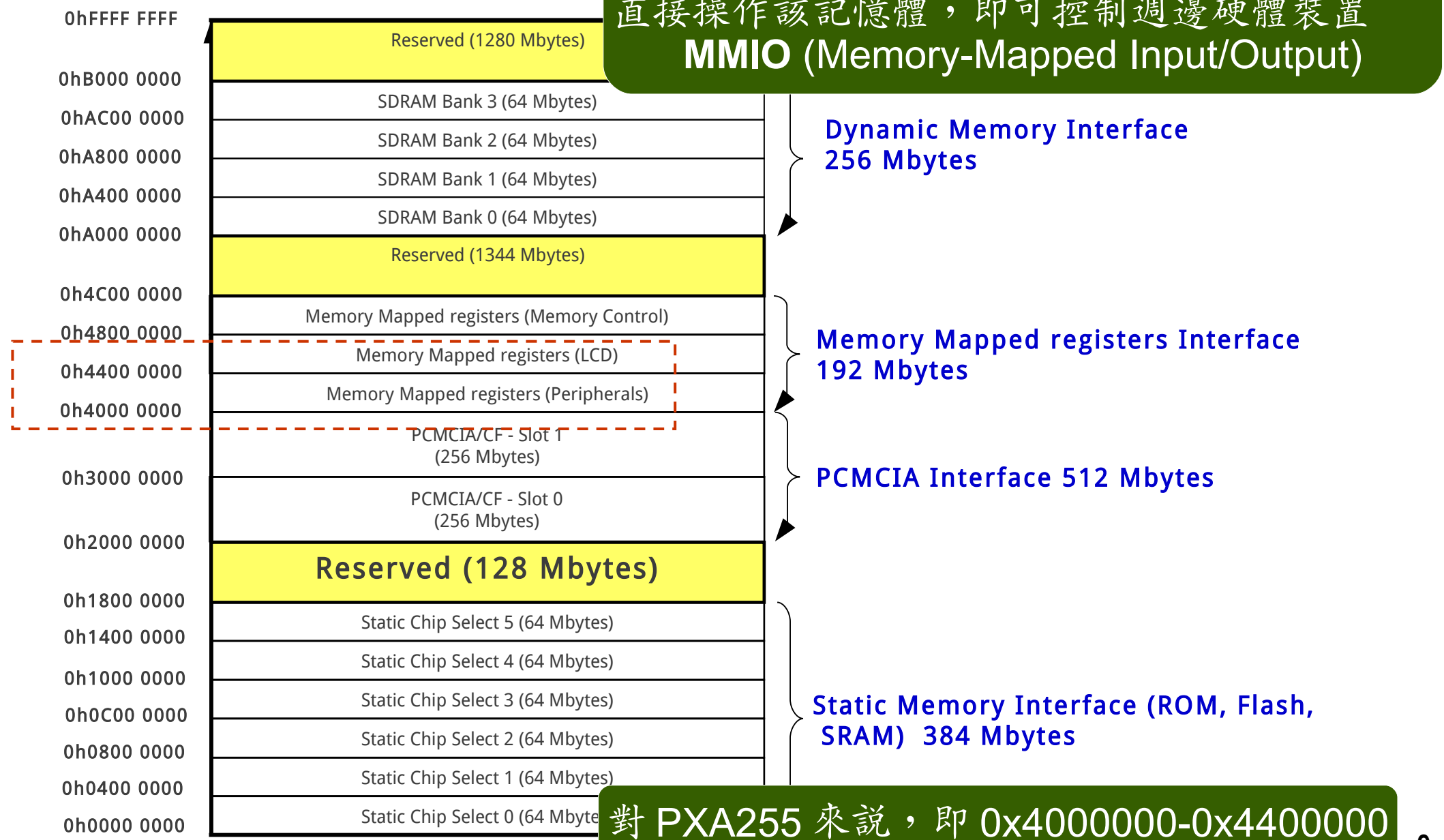

IRQ
FIQ
CP 14
Performance
Monitoring
CP 15
Config
Registers
Coprocessor
Interface
Interrupt
Request
Instruction
Execution
Core
Data
Address
Data
Branch Target Buffer
Trace
Buffer
Instruction
Cache
32KBytes
Mini I-Cache
2 KBytes
MMU
Core Memory Bus
CP0
Multiplier /
Accumulator
Data Cache
32 KBytes
Mini D-Cache
2 KBytes
MMU
Write
Buffer
JTAG
Debug
System
Management

PXA255 記憶體模型

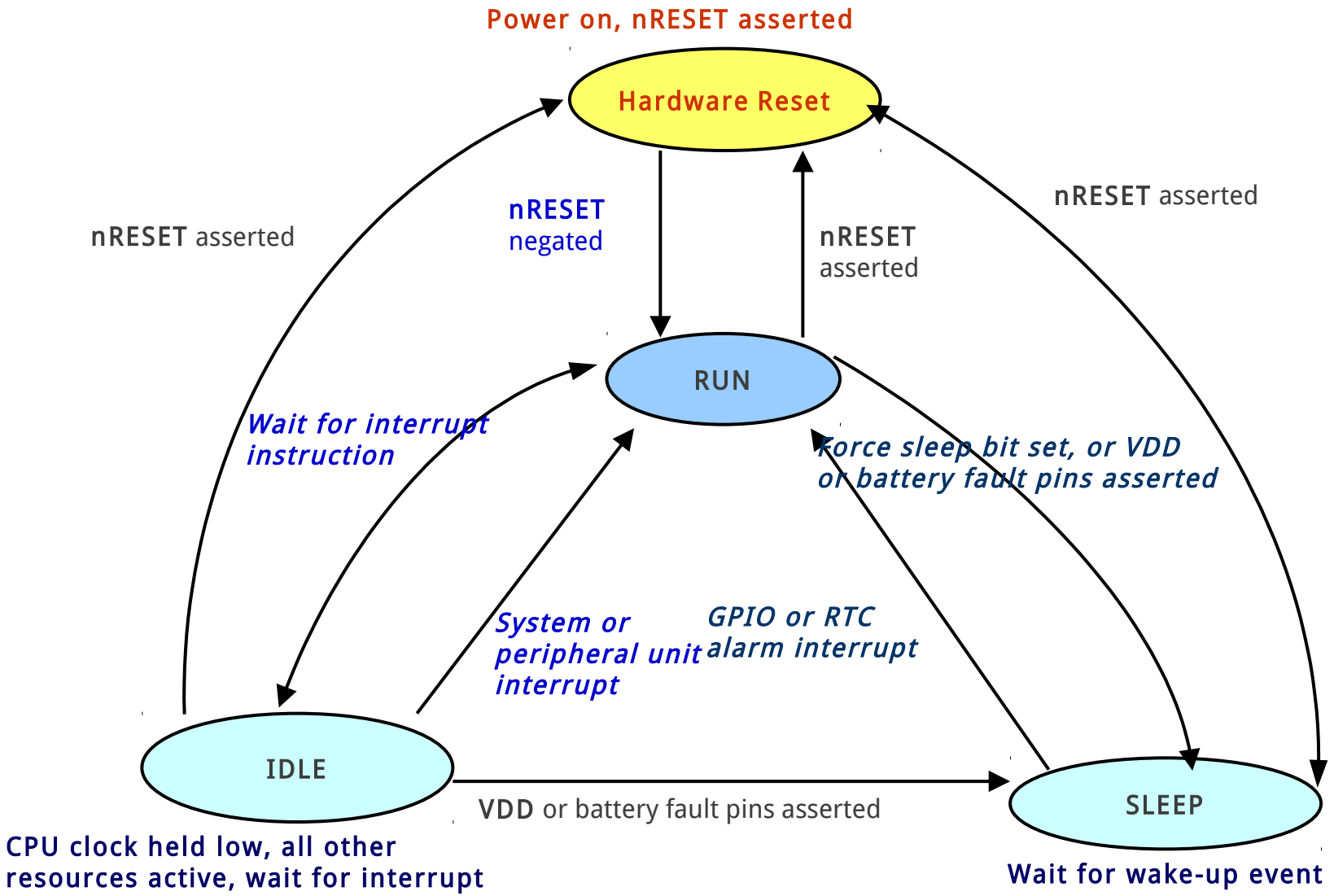


http://www.intel.com/design/pca/prodbref/252780docs.htm

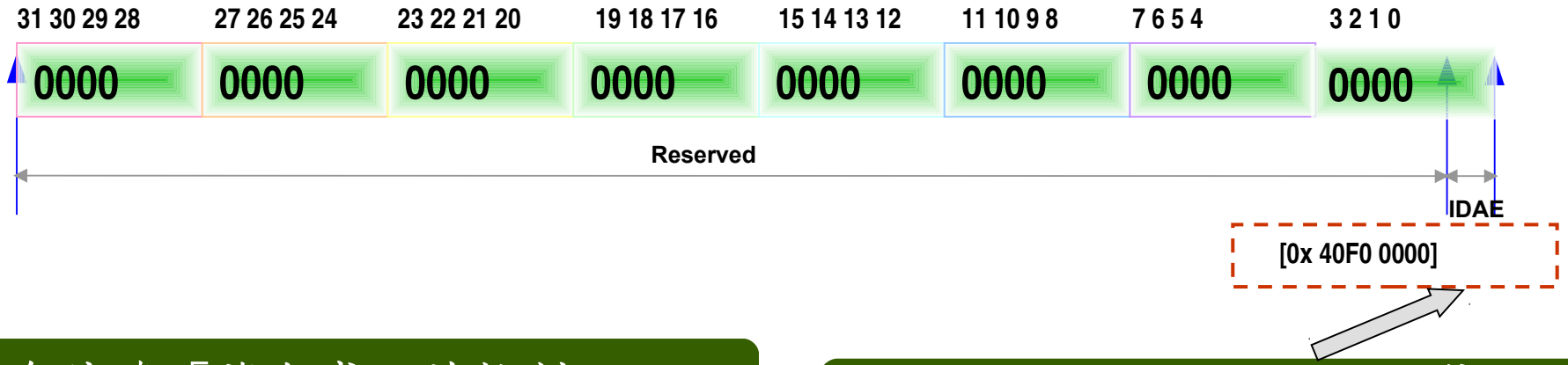
Memory Map


週邊 (peripheral) 被對應到特定的記憶體區段，直接操作該記憶體，即可控制週邊硬體裝置
MMIO (Memory-Mapped Input/Output)
0hFFFF FFFF
Reserved (1280 Mbytes)
0hB000 0000
SDRAM Bank 3 (64 Mbytes)
0hAC00 0000
SDRAM Bank 2 (64 Mbytes)
0hA800 0000
SDRAM Bank 1 (64 Mbytes)
0hA400 0000
SDRAM Bank 0 (64 Mbytes)
0hA000 0000
Reserved (1344 Mbytes)
0h4C00 0000
Memory Mapped registers (Memory Control)
0h4800 0000
Memory Mapped registers (LCD)
0h4400 0000
Memory Mapped registers (Peripherals)
0h4000 0000
PCMCIA/CF - Slot 1 (256 Mbytes)
0h3000 0000
PCMCIA/CF - Slot 0 (256 Mbytes)
0h2000 0000
Reserved (128 Mbytes)
0h1800 0000
Static Chip Select 5 (64 Mbytes)
0h1400 0000
Static Chip Select 4 (64 Mbytes)
0h1000 0000
Static Chip Select 3 (64 Mbytes)
0h0C00 0000
Static Chip Select 2 (64 Mbytes)
0h0800 0000
Static Chip Select 1 (64 Mbytes)
0h0400 0000
Static Chip Select 0 (64 Mbyte
0h0000 0000
Dynamic Memory Interface 256 Mbytes
Memory Mapped registers Interface 192 Mbytes
PCMCIA Interface 512 Mbytes
Static Memory Interface (ROM, Flash, SRAM) 384 Mbytes
對 PXA255 來說，即 0x4000000-0x4400000

PXA255 的執行模式



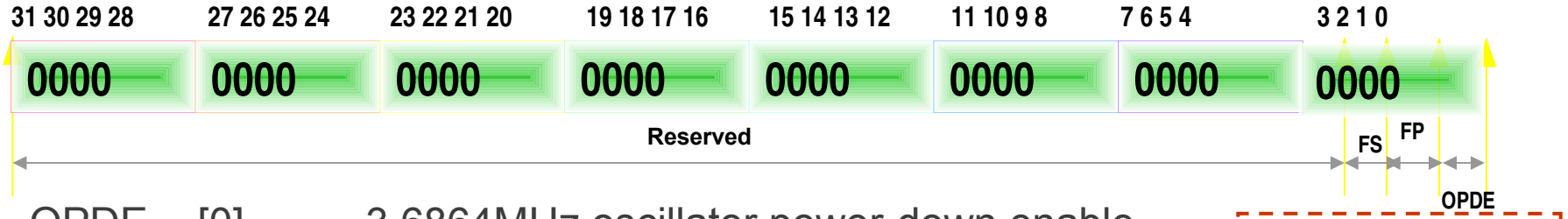
PMCR (Power Manager Control Register)



通常會伴隨「複合式」的控制 register

Memory-mapped 位址
同樣落於 0x4000000-0x4400000 的範圍

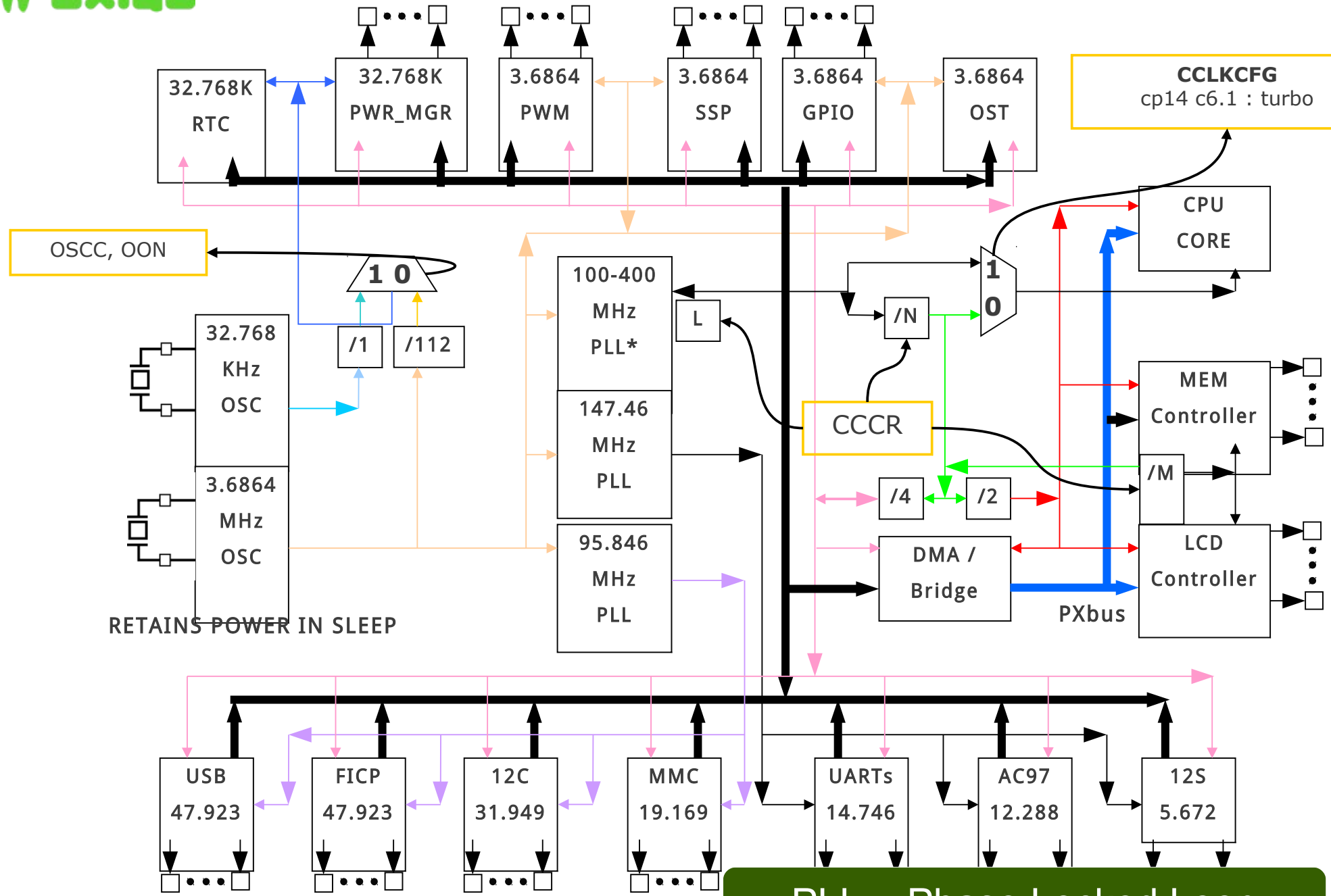
PCFR (Power Manager General Configuration Register)



OPDE	[0]	3.6864MHz oscillator power-down enable.
FP	[1]	Float PCMCIA controls during Sleep Mode.
FS	[2]	Float Static Chip Selects during Sleep Mode.

Memory-mapped 位址

PXA255 Processor Clock

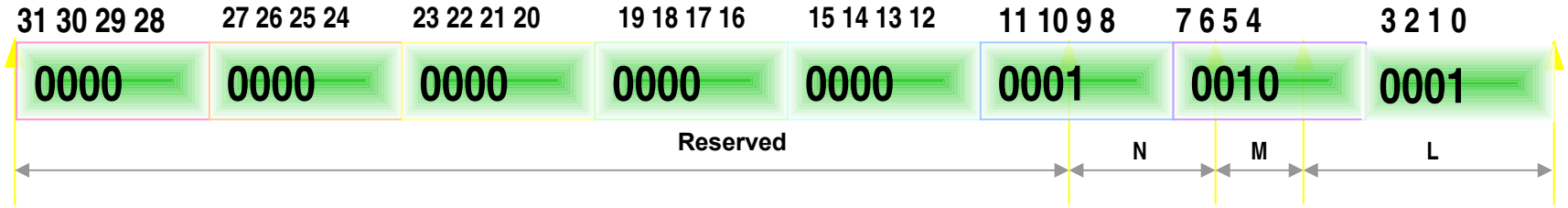


PLL = Phase Locked Loop

Core PLL Output Frequencies

L	M	Turbo Mode Frequency (MHz) for Values "N" and Core Clock Configuration Register (CCCR[15:0]) Programming for Values of "N":				PXbus Frequency	MEM, LCD Frequency (MHz)	SDRAM max Freq
		1.00 (Run)	1.50	2.00	3.00			
27	1	99.5 @1.0 V	—	199.1 @1.0 V	298.6 @1.1 v	50	99.5	99.5
36	1	132.7 @1.0 V	—	—	—	66	132.7	66
27	2	199.1 @1.0 V	298.6 @1.1 v	398.1 @1.3 V	—	99.5	99.5	99.5
36	2	265.4 @1.1 V	—	—	—	132.7	132.7	66
45	2	331.8 @1.3 V	—	—	—	165.9	165.9	83
27	4	398.1 @1.3 V	—		—	196	99.5	99.5

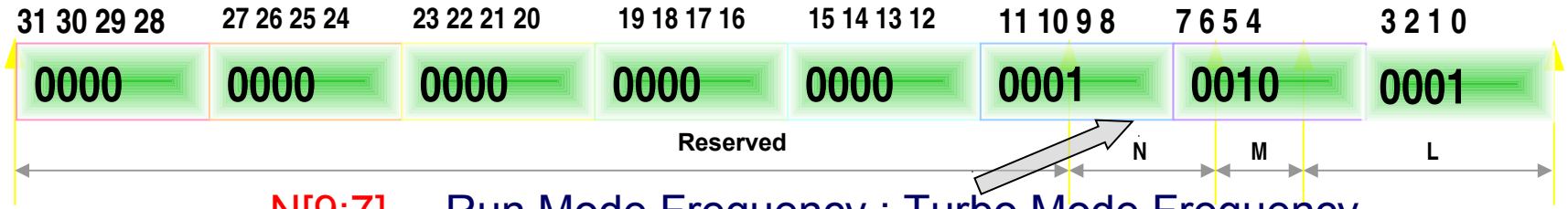
CCCR (Core Clock Configuration Register)



CuRT_v1/arch/arm/mach-pxa/start.S

```
init_clock_reg:         /* PXA250 Clock Register initialization */
        ldr r1, =CLK_BASE  /* base of clock registers */
        ldr r2, =0x00000241     /* memory clock: 100MHz,
                                   normal core clock: 200MHz,
                                   turbo mode: 400MHz */
        str r2, [r1, #CLK_CCCR]
```

CCCR (Core Clock Configuration Register)



N[9:7] Run Mode Frequency : Turbo Mode Frequency

Turbo Mode Freq. = Run Mode Frequency * N

000 , 001 , 101 , 111 – Reserved

001 (Multiplier) = 1

011 (Multiplier) = 1.5

100 (Multiplier) = 2

110 (Multiplier) = 3

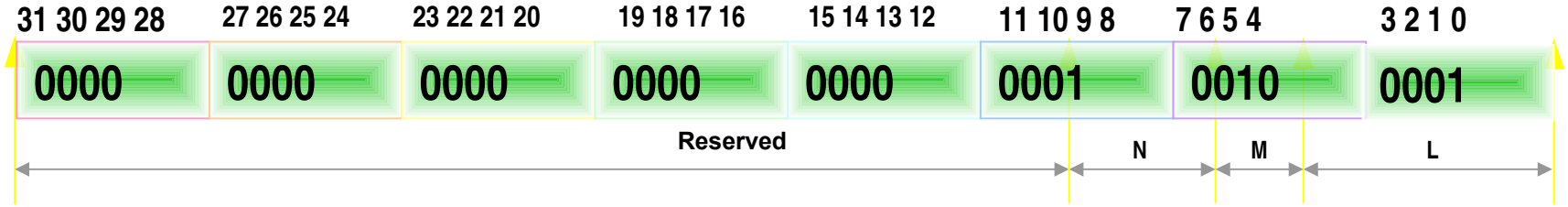
☞ Turbo Mode Freq(398.1MHz) = Run Mode Freq(199.1MHz) * N(2)

Address : 0x41300000

CuRT_v1/arch/arm/mach-pxa/start.S

```
init_clock_reg:         /* PXA250 Clock Register initialization */
        ldr r1, =CLK_BASE  /* base of clock registers */
        ldr r2, =0x00000241      /* memory clock: 100MHz,
                                    normal core clock: 200MHz,
                                    turbo mode: 400MHz */
        str r2, [r1, #CLK_CCCR]
```

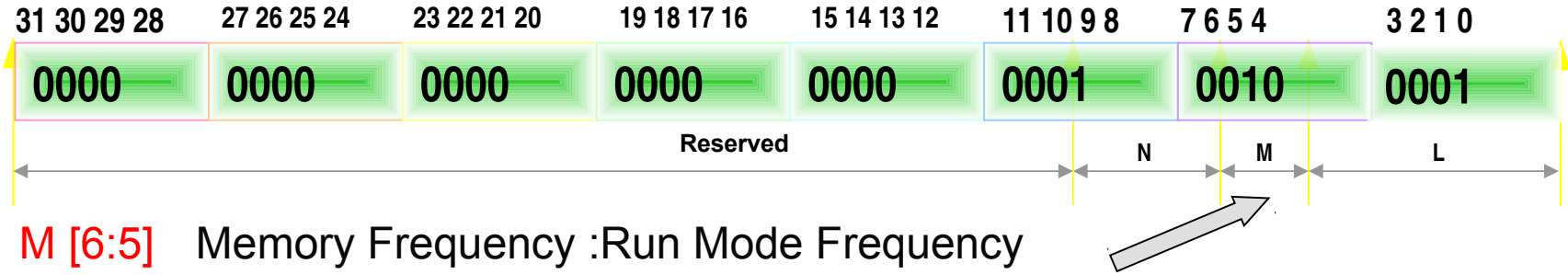
CCCR (Core Clock Configuration Register)



CuRT_v1/includes/arch/arm/mach-pxa/pxa255.h

```
/** Clocks Manager */
#define CLK_BASE 0x41300000
#define CLK_REG(_x_)    *(vulong *)(CLK_BASE + _x_)
#define CLK_CCCR 0x00   /* Core Clock Configuration Register */
#define CLK_CKEN 0x04   /* Clock Enable Register */
#define CLK_OSCC 0x08   /* Oscillator Configuration Register
```

CCCR (Core Clock Configuration Register)



M [6:5] Memory Frequency :Run Mode Frequency
Memory Freq = Crystal Frequency * L

00 , 11 – Reserved
01(Multiplier) = 1
10(Multiplier) = 2

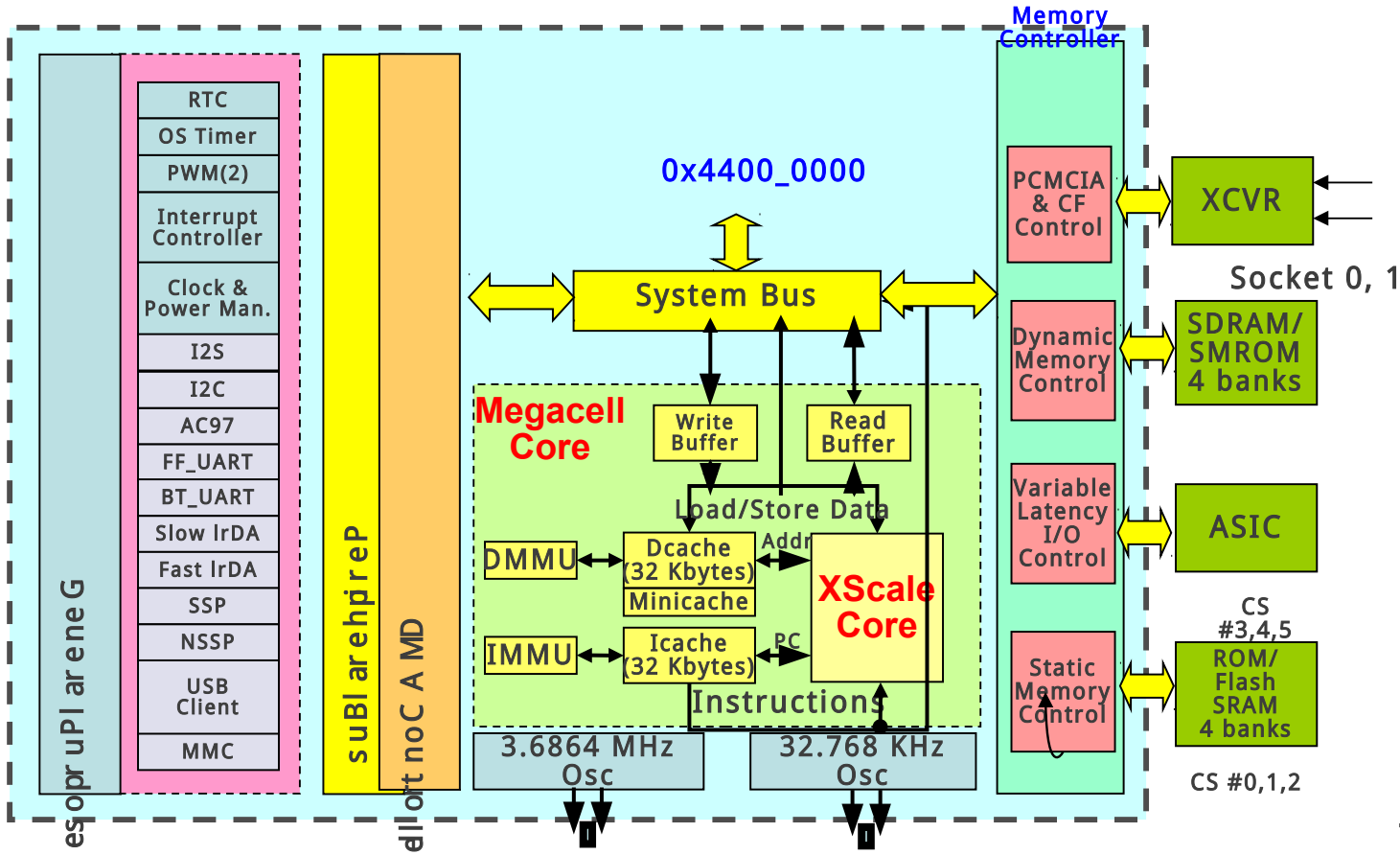
☞ Memory Freq(99.5MHz) = Crystal Frequency(3.6864MHz) * L(27)

L[4:0] Crystal Frequency: Memory Frequency (3.6864MHz Crystal)

00000 , 00110 to 11111 – Reserved
00001(Multiplier) = 27 (Memory Freq: 99.53MHz)
00010(Multiplier) = 32 (Memory Freq: 117.96MHz)
00011(Multiplier) = 36 (Memory Freq: 132.71MHz)
00100(Multiplier) = 40 (Memory Freq: 147.46MHz)
00101(Multiplier) = 45 (Memory Freq: 165.89MHz)

GPIO (General Purpose I/O)

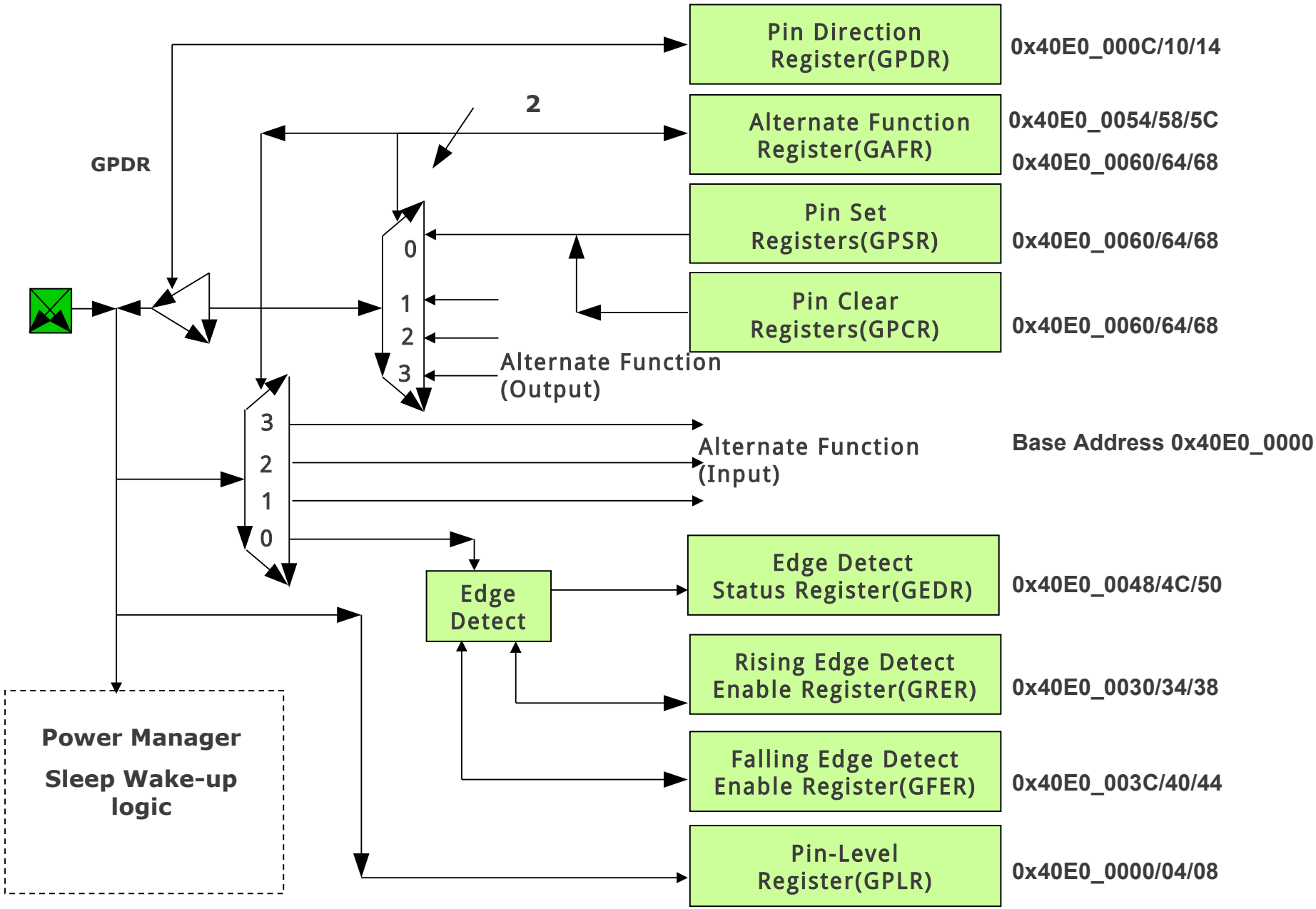
- Modem control signals for UART (CTS, RTS, CD, etc) implemented via GPIO signals
- GPIO[58:73] = dual panel color or 16 bit parallel input on LCD
- GPIO[23:27] = SPI if both synchronous serial protocols are required in a single system



GPIO

- GPIO Pin Direction Register (GPDR)
- GPIO Alternate Function Register (GAFR)
- GPIO Pin Set Register (GPSR)
- GPIO Pin Clear Register (GPCR)
- GPIO Falling Edge Detect Enable Register (GFER)
- GPIO Rising Edge Detect Enable Register (GRER)
- GPIO Edge Detect Status Register (GEDR)
- GPIO Pin Level Register (GPLR)

GPIO Block Diagram



CuRT_v1/arch/arm/mach-pxa/start.S

```
init_gpio:
        // FFUART

        ldr r12, =FFUART_BASE

        ldr r0, =0x00000000
        str r0, [r12, #FFLCR]

       …
        // First set the output values to a safe/disabled state
        // before we change any GPIO's outputs start by settings
        // all of them high which is the safest for most signals

        ldr r12, =GPIO_BASE

        ldr r0, =0xffffffff
        str r0, [r12, #GPIO_GPSR0]
        str r0, [r12, #GPIO_GPSR1]
        str r0, [r12, #GPIO_GPSR2]
```

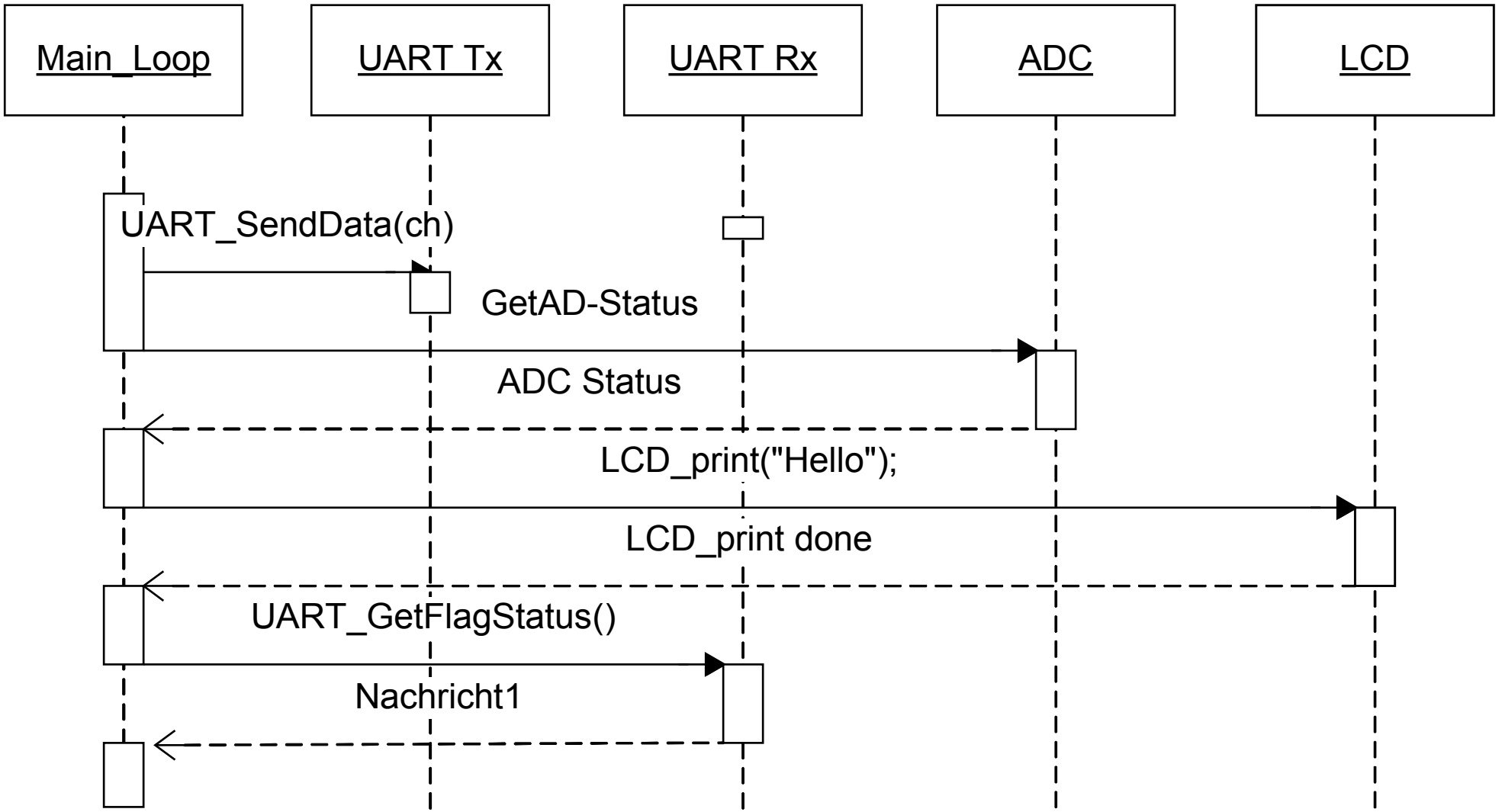
CuRT_v1/includes/arch/arm/mach-pxa/pxa255.h

```
/** General Programmable I/O */

#define GPIO_BASE 0x40E00000

#define GPIO_REG(_x_) *(volatile unsigned long *)(GPIO_BASE + _x_)
#define GPIO_GPLR0    0x00 /* GPIO<31: 0>   status register */
#define GPIO_GPLR1    0x04 /* GPIO<63:32>   status register */
#define GPIO_GPLR2    0x08 /* GPIO<80:64>   status register */
```

Register Type	Register Function	GPIO[15:0]	GPIO[31:16]	GPIO[47:32]	GPIO[63:48]	GPIO[79:64]	GPIO[80]
GPLR	Monitor Pin State	GPLR0		GPLR1		GPLR2	
GPSR	Control Output Pin State	GPSR0		GPSR1		GPSR2	
GPCR		GPCR0		GPCR1		GPCR2	
GPDR	Set Pin Direction	GPDR0		GPDR1		GPDR2	
GRER	Detect Rising/ Falling Edge	GRER0		GRER1		GRER2	
GFER		GFER0		GFER1		GFER2	
GEDR	Detect Edge Type	GEDR0		GEDR1		GEDR2	
GAFR	Set Alternate Functions	GAFR0_L	GAFR0_U	GAFR1_L	GAFR1_U	GAFR2_L	GAFR2_U

- PXA255 SoC 與 CuRT 的硬體啓動程序
- ARM Interrupt, ISR, Exception 的處理
- ARM 定址與組合語言概況

如果沒有 interrupt ，該會如何？



➔ The main() function executes all peripheral calls in a fixed sequence

interrupt 的定義與特性

- Interrupts are asynchronous events that may happen any time
- Interrupts stop the execution of the current task
  - The processor jumps into the interrupt service routine (ISR)
  - The **short** ISR is executed
  - Control is given back to the previously executing task
- Interrupts may have priorities.
- Concurrent interrupts (interrupts that happen at the same time) are serviced according to their priority
- Interrupts may be enabled or disabled
- Library functions that may be executed by an ISR must be thread-safe (they have to adhere to some specific rules)
- An ISR should if possible not trigger another interrupt

ARM Interrupt Controller

- All interrupts routed to FIQ or IRQ
- Two level interrupt structure
  - What module caused interrupt
    - Serial channel, DMA, Power Management, etc
  - Why did an interrupt occur there?
    - RX, TX, over-run, under-run, Data Done, Battery Fault, etc
- Template for servicing interrupts provided with firmware
- Peripheral/PCMCIA interrupt mask in each module
- GPIO masks determined per pin or group of pins

ARM Interrupt

- Vector table
  - Reserved area of 32 bytes at the end of the memory map
  - One word of space for each exception type
  - Contains a Branch or Load PC instruction for the exception handler
- Exception modes and registers
  - Handling exceptions changes program from user to non-user mode
  - Each exception handler has access to its own set of registers

CuRT_v1/arch/arm/mach-pxa/start.S

```
    /* exception handler vector table */
_start:
    b reset_handler
    b und_handler
    b swi_handler
    b abt_pref_handler
    b abt_data_handler
    b not_used
    b irq_handler
    b fiq_handler
```

Exception	Description
Reset	Occurs when the processor reset pin is asserted. This exception is only expected to occur for signalling power-up, or for resetting as if the processor has just powered up. A soft reset can be done by branching to the reset vector (0x0000).
Undefined Instruction	Occurs if neither the processor, or any attached coprocessor, recognizes the currently executing instruction.
Software Interrupt (SWI)	This is a user-defined synchronous interrupt instruction. It allows a program running in User mode, for example, to request privileged operations that run in Supervisor mode, such as an RTOS function.
Prefetch Abort	Occurs when the processor attempts to execute an instruction that was not fetched, because the address was illegal[a].
Data Abort	Occurs when a data transfer instruction attempts to load or store data at an illegal address[a].
IRQ	Occurs when the processor external interrupt request pin is asserted (LOW) and the I bit in the CPSR is clear.
FIQ	Occurs when the processor external fast interrupt request pin is asserted (LOW) and the F bit in the CPSR is clear.

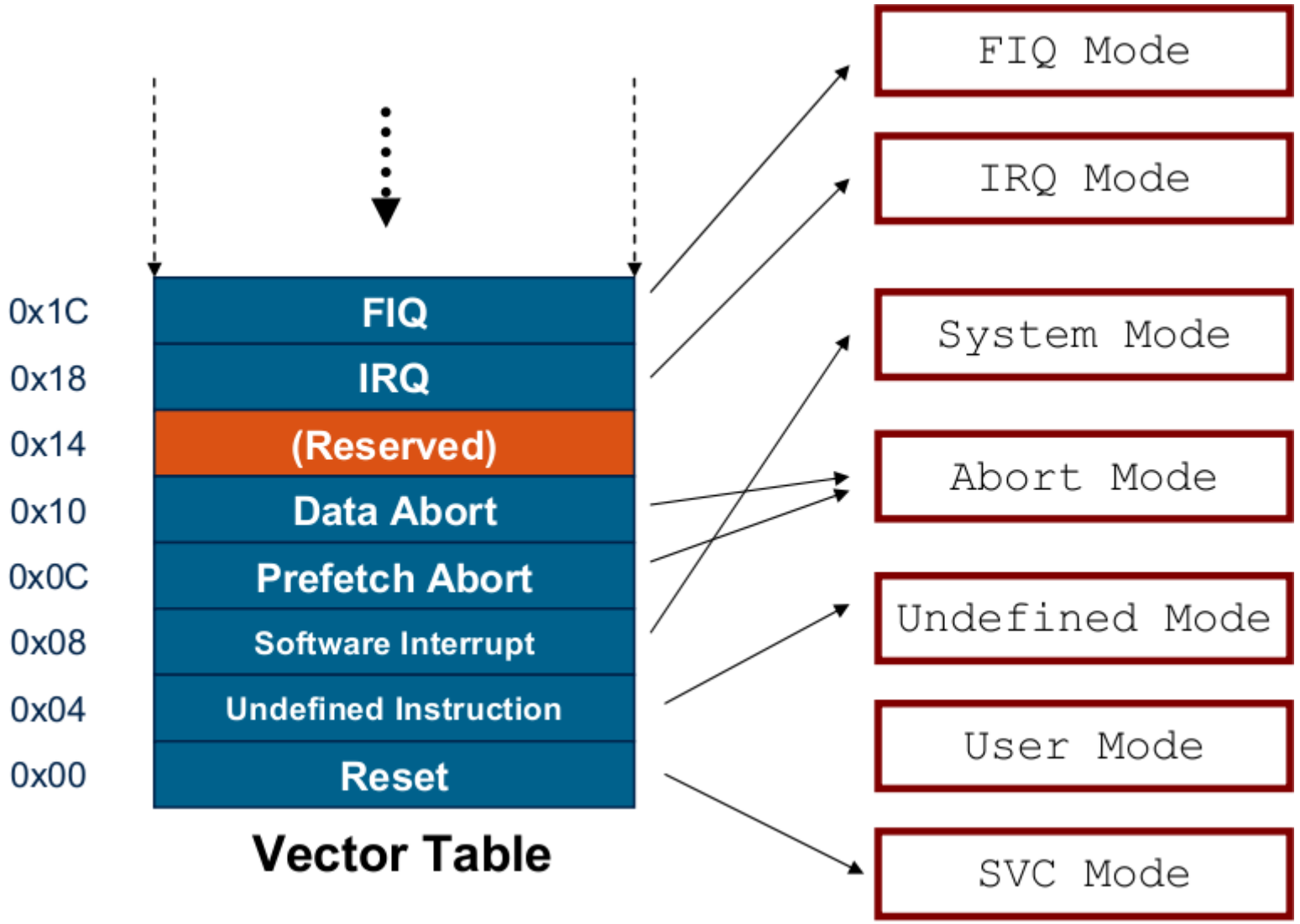
CuRT_v1/arch/arm/mach-pxa/start.S

```
    /* exception handler vector table */
_start:
    b reset_handler
    b und_handler
    b swi_handler
    b abt_pref_handler
    b abt_data_handler
    b not_used
    b irq_handler
    b fiq_handler
```

如果 Exceptions 同時發生，會如何？

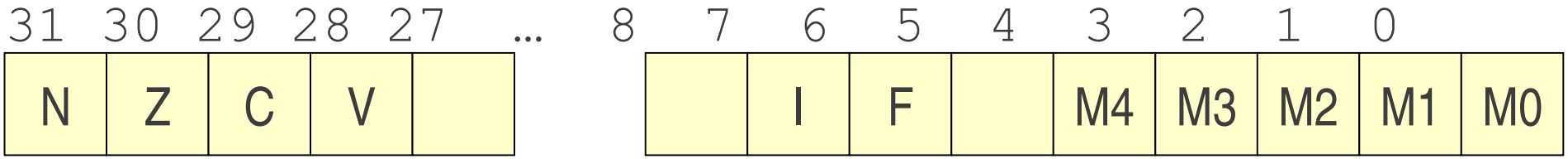
Vector address	Exception type	Exception mode	Priority (1=high, 6=low)
0x0	Reset	Supervisor (SVC)	1
0x4	Undefined Instruction	Undef	6
0x8	Software Interrupt (SWI)	Supervisor (SVC)	6
0xC	Prefetch Abort	Abort	5
0x10	Data Abort	Abort	2
0x14	*Reserved*	*Not applicable*	*Not applicable*
0x18	Interrupt (IRQ)	Interrupt (IRQ)	4
0x1C	Fast Interrupt (FIQ)	Fast Interrupt (FIQ)	3

ARM Exception and Modes



IRQ 與 FIQ

- Program Status Register



  - 若要抑制 interrupts ，將 "F" 或 " I" bit 設定爲 1

- 一旦 interrupt 觸發，處理器將變更至 FIQ32_mode registers 或 IRQ32_mode registers
  - Switch register banks
  - Copies CPSR to SPSR_mode (saves mode, interrupt flags, etc.)
  - Changes the CPSR mode bits (M[4:0])
  - Disables interrupts
  - Copies PC to R14_mode (to provide return address)
  - Sets the PC to the vector address of the exception handler

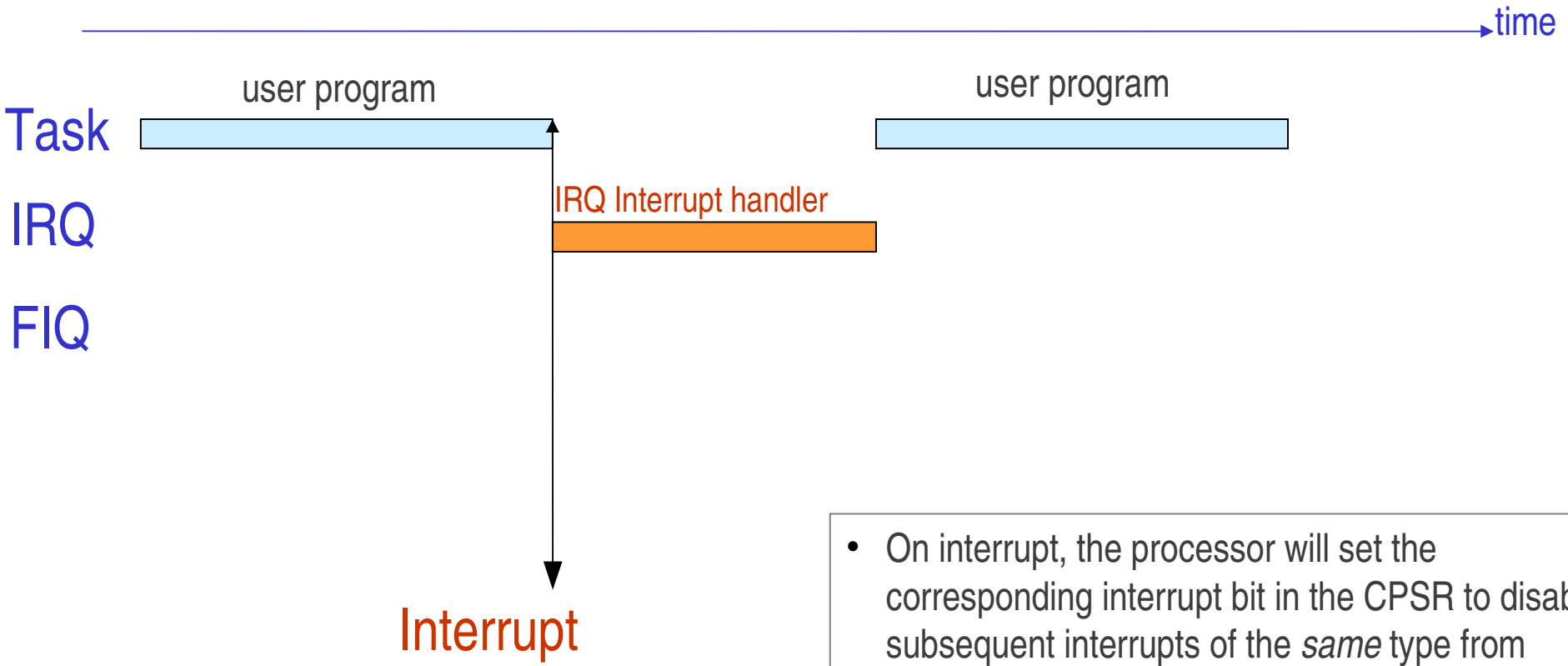
CuRT_v1/arch/arm/mach-pxa/arm_port.S

```
irq_service_routine:
    msr CPSR_c, #(NO_INT | IRQ32_MODE)
    stmfd sp!, {r1-r3}     // push working registers onto IRQ stack
    mov r1, sp          // save IRQ stack pointer
    add sp, sp, #12            // adjust IRQ stack pointer
    sub r2, lr, #4          // adjust pc for return

    mrs r3, SPSR            // copy SPSR (interrupted thread's CPSR)
    msr CPSR_c, #(NO_INT | SVC32_MODE)    // change to SVC mode

                  // save thread's context onto thread's stack
    stmfd sp!, {r2}            // push thread's return pc
    stmfd sp!, {lr}            // push thread's LR
    stmfd sp!, {r4-r12}        // push thread's r12-r4

    ldmfd r1!, {r4-r6}     // move thread's r1-r3 from IRQ stack to
                  // SVC stack
    stmfd sp!, {r4-r6}
    stmfd sp!, {r0}            // push thread's r0 onto thread's stack
    stmfd sp!, {r3}            // push thread's CPSR(IRQ's SPSR)

    bl enter_interrupt
    ...
```

- 一旦interrupt觸發，處理器將變更至FIQ32_mode registers 或IRQ32_mode registers
  - Switch register banks
  - Copies CPSR to SPSR_mode (saves mode, interrupt flags, etc.)
  - Changes the CPSR mode bits (M[4:0])
  - Disables interrupts
  - Copies PC to R14_mode (to provide return address)
  - Sets the PC to the vector address of the exception handler

Interrupt Handlers

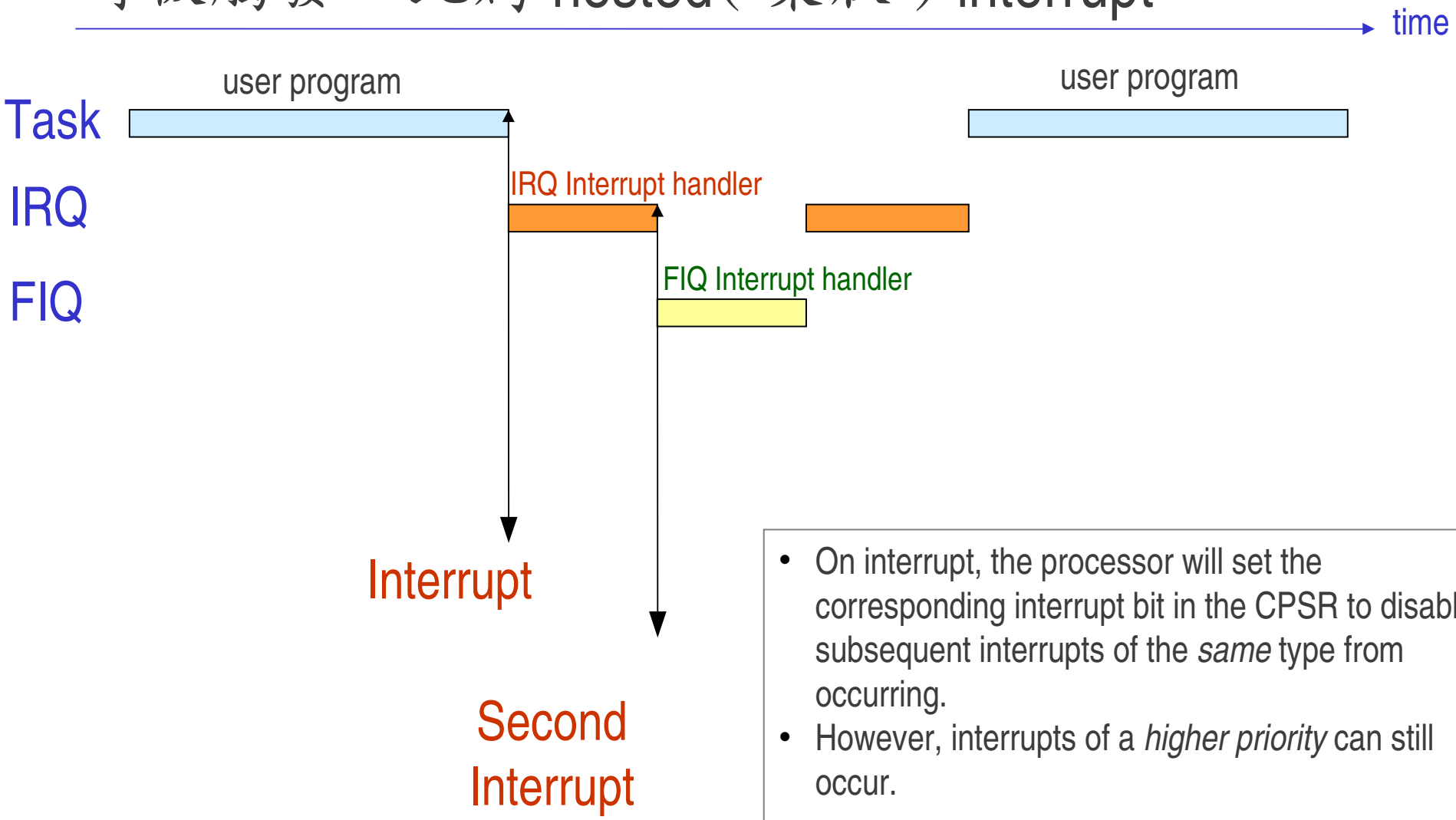
- 當 interrupt 發生時，硬體會跳躍到 interrupt handler



- On interrupt, the processor will set the corresponding interrupt bit in the CPSR to disable subsequent interrupts of the *same* type from occurring.
- However, interrupts of a *higher priority* can still occur.

Nested/Re-entrant Interrupts

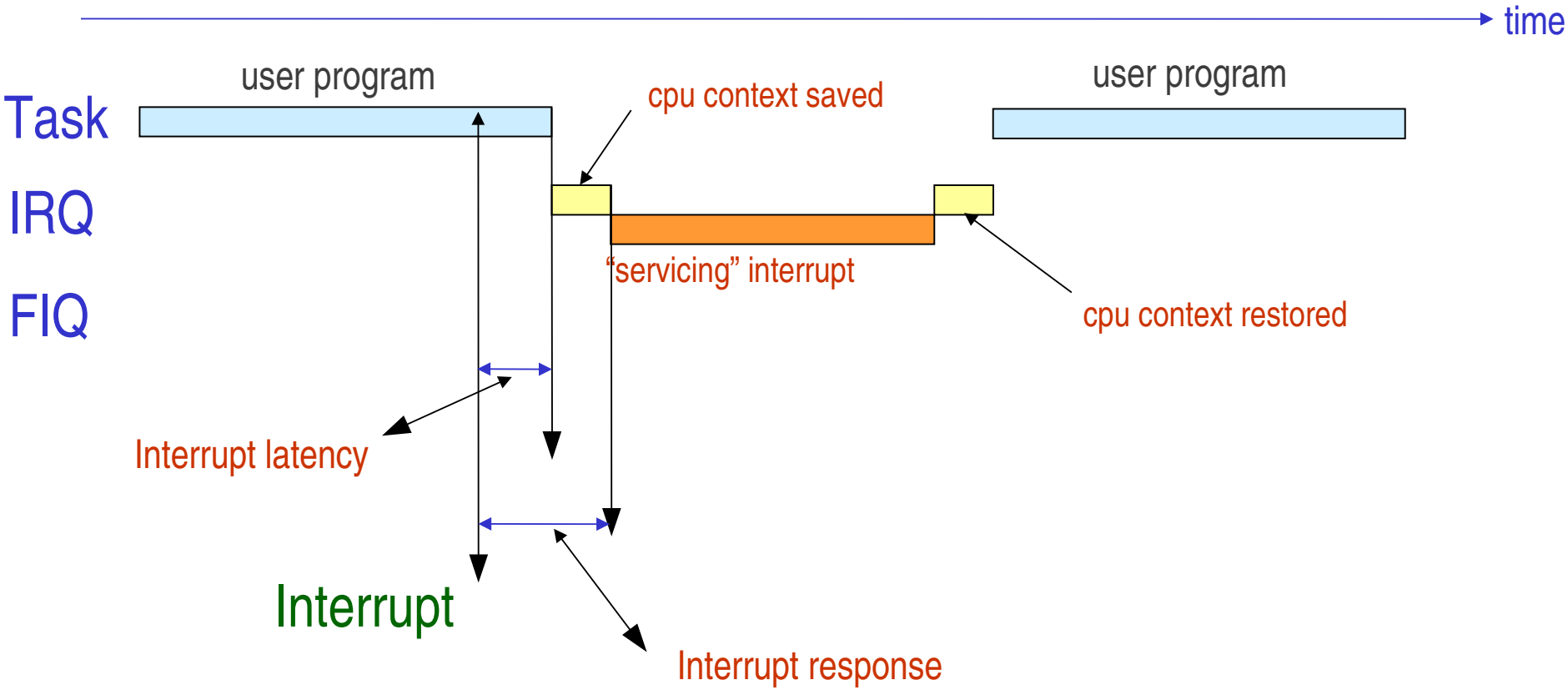
- 但是， interrupts 也可能在執行 interrupt handlers 時被觸發，此爲 nested(巢狀) interrupt



- On interrupt, the processor will set the corresponding interrupt bit in the CPSR to disable subsequent interrupts of the *same* type from occurring.
- However, interrupts of a *higher priority* can still occur.

Interrupts 的時序

- 在 interrupt handler 實際運作前，必須保存目前程式 (context) 的 register (若觸及這些 register)
- 這也是何以 FIQ 需要額外 register 的緣故，爲了降低 CPU 保存 context 的成本開銷



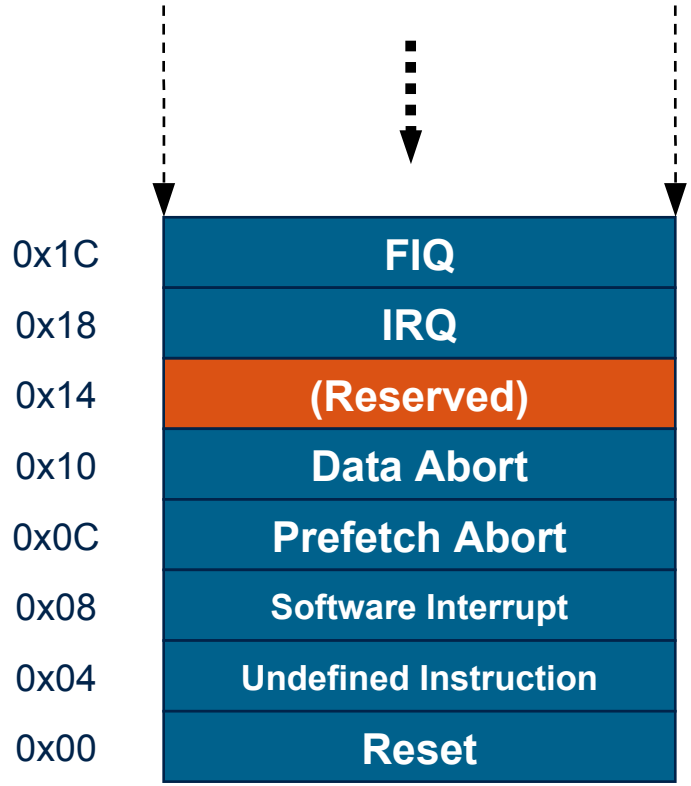
Exception Handling

- When an exception occurs, the ARM:
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits
    - Change to ARM state
    - Change to exception mode
    - Disable interrupts (if appropriate)
  - Stores the return address in LR_<mode>
  - Sets PC to vector address

To return, exception handler needs to:

- Restore CPSR from SPSR_<mode>
- Restore PC from LR_<mode>

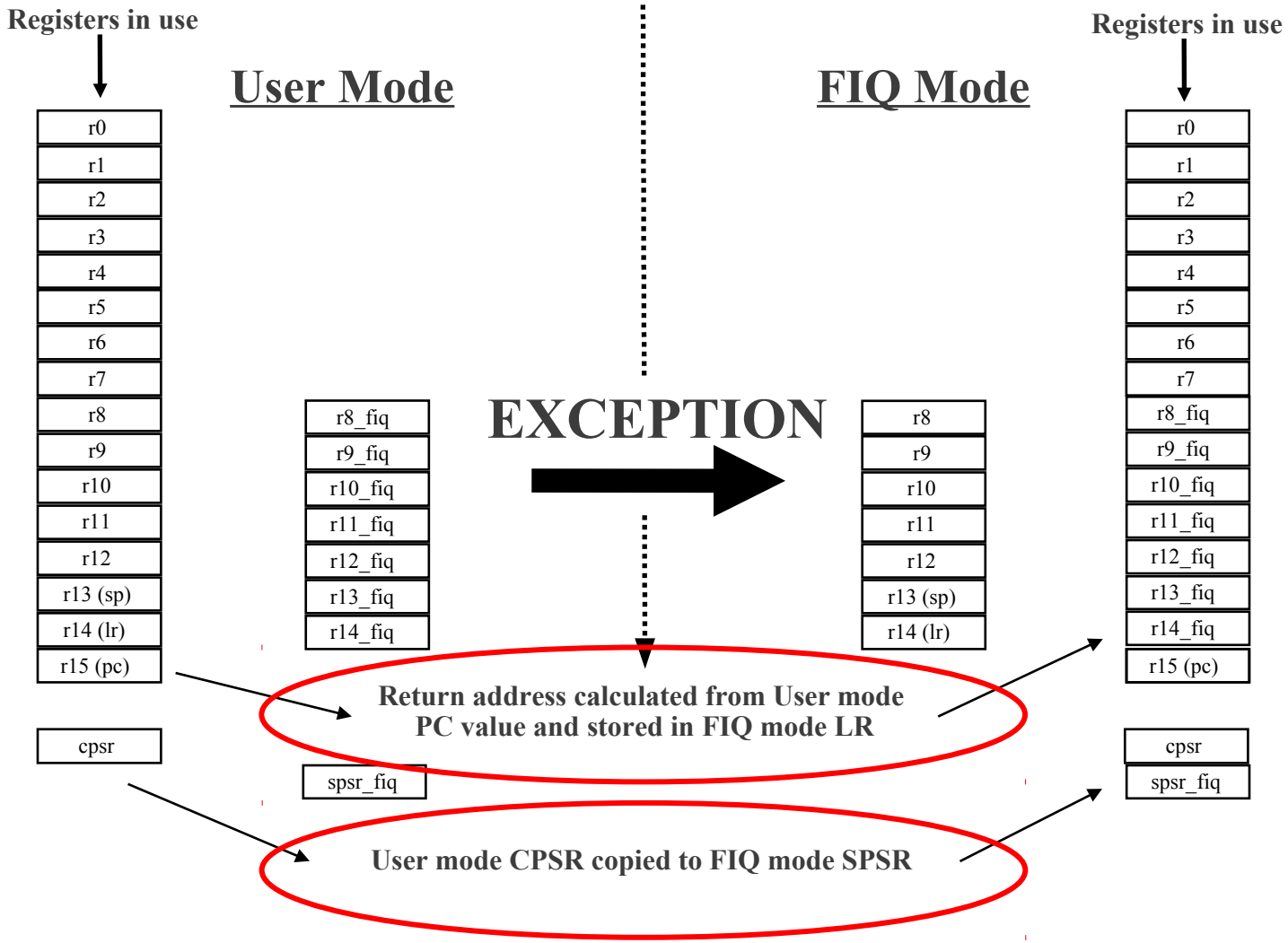
This can only be done in ARM state.



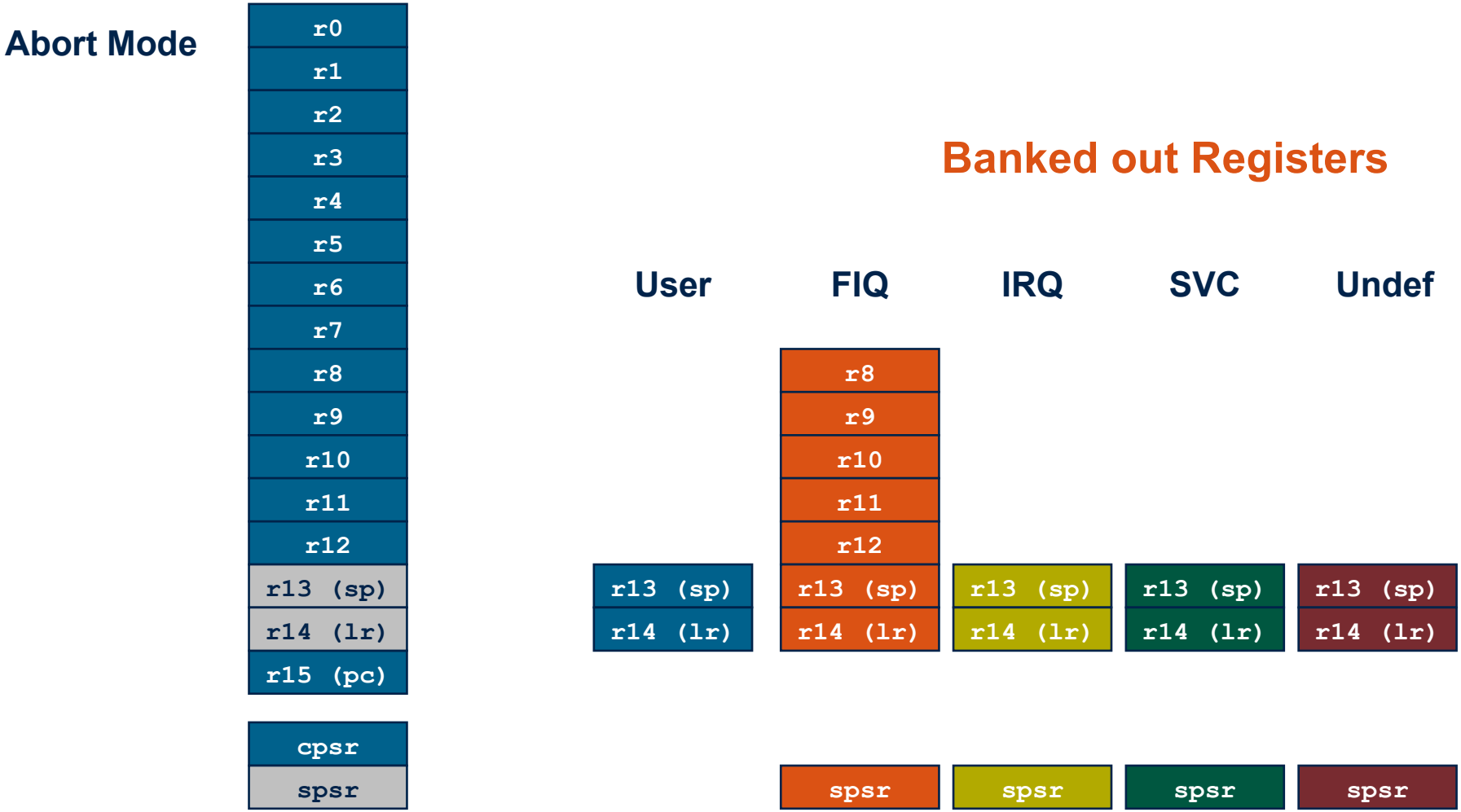
Vector Table

Vector table can be at **0xFFFF0000** on ARM720T and on ARM9/10 family devices

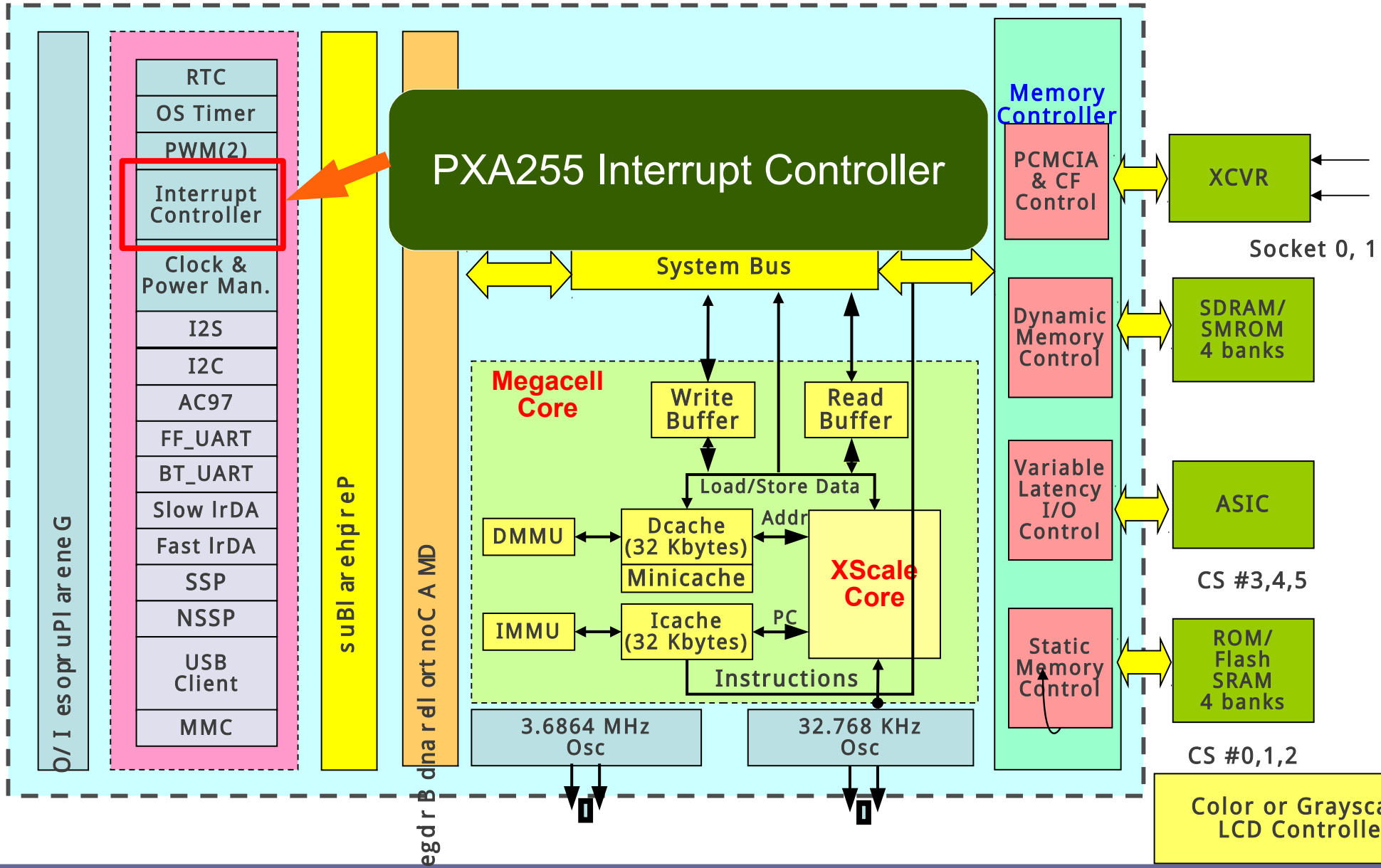
案例：從 user mode 切到 FIQ mode



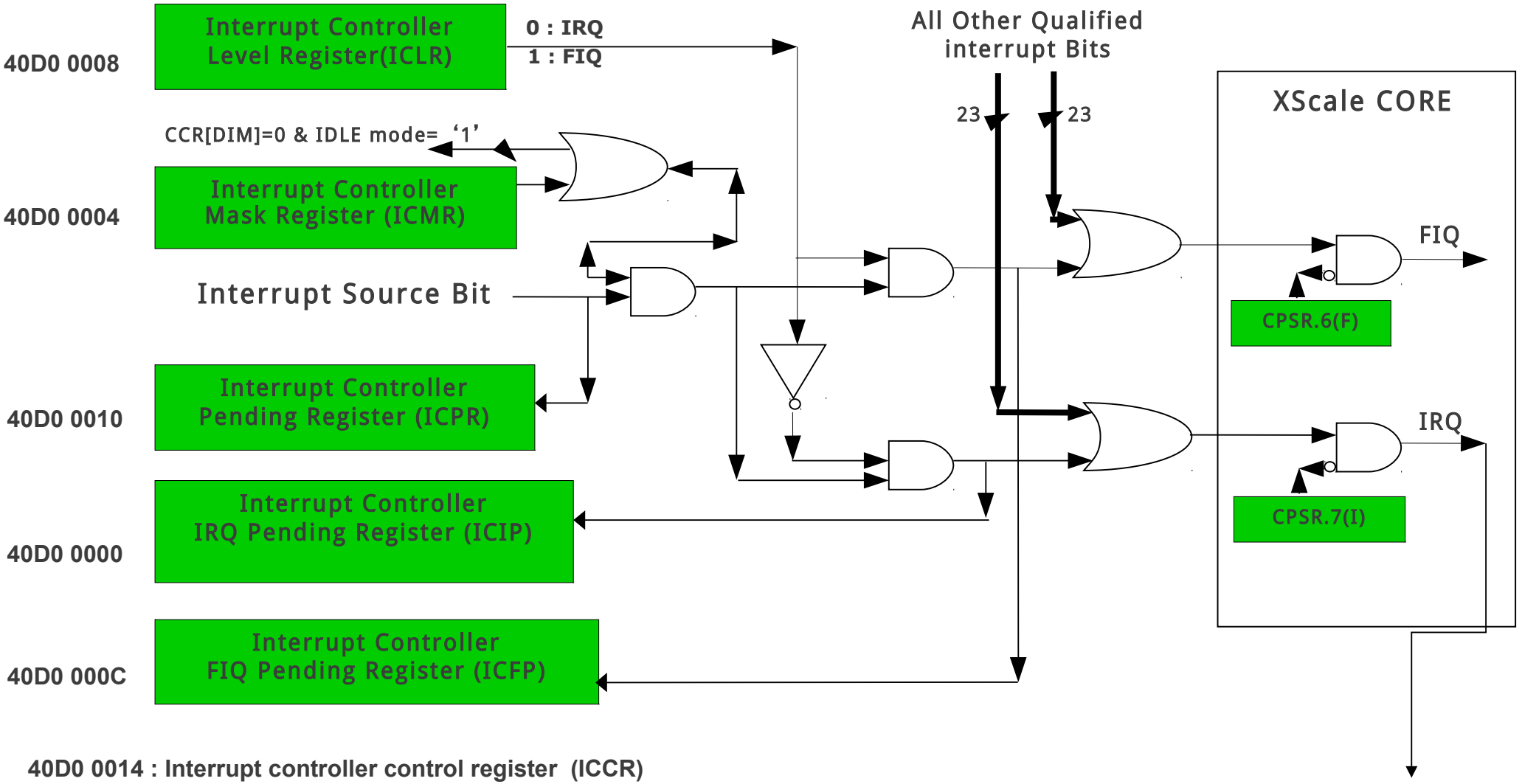
Current Visible Registers



PXA255 Function Block



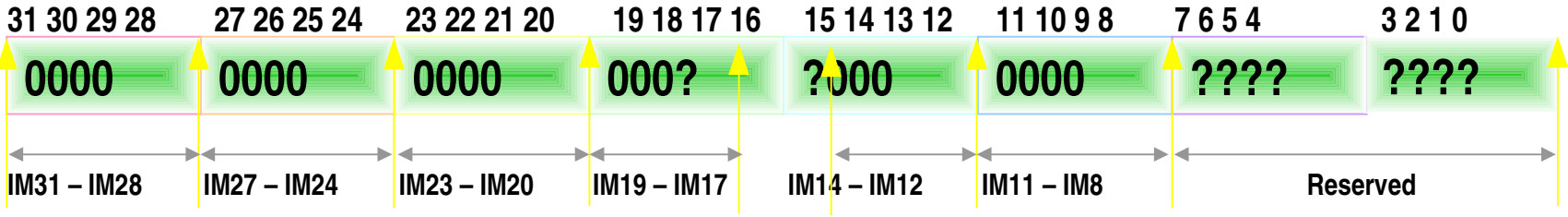
PXA255 Interrupt controller



40D0 0014 : Interrupt controller control register (ICCR)

ICCR.0 : disable idle mask(DIM)

ICMR (Interrupt Controller Mask Register)



IM[x] Interrupt Mask 'x' (where x= 8 through 14 and 17 through 31).

0 – Pending interrupt is masked from becoming active (interrupts are NOT sent to CPU or Power Manager).

1 – Pending interrupt is allowed to become active (interrupts are sent to CPU and Power Manager).

NOTE: **In idle mode, the IM bits are ignored if ICCR[DIM] is cleared.**

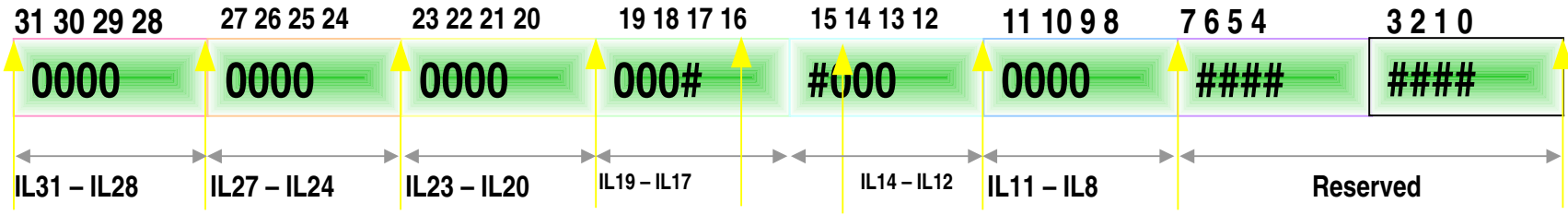
Reserved[0-7, 15, 16]

Physical Address : 0x40D0/0004

CuRT_v1/arch/arm/mach-pxa/start.S

```
        /*
         * Initializing PXA250 interrupt controller.
         */
mask_off_int_reg: /* Mask off all IRQs and FIQs */
        ldr r1, =(INT_BASE | INT_ICMR)
        ldr r2, =0x0 /* interrupt gets mask off */
        str r2, [r1]
```

ICLR (Interrupt Controller Level Register)



IL[x] Interrupt Level 'x' (where n = 8 through 14 and 17 through 31).

0 – Interrupt routed to IRQ interrupt input.

1 – Interrupt routed to FIQ interrupt input.

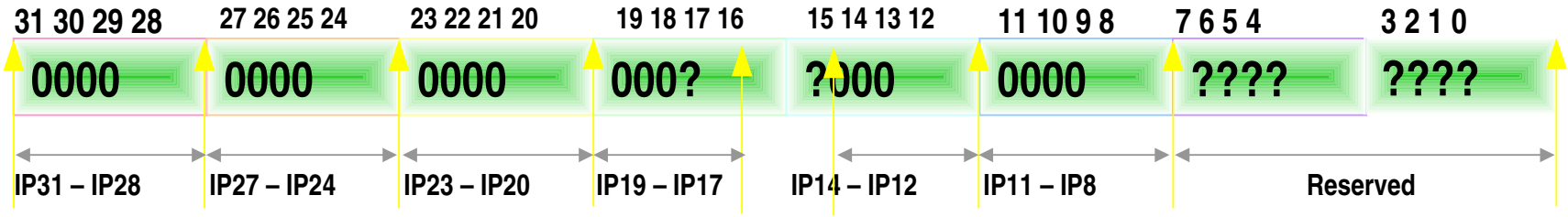
Reserved[0-7, 15, 16]

Physical Address : 0x40D0/0008

CuRT_v1/arch/arm/mach-pxa/port.c

```
void init_os_timer()
{
    INT_REG(INT_ICLR) &= ~BIT26;
    TMR_REG(TMR_OSMR0) = PXA255_TMR_CLK / OS_TICKS_PER_SEC;
    TMR_REG(TMR_OSMR1) = 0x3FFFFFFF;
    TMR_REG(TMR_OSMR2) = 0x7FFFFFFF;
    TMR_REG(TMR_OSMR3) = 0xBFFFFFFF;
    TMR_REG(TMR_OSCR) = 0x00;
    TMR_REG(TMR_OSSR) = BIT0;
    TMR_REG(TMR_OIER) = BIT0;
    INT_REG(INT_ICMR) |= BIT26;
}
```

ICIP (Interrupt Controller IRQ Pending Register)



IP[x] : IRQ Pending x (where x = 8 through 14 and 17 through 31).

0 – IRQ NOT requested by any enabled source.

1 – IRQ requested by an enabled source.

Reserved[0-7, 15, 16]

Physical Address : 0x40D0/0000

CuRT_v1/arch/arm/mach-pxa/port.c

```
void interrupt_handler()
{
    if (INT_REG(INT_ICIP) & BIT26) {
        TMR_REG(TMR_OSCR) = 0x00;
        advance_time_tick();
        TMR_REG(TMR_OSSR) = BIT0;
    }
}
```

- PXA255 SoC 與 CuRT 的硬體啓動程序
- ARM Interrupt, ISR, Exception 的處理
- ARM 定址與組合語言概況

- 指令語法
  - <opcode>{<cond>}{S} <Rd>, <Rn>, <shifter-operand>

```
ADD     r0, r1, r2     ; r0 = r1 + r2, don't update flags
ADDS    r0, r1, r2     ; r0 = r1 + r2, and update flags
ADDCSS  r0, r1, r2     ; If C flag set then r0 = r1 + r2, and update flags
CMP     r0, r1         ; update flags based on r0-r1.
```

ARM 組合語言強大的語法

類 C 程式碼

```
if (z==1) R1=R2+(R3*4)
```

可編譯爲以下的 ARM 組合語言指令

```
EQADDS R1, R2, R3, LSL #2
```

→ 只要一道指令

ARM 的指令集概述

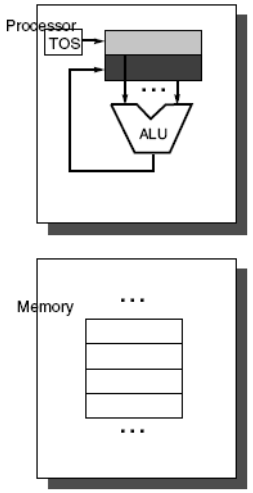
- ARM 指令集採用 Load / Store 架構
  - 也即指令集僅能處理暫存器中的資料，且處理結果都要再放回暫存器中

堆疊定址

```
Push   A
Push   B
Add
```

» Pop the top-2 values of the stack (A, B) and push the result value into the stack

```
Pop    C
```

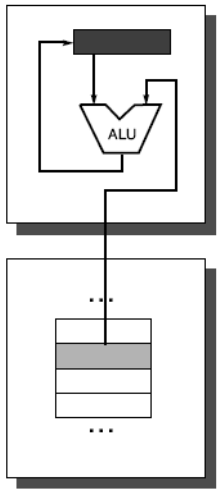


累加器定址

```
Load   A
Add    B
```

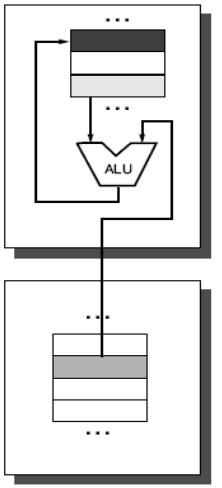
» Add AC (A) with B and store the result into AC

```
Store  C
```



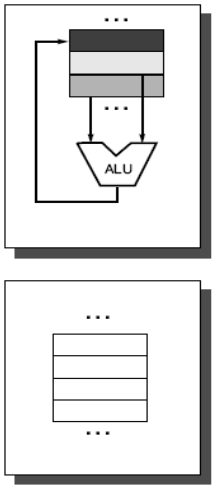
暫存器－記憶體定址 (register-memory)

```
Load    R1, A
Add     R3, R1, B
Store   R3, C
```



暫存器定址 (load-store)

```
Load    R1, A
Load    R2, B
Add     R3, R1, R2
Store   R3, C
```



ARM 指令集概述

- ARM 指令集分爲
  - 跳躍指令
  - 資料處理指令
  - 程式狀態暫存器 (PSR) 處理指令
  - 載入 / 存回指令
  - 協同處理器指令
  - 例外事件產生指令

僅探討 CuRT 所用的指令

ARM 指令的定址模式

- 定址方式指 CPU 根據指令中所給予的「位址」訊息來尋找出「實體位址」的方式
- ARM 指令的定址模式
  - 立即定址
  - 暫存器定址
  - 暫存器間接定址
  - 基底定址
  - 相對定址
  - 多暫存器定址
  - 堆疊定址

定址模式 :: 立即定址

- 立即定址： 運算元本身就在指令中直接加以設定，只要取出指令也就取到的運算元

  ADD R3, R3, #1 ; R3 ← R3+1

  - 將 R3 暫存器的內容加 1，結果放回 R3 中

  ADD R8, R7, #&ff ; R8 ← R7[7:0]

  - 將 32-bit 的 R7 取其低 8-bit 的數值，即作 AND 邏輯運算，然後將結果傳至 R8 中

- 第二個來源運算元即爲立即數值，並要求以「#」爲首碼
- 對於十六進制表示的立即數值，在 「#」後加上「0x」 或 「&」

定址模式 :: 暫存器定址

- 暫存器定址：利用「暫存器中的內含值」作爲運算元，這種定址方式是各類處理器常用的方式，也是執行效率較高的定址方式

```
ADD    R0, R1, R2        ; R0 ← R1 + R2
```

  - 定址碼即爲暫存器編號，暫存器內容爲運算元
  - 第一個是目的（結果）暫存器，第二個是來源（運算元）暫存器，第三個是來源（運算元）暫存器
  - 該指令將暫存器 R1 和 R2 的內容相加，其結果存放在暫存器 R0 中

定址模式 :: 暫存器間接定址

- 暫存器間接定址：以暫存器中的數值作爲運算元的位址，而運算元本身是存放在記憶體中

ADD R0, R1, [R2] ; R0 ← R1+[R2]

- 以暫存器 R2 的數值作爲運算元的位址，在記憶體中取得一個運算元後與 R1 相加，結果存入暫存器 R0 中

LDR R0, [R1] ; R0 ← [R1]

- 將 R1 所指向的記憶體單元的內容載入至 R0

STR R0, [R1] ; [R1] ← R0

- 將 R0 取回至 R1 所指向的記憶體單元中

- 指令中的定址碼設定一組通用的暫存器編號，被指定的暫存器中存放運算元的有效位址
  - 運算元則存放在記憶體單元中，暫存器即爲位址指標
  - 暫存器間接定址使用一個暫存器（基底暫存器）的數值作爲記憶體的位址

定址模式 :: 基底定址 (1/3)

- 基底定址：將暫存器（該暫存器稱之爲基底暫存器）的內容與指令中所給予的位址偏移量加以相加，並進而得到一個運算元的有效位址

LDR R0, [R1, #4] ; R0 ← [R1+4]（前索引定址）

- 基底定址是用來處理基底附近的記憶體，包含二種定址：

  基底加偏移量：前索引 (Pre-Index) 與後索引 (Post-Index) 定址

  （基底 + 索引定址）

- 暫存器間接定址則是偏移量爲 0 的基底加上偏移定址的方式
- 但基底加偏移定址中的基底暫存器包含的並非是確定的位址，基底需加（減）最大 4KB 的偏移來計算出所要處理的位址，也就是前索引與後索引定址的計算

定址模式 :: 基底定址 (2/3)

- 除找到基底定址所指向的記憶體資料外，還可改變這基底暫存器

```
LDR    R0, [R1, #4] !    ; R0 ← [R1+4], R1 ← R1+4
```

  - 改變基底暫存器來指向下一個所傳送的位址，這對於多筆資料傳送很有用
  - 其中「!」表示指令在完成資料傳送後，同時更新基底暫存器
  - ARM 對這種自動索引的方式，不消耗額外的週期

定址模式 :: 基底定址 (3/3)

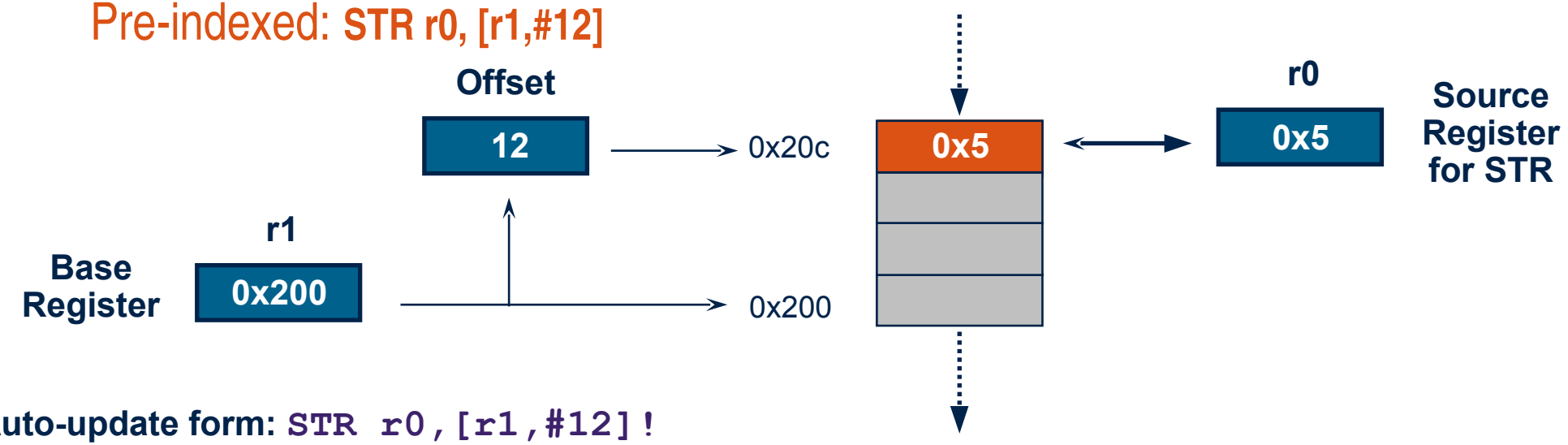
- 後索引定址：基底不包含偏移量來做爲傳送的位址，且再傳送後，自動加上索引的方式

```
LDR     R0, [R1], #4      ; R0 ← [R1], R1 ← R1+4
```

  - 沒有使用「!」，只用了立即數值的偏移量來作爲基底暫存器的變化量
- 此外，基底加上索引定址的方式是在指令中指定一個暫存器爲基底，然後再指定另一個暫存器當索引

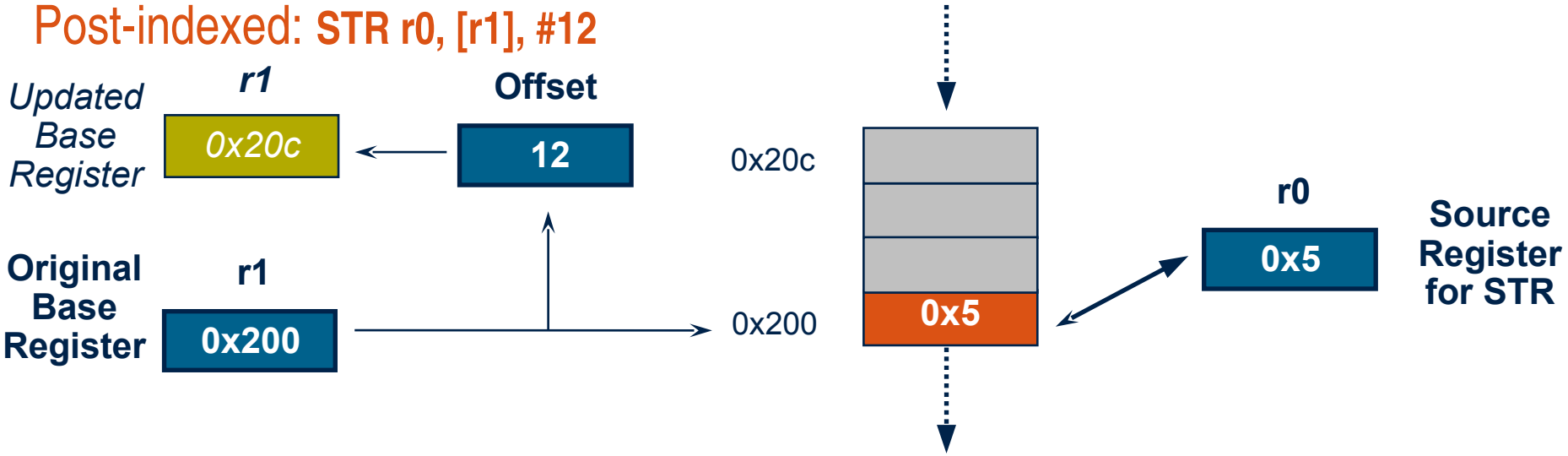
```
LDR     R0, [R1, R2]      ; R0 ← [R1+R2]
```

Pre-indexed: STR r0, [r1,#12]



Auto-update form: `STR r0,[r1,#12]!`

Post-indexed: STR r0, [r1], #12



定址模式 :: 相對定址

- 相對定址：以程式計數器 PC 的目前數值作爲基底位址，指令中的位址標號作爲偏移量，再將兩者相加之後得到有效位址

```
BL      NEXT            ; 跳躍到副程式 NEXT 處執行
.....
NEXT
.....
MOV     PC, LR          ; 從副程式返回
```

- 上述程式完成了副程式的跳躍與返回
- 其中，跳躍指令 BL 採用了相對定址的方式

定址模式 :: 多暫存器定址

- 多暫存器定址：一道指令可以完成多個暫存器值的傳送

```
LDMIA    R0, {R1, R2, R3, R4}        ; R1 ← [R0]
                                     ; R2 ← [R0+4]
                                     ; R3 ← [R0+8]
                                     ; R4 ← [R0+12]
```

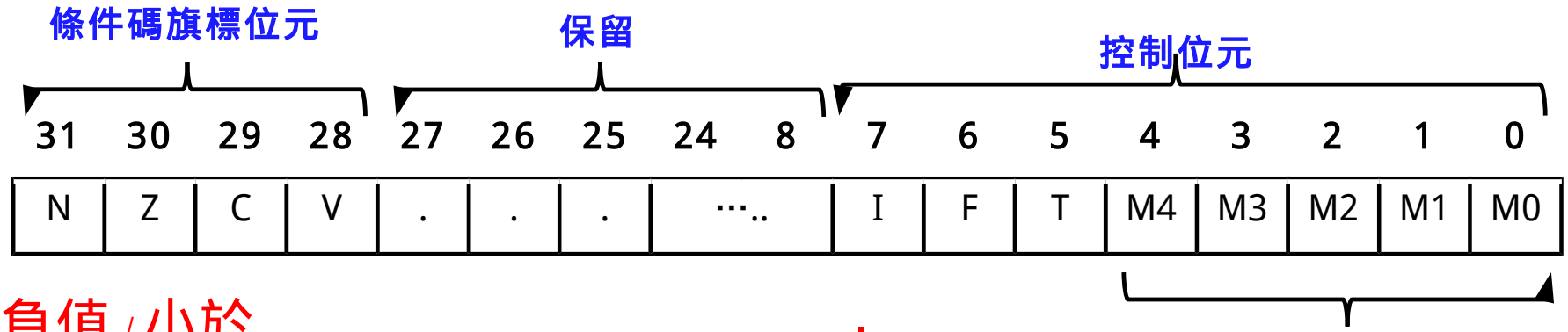
  - 最多傳送 16 個暫存器
  - 該指令的字尾「IA」表示在每次執行完 Load / Store 操作後，R0 按字元組長度增加，因此，這道指令可將連續記憶體單元的數值傳送到 R1 ~ R4

定址模式 :: 堆疊定址

- 堆疊是按照特定順序進行存取的記憶體區塊
- 透過後進先出（LIFO）或是先進後出（FILO）的順序
- 堆疊定址是隱含的，透過一個堆疊指標器來指向一塊堆疊區域之堆疊的頂端。記憶體堆疊分爲兩種：
  - 向上生長：向高位址方向生長，遞增堆疊
  - 向下生長：向低位址方向生長，遞減堆疊
- 堆疊指標指向最後進入的有效數據，稱爲滿堆疊
- 堆疊指標指向下一個數據放入的空位置，稱爲空堆疊
- 可分爲滿遞增（FA）、空遞增（EA）、滿遞減（FD）、空遞減（ED）
- 透過 push 寫入資料與 pop 讀取資料

程式狀態暫存器

- ARM 包含一個目前程式狀態暫存器（CPSR）和 5 個儲存程式狀態暫存器（SPSR）。 SPSR 用來進行例外事件處理，其功能包括：
  - 保存 ALU 中目前操作資訊
  - 控制允許和禁止中斷
  - 設定處理器的執行模式
- 程式狀態暫存器（CPSR）的每位元意義如下：



N: 負值 / 小於
Z: 0
C: 進位 / 借位 / 擴展
V: 溢位

I: 除能
F: 除能
T: 狀態位元

條件碼旗標欄位 (1/2)

- 條件碼旗標欄位 (Condition Code Flags)
  - N、Z、C 與 V 均爲條件碼旗標位元。它們的內容可被算術或邏輯運算的結果而有所改變，並且可以決定某條指令是否被執行
    - 在 ARM 狀態下，絕大多數的指令都是有條件執行
    - 在 Thumb 狀態下，僅有跳躍指令都是有條件執行

條件碼旗標欄位 (2/2)

旗標位元	意義
N	負值旗標，當用 2 的補數所表示的有號數進行運算時， N=1 ，表示運算的結果爲負數；N=0 ，表示運算的結果爲正數或零
Z	零值旗標， Z=1 表示運算的結果爲零；Z=0 表示運算的結果爲非零
C	進位旗標，有 4 種方法來設定 C 的值： •加法運算（包括比較指令 CMN)： 當運算結果產生了進位時（無號數溢出），C=1 ，否則 C=0 •減法運算（包括比較指令 CMP)： 當運算時產生了借位（無號數溢出），C=0 ，否則 C=1。 •對於包含移位操作的非加 / 減運算指令， C 爲移出值的最後一位元 •對於其他的非加 / 減運算指令， C 值通常不改變
V	溢位旗標，有 2 種方法來設置 V 的值： •對於加 / 減運算指令，當運算元和運算結果爲 2 補數表示的有號數時， V=1 表示有號數溢位，反之則 V=0 •對於其他的非加 / 減運算指令， V 值通常不改變
Q	在 ARM v5 及以上版本的 E 系列處理器中，用 Q 旗標位元來指示增強的 DSP 運算指令是否發生了溢位。在其他版本的處理器中， Q 旗標位元無定義

控制欄位 (1/2)

- PSR 的低 8 位元（包括 I、F、T 和 M[4:0]）稱之爲控制位元。當發生例外事件時，這些位元可被改變。若處理器要執行特權模式時，這些位元也可由程式來修改
  - 中斷禁止位元 I 與 F
    - I =1，禁止 IRQ 中斷
    - F = 1，禁止 FIQ 中斷
  - T 旗標位元： 該位元反映處理器的執行狀態
    - T = 0， ARM 狀態
    - T = 1， Thumb 狀態
  - 執行模式位元 M[4:0]: M0、M1、M2、M3 與 M4 是模式位元。這些位元決定了處理器的執行模式

控制欄位 (2/2)

- 執行模式位元 M[4:0] 的具體定義表

M[4:0]	處理器模式	可存取的暫存器
0b10000	使用者模式	PC, CPSR, R0~R14
0b10001	FIQ 模式	PC, CPSR, SPSR_FIQ, R14_FIQ~R8_FIQ, R7~R0
0b10010	IRQ 模式	PC, CPSR, SPSR_IRQ, R14_IRQ, R13_IRQ, R12~R0
0b10011	管理者模式	PC, CPSR, SPSR_SVC, R14_SVC, R13_SVC, R12~R0
0b10111	終止模式	PC, CPSR, SPSR_ABT, R14_ABT, R13_ABT, R12~R0
0b11011	未定義模式	PC, CPSR, SPSR_UND, R14_UND, R13_UND, R12~R0
0b11111	系統模式	PC, CPSR (ARM v4+), R14~R0

保留位元

- PSR 中的其餘位元爲保留位元，當改變 PSR 中的條件碼旗標位元或控制位元時，保留位元不能被改變，在程式中也不要用保留位元來儲存資料
- 這些保留位元將應用於未來 ARM 版本

例外事件 (1/2)

- 例外事件：當正常的程式執行流程發生暫時的停止時
  - ARM 的例外事件可分 3 類：
    - 指令執行引起的直接例外：軟體中斷，未定義指令（包括所要求的協同處理器不存在時的協同處理器指令）和預取指令中止（因預取指令過程中的記憶體故障導致的無效指令）
    - 指令執行引起的間接例外：資料中止（在 Load 和 Store 資料存取時的記憶體故障）
    - 外部產生的與指令流無關的例外：重置、IRQ 和 FIQ
  - 例如：處理一個外部的中斷請求。在處理例外事件之前，目前處理器的狀態必須被加以保留，這樣當例外事件完成後，目前的程式才可以繼續執行
  - 允許多個例外事件同時發生，但有固定的優先順序
  - ARM 的例外事件與 8 -bit/16-bit 微處理器系列結構的中斷相似，但概念不完全相同

例外事件 (2/2)

例外事件類型	具體含義
重置 (1)	當處理器的重置電位（nRESET）有效時，產生重置例外事件，程式會跳到重置例外事件處理程式處開始執行
未定義指令 (7)	當 ARM 或輔助運算器遇到不能處理的指令時，產生未定義指令例外事件。但我們可用這種例外事件的機制來進行軟體的模擬與除錯的目的
軟體中斷 (6)	該例外事件由執行 SWI 指令產生，可用於使用者模式下的程式來引用特權操作指令。但我們可用這種例外事件的機制來實現系統功能的引用
指令預取終止 (5)	若處理器預取指令的位址不存在，或該位址不允許目前指令來存取，記憶體會向處理器發出終止信號，但當預取的指令被執行時，才會產生指令預取終止的例外事件
資料終止 (2)	若處理器資料存取指令位址不存在，或該位址不允許目前指令來存取，產生資料終止的例外事件
IRQ (4)	當處理器的外部中斷請求接腳（nIRQ）有效，且 CPSR 中的 I 位元爲 0 時，會產生 IRQ 例外事件。系統的外部設備可通過該例外事件來請求中斷服務
FIQ (3)	當處理器的快速中斷請求接腳（nFIQ）有效，且 CPSR 中的 F 位元爲 0 時，就會產生 FIQ 例外事件

ARM 指令 :: 跳躍指令

- 跳躍指令用於實現程式流程的跳躍，有二種方法可實現：
  - 使用專門的跳躍指令
  - 直接向程式計數器 PC 寫入跳躍位址值，可在 4GB 的位址空間中任意跳躍
- ARM 指令集中的跳躍指令可從目前指令向前、或向後的 32MB 的位址空間的跳躍。包括以下 4 個指令：
  - **B**: Branch（分歧），跳躍指令
  - **BL**: Branch with Link（分歧連結），包含返回的跳躍指令，即呼叫副程式
  - **BLX**: Branch and Exchange Instruction Set（分歧交換），包括返回和狀態切換的跳躍指令
  - **BX**: Branch with Link and Exchange Instruction Set（分歧連結交換），包含狀態切換的跳躍指令

跳躍指令 ::B

- B 指令格式：

  B{條件} 目的位址

  - 爲單純的跳躍指令，程式控制權從目前的位址轉移到另一個位址
  - 若超過 32MB 範圍時，可用 BX 、 LDR 指令直接改變 PC 值

```
B       LABEL1          ; 程式無條件跳躍至 LABEL1 處執行
CMP     R1, #0
BEQ     LABEL1          ; 若 Z 旗標欄位被設定，則跳躍至 LABEL1
BCS     LABEL2          ; 若 C 進位旗標被設定，則跳躍至 LABEL2
```

跳躍指令 ::BL

- BL 指令格式：

BL{條件} 目的位址

- 除轉移程式的控制權外，同時也將程式執行順序的下一個位址紀錄到鏈結暫存器（LR）中，如此可作爲呼叫副程式呼叫時使用
- 副程式距離 PC 的範圍與指令 B 一樣：+/–32MB
- 當副程式執行完畢後，可將鏈結暫存器（LR）的值複製回 PC 中，即可達到返回原程式的目的

```
BL   SUB1    ; 當程式無條件呼叫副程式 SUB1，同時將目前的 PC 值保存到 R14
...
SUB1                ; 副程式進入點
MOV      PC, LR     ; 返回原程式
```

跳躍指令 ::BX

- BX 指令格式：

BX{條件} 目的位址

  - 將暫存器 <Rm> 的數值複製至 PC 中，以達到轉移程式控制權
  - 根據暫存器 <Rm> 的最低位元 <Rm>[0] 來變更指令集狀態，<Rm>[0] 爲 1，則變更爲 THUMB 指令集狀態。<Rm>[0] 爲 0，則變更爲 ARM 指令集狀態； 其餘 <Rm>[31:1] 移入 PC
  - 此指令可將程式控制權轉移到 4GB 絕對位址的任一位址

```
CODE   32  ; 從此處起的程式以 ARM 指令集編譯
...
BX   R0      ; 若 R0[31:1] 爲位址 LABEL1 ， R0[0] 爲 1 ,
             ; 當跳躍至 LABEL1 處執行時，切換爲 THUMB 指令集狀態
....
CODE   16  ; 指示從此處的程式爲 THUMB 指令集狀態
LABEL   1:  ; LABEL1 程式進入點
```

跳躍指令 :: BLX (1/2)

- BLX 指令格式：

BLX 目的位址

- 從 ARM 指令集跳躍到指令中所指定的目的位址，並將處理器的工作狀態從 ARM 狀態切換到 THUMB 狀態，該指令同時將 PC 的目前內容保存到暫存器 R14 中
- 此指令有兩種使用方式。第一種爲與 BX 搭配，可將程式控制權轉移到 4GB 絕對位址的任一字元位址

```
CODE    32      ; 從此處起的程式以 ARM 指令集編譯
…
BLX   R0    ; 呼叫副程式 SUB1 ， R0[0] 爲 1 ，切換爲 THUMB 指令集狀態
…
CODE   16  ; 指示從此處的程式爲 THUMB 指令集狀態
SUB1 :      ; SUB1 程式進入點
BX      R14
```

跳躍指令 ::BLX (2/2)

- 第二種是轉移程式控制權，同時也將原本下一個執行位址紀錄到 LR 暫存器中，並變更 THUMB 指令集狀態。搭配 BL 與 B 指令
- 副程式距離 PC 的範圍與指令 B 一致：+/ – 32MB

```
CODE   32      ; 從此處起的程式以 ARM 指令集編譯
.
BLX    R0      ; 呼叫副程式 SUB1 ，且改爲 THUMB 指令集狀態
.
CODE   16      ; 指示從此處的程式爲 THUMB 指令 集狀態
SUB1 :         ; SUB1 程式進入點
```

資料處理指令

- 資料處理指令可分爲資料傳送指令、算數邏輯運算指令和比較指令
  - 資料傳送指令：用於暫存器和記憶體之間進行資料的雙向傳輸
  - 算數邏輯運算指令：完成常用的算術與邏輯的運算，該類指令不但是將運算結果保存在目的暫存器中，同時更新 CPSR 中相應的條件旗標位元
  - 比較指令：不保存運算結果，只更新 CPSR 中相應的條件旗標位元
- 資料處理指令包括算術指令（加，減）、邏輯指令（AND、OR、NOT、XOR、位元清除）、比較測試指令，以及複製指令

MOV 指令

- MOV 指令格式

MOV{條件}{S} 目的暫存器， 來源暫存器

- 可從另一個暫存器、被移位的暫存器或將一個立即數載入到目的暫存器。其中，S 選項決定指令的操作是否影響 CPSR 中條件旗標位元的值，當沒有 S 時，指令不更新 CPSR 中條件旗標位元的值
- 程式範例

```
MOV   R1, R0          ; 將暫存器 R0 的值傳送到暫存器 R1
MOV   PC, R14         ; 將暫存器 R14 的值傳送到 PC
                      ; 常用於副程式返回
MOV   R1, R0, LSL #3  ; 將暫存器 R0 的值左移 3 位元後傳送到 R1
```

MVN 指令

- MVN 指令格式：

MVN{條件}{S} 目的暫存器， 來源暫存器

- 可從另一個暫存器被移位的暫存器，或將一個立即數載入到目的暫存器。與 MOV 指令不同之處是在傳送之前，被加以反相了。其中，S 選項決定指令的操作是否影響 CPSR 中條件旗標位元的值，當沒有 S 時，指令不更新 CPSR 中條件旗標位元的值
- 程式範例

```
MVN   R0, #0          ; 將立即數 0 取反相值傳送到暫存器 R0
                      ; 完成後， R0= – 1
```

CMP 指令

- CMP 指令格式：

CMP{條件} 運算元 1, 運算元 2

  - 用於把一個暫存器的內容和另一個暫存器的內容，或立即數進行比較。同時更新 CPSR 中條件旗標位元的值
  - 該指令進行一次減法運算，但不儲存結果，只更改條件旗標位元
  - 程式範例

```
CMP   R1, R0    ; 將暫存器 R1 的值與暫存器 R0 的值相減，
                ; 並根據結果設定 CPSR 的旗標位元
CMP   R1, #100  ; 將暫存器 R1 的值與立即數 100 相減，
                ; 並根據結果設定 CPSR 的旗標位元
```

TST 指令

- TST 指令格式：

TST{條件} 運算元 1, 運算元 2

- 用於把一個暫存器的內容和另一個暫存器的內容，或立即數進行每一位元的 AND 運算，並根據結果更改條件旗標位元
- 運算元 1 是要測試的資料，而運算元 2 是一個位元遮罩，該指令一般用來檢測是否設定了特殊位元
- 程式範例

```
TST   R1, #%1    ; 用於測試在暫存器 R1 中是否設定了最低位元
                 ;(% 表示二進制數值 )
TST   R1, #oxffe ; 將暫存器 R1 的值與立即數 oxffe 按位元作 AND
                 ; 運算，並根據結果來設定 CPSR 的旗標位元
```

ADD 指令

- ADD 指令格式：

ADD{條件}{S} 目的暫存器，運算元 1, 運算元 2

  - 用於把兩個運算元相加，並將結果值存放到目的暫存器中
  - 運算元 1 應是個暫存器，運算元 2 可以是個暫存器、被移位的暫存器，或是個立即值
  - 程式範例

```
ADD   R0, R1, R2          ; R0 = R1 + R2
ADD   R0, R1, #256        ; R0 = R1 + 256
ADD   R0, R2, R3, LSL#1   ; R0 = R2 + (R3 << 1)
```

SUB 指令

- SUB 指令格式：

SUB {條件} {S} 目的暫存器， 運算元 1，運算元 2

  - 用於把運算元 1 減去運算元 2，並將結果值存放到目的暫存器中
  - 運算元 1 應是個暫存器，運算元 2 可以是個暫存器、被移位的暫存器，或是個立即值
  - 程式範例

```
SUB   R0, R1, R2          ; R0 = R1 – R2
SUB   R0, R1, #256        ; R0 = R1 – 256
SUB   R0, R2, R3, LSL#1   ; R0 = R2 – (R3 << 1)
```

AND 指令

- AND 指令格式：

AND{條件}{S} 目的暫存器，運算元 1, 運算元 2

  - 用於在兩個運算元上進行邏輯 AND 運算，並將結果值存放到目的暫存器中。該指令常用於遮罩運算元 1 的某些位元
  - 運算元 1 應是個暫存器，運算元 2 可以是個暫存器、被移位的暫存器，或是個立即值
  - 程式範例：

```
AND   R0, R0, #3        ; 保持 R0 的第 0 與 1 位 (3=%11)，
                        ; 其餘位元清除
```

ORR 指令

- ORR 指令格式：

ORR{條件}{S} 目的暫存器，運算元1, 運算元2

- 用於在兩個運算元上進行邏輯 OR 運算，並將結果值存放到目的暫存器中。該指令常用於設置運算元 1 的某些位元
- 運算元 1 應是個暫存器，運算元 2 可以是個暫存器、被移位的暫存器，或是個立即值
- 程式範例：

```
ORR   R0, R0, #3        ; 設置 R0 的第 0 與 1 位
                        ; 其餘位元保持不變
```

BIC 指令

- BIC 指令格式：

BIC{條件}{S} 目的暫存器，運算元 1, 運算元 2

- 用於清除運算元 1 的某些位元，並將結果值存放到目的暫存器中。該指令常用於反轉運算元 1 的某些位元
- 運算元 1 應是個暫存器，運算元 2 可以是個暫存器、被移位的暫存器，或是個立即值
- 程式範例：

```
BIC   R0, R0, #%1011    ; 清除 R0 的第 0 、 1 與 3 位元
                        ; 其餘位元保持不變
```

程式狀態暫存器 (PSR) 存取指令

- 用於在「程式狀態暫存器」 (PSR) 和通用暫存器間傳送資料
- 包括兩種指令
  - MRS：程式狀態暫存器 → 通用暫存器的資料傳送
  - MSR：通用暫存器 → 程式狀態暫存器的資料傳送

MRS 指令

- MRS 指令格式

MRS {條件} 通用暫存器，程式狀態暫存器 (CPSR 或 SPSR)

- 用於 CPSR / SPSR 的內容傳送到通用暫存器中，一般用於以下幾種情況：
  - 當需要改變 CPSR / SPSR 的內容時，可用 MRS 將程式狀態暫存器的內容讀入通用暫存器，修改後再寫回程式狀態暫存器
  - 當在例外事件處理或程式切換時，需要保持 CPSR / SPSR 的值，可先用該指令讀出程式狀態暫存器的值，然後加以保存
- 程式範例：

```
MRS     R0, CPSR      ; 傳送 CPSR 的內容到 R0
MRS     R0, SPSR      ; 傳送 SPSR 的內容到 R0
```

MSR 指令 (1/3)

- MSR 指令的格式

MSR{條件} 程式狀態暫存器 (CPSR 或 SPSR)_<區域>, 運算元

- 用於將運算元的內容傳送到 CPSR/SPSR 的特定域中。其中，運算元可以爲通用暫存器或立即數。<區域>用於設定程式狀態暫存器中需要操作的位元，32 位元的程式狀態暫存器可分 4 個區域：

  位元 [31:24] 爲條件旗標位元區域，用 f 表示

  位元 [23:16] 爲狀態位元區域，用 s 表示

  位元 [15: 8] 爲擴展位元區域，用 x 表示

  位元 [0: 7] 爲控制位元區域，用 c 表示

- 常用於恢復或改變 CPSR/SPSR 的內容，在使用時，一般要在 MSR 指令中指明要操作的區域

- 程式範例：

```
MSR     CPSR, R0      ; 傳送 R0 的內容到 CPSR
MSR     SPSR, R0      ; 傳送 R0 的內容到 SPSR
MSR     CPSR_c, R0    ; 傳送 R0 的內容到 CPSR
                      ; 但僅修改 CPSR 中的控制位元區域
```

MSR 指令 (2/3)

- 在使用者模式的指令動作範例：

```
MSR     CPSR_all, Rm   ; 傳送 Rm[31:28] 的內容到 CPSR[31:0]
MSR     CPSR_flg, Rm   ; 傳送 Rm[31:28] 的內容到 CPSR[31:28]
MSR     CPSR_flg, #0xA0000000 ; 設定 CPSR[31:28] 爲 0xA
                              ; (設定 N 、 C； 清除 Z 、 V)
MRS     Rd, CPSR              ; 傳送 Rm[31:0] 的內容到 CPSR[31:0]
```

MSR 指令 (3/3)

- 在特權模式的指令動作範例：

```
MSR     CPSR_all, Rm            ; 傳送 Rm[31:0] 的內容到 CPSR[31:0]
MSR     CPSR_flg, Rm            ; 傳送 Rm[31:28] 的內容到 CPSR[31:28]
MSR     CPSR_flg, #0x50000000   ; 設定 CPSR[31:28] 爲 0x5
                                ;(設定 Z、V; 清除 N、C)
MSR     SPSR_all, Rm            ; 傳送 Rm[31:0] 的內容到 SPSR[31:0]
MSR     SPSR_flg, Rm            ; 傳送 Rm[31:28] 的內容到 SPSR[31:28]
MSR     SPSR_flg, #0xC0000000   ; 設定 SPSR[31:28] 爲 0xC
                                ;(設定 N、Z; 清除 C、V)
MRS     Rd, SPSR                ; 傳送 Rm[31:0] 的內容到 SPSR[31:0]
```

Load / Store 指令

- ARM 微處理器支援 Load / Store 指令，用於暫存器和記憶體之間傳送資料，載入指令用於將記憶體的資料傳送到暫存器，存回指令則完成相反的動作
- 常用的 Load / Store 指令：
  - LDR　字元組資料載入
  - STR　字元組資料存回
  - LDRB　位元組資料載入
  - STRB　位元組資料存回
  - LDRH　半字元組資料載入
  - STRH　半字元組資料存回

LDR 指令

- LDR 指令格式：

LDR {條件} 目的暫存器,<記憶體位址>

- 程式範例

```
LDR   R0, [R1]            ; 將記憶體位址爲 R1 的字元組資料讀到 R0
LDR   R0, [R1, R2]        ; 將記憶體位址爲 R1+R2 的字元組資料讀到 R0
LDR   R0, [R1, #8]        ; 將記憶體位址爲 R1+8 的字元組資料讀到 R0
LDR   R0, [R1, R2] !      ; 將記憶體位址爲 R1+R2 的字元組資料讀到 R0 ，
                          ; 並將新位址 R1+R2 寫入 R1
LDR   R0, [R1, #8] !      ; 將記憶體位址爲 R1+8 的字元組資料讀到 R0 ，
                          ; 並將新位址 R1+8 寫入 R1
LDR   R0, [R1], R2        ; 將記憶體位址爲 R1 的字元組資料讀到 R0 ，
                          ; 並將新位址 R1+R2 寫入 R1
LDR   R0, [R1, R2, LSL #2] !   ;將記憶體位址爲 R1+R2x4 的字元組資料
                               ; 讀到R0 ，並將新位址 R1+R2x4 寫入 R1
LDR   R0, [R1], R2, LSL #2     ;將記憶體位址爲 R1 的字元組資料讀到 R0 ，
                               ; 並將新位址 R1+R2x4 寫入 R1
```

STR 指令

- STR 指令格式：

STR {條件} 來源暫存器 ,< 記憶體位址 >

- 程式範例

```
STR   R0, [R1, #8]      ; 將 R0 中的字元組資料寫入到
                        ; 以 R1 爲位址的記憶體中，並將新位址
                        ; R1+8 寫入 R1
STR   R0, [R1, #8] !    ; 將 R0 中的字元組資料寫入到
                        ; 以 R1+8 爲位址的記憶體中，並將新位址
                        ; R1+8 寫入 R1
```

Part II 回顧

- ARM 中斷
- ARM 例外處理
- ARM 組合語言概況

參考資訊

- ARM Limited *ARM Architecture Reference Manual*, Addison Wesley, June 2000
- ARM Architecture Manual
- The ARM Instruction Set – ARM University Program
- Steve Furber *ARM System-On-Chip Architecture (2nd edition),* Addison Wesley, March 2000
- Intel Xscale Programmers Reference Manual