Cortex-M3 특징

2010.02.08

Cortex A-, R-, M-series

Cortex-Series	Description
Cortex-A	▪ A – Application ▪ 복잡한 OS 및 사용자 응용 프로그램에 사용하는 응용 프로그램 프로세서 계열 ▪ ARM, Thumb, Thumb-2 명령어 세트 지원 ▪ Cortex-A8 및 Cortex-A9 프로세서가 있다
Cortex-R	▪ R – Real-Time ▪ 실시간 시스템을 위한 임베디드 프로세서 계열 ▪ ARM, Thumb, Thumb-2 명령어 세트 지원 ▪ Cortex-R4, Cortex-R4F, Cortex-R4X 프로세서
Cortex-M	▪ **M – Microcontroller** ▪ 저가형 응용프로그램에 최적화된 임베디드 프로세서 계열에 적합하게 설계 ▪ **Thumb-2** 명령어 세트만 지원 ▪ **Cortex-M3** 및 Cortex-M0, M1 FPGA 프로세서

ARM Licenses

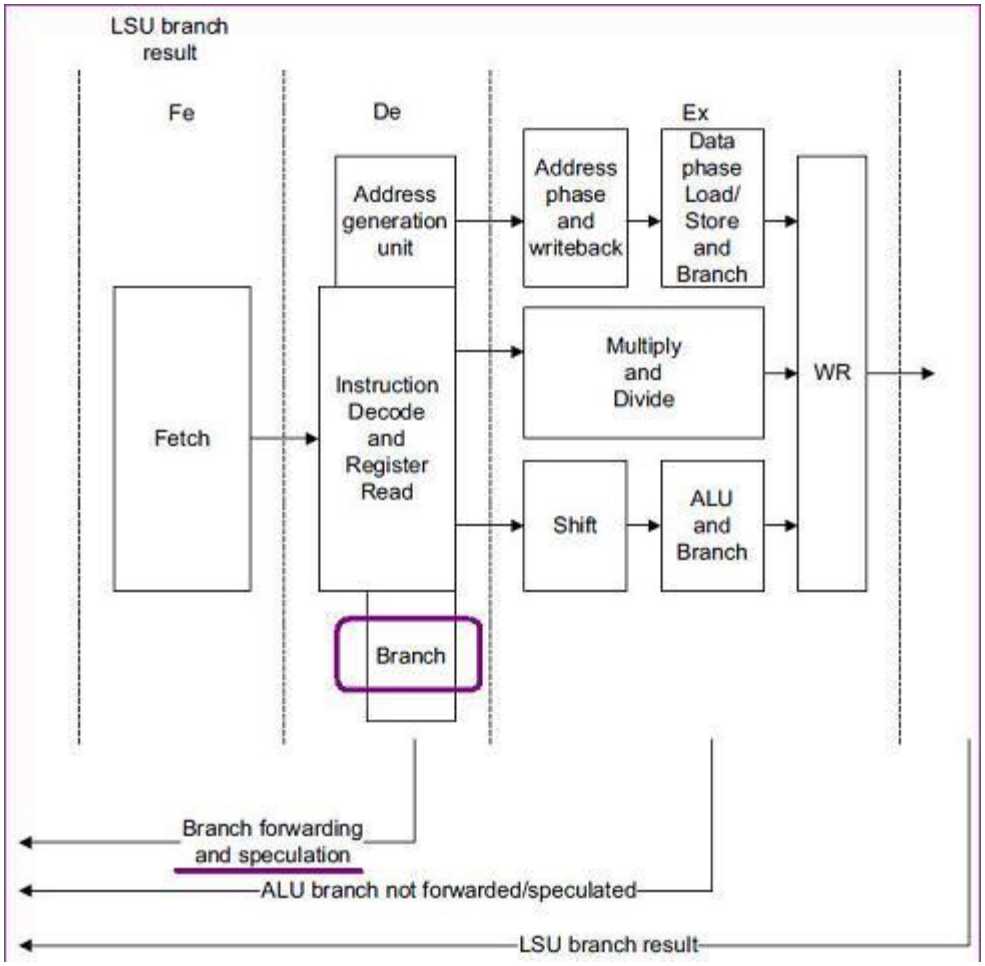
Processor Family	Licenses
Cortex™	69*
ARM11™	72
ARM9™	253
ARM7™	171

Processor	Licenses	Public Licensees
Cortex-A9	9	NEC Electronics, nVIDIA, STMicroelectronics, Texas Instruments, Toshiba, Mindspeed Technologies
Cortex-A8	9	Broadcom Corporation, Freescale Semiconductor, Matsushita, Samsung Electronics, STMicroelectronics, Texas Instruments, PMC-Sierra
Cortex-R4 (F)	14	Broadcom Corporation, Texas Instruments, Toshiba
Cortex-M3	29	Accent Srl, Actel Corporation, Broadcom Corporation, Ember, Energy Micro, Fujitsu, Luminary Micro, NXP, Fuzhou Rockchip Electronics CO., Ltd., STMicroelectronics, Texas Instruments, Toshiba, Zilog,
Cortex-M0	5	NXP, Triad Semiconductor, Melfas

ARM7 Architecture와 비교

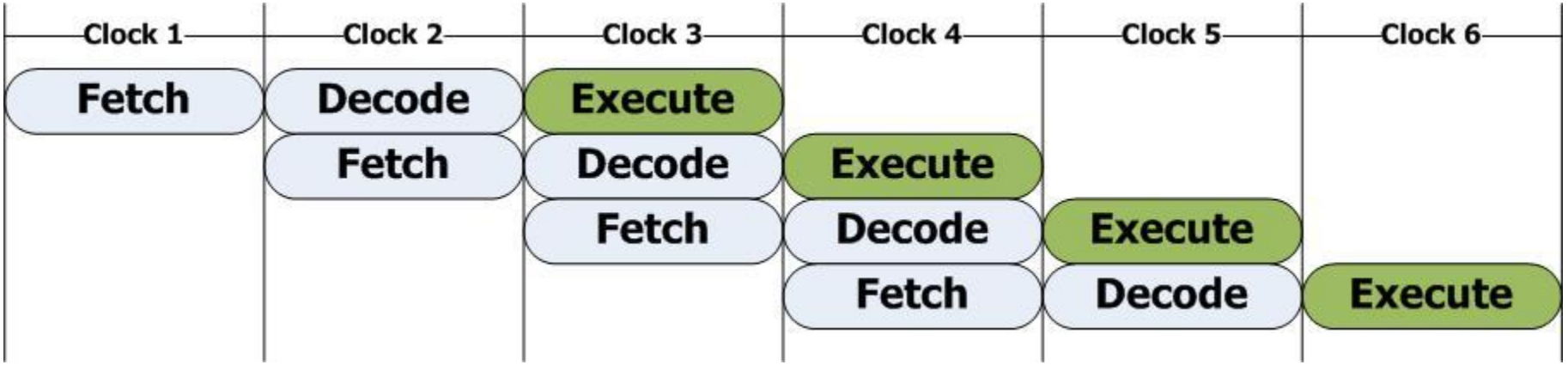
	ARM7TDMI	ARM Cortex-M3
Architecture	ARMv4T (von Neumann)	ARMv7-M (Harvard)
ISA Support	Thumb / ARM	Thumb / **Thumb-2**
Pipeline	3-stage	3-stage + **branch speculation**
Interrupts	FIQ / IRQ	**NMI +1 to 240 physical interrupts**
Interrupt Latency	24 - 42 cycles	**12 cycles**
Inter-Interrupt Latency	24 cycles	**6 cycles**
Sleep Modes	None	Integrated
Memory Protection	None	8 region **MPU**
Dhrystone	0.95 DMIPS/MHz (ARM) 0.74 DMIPS/MHz (Thumb)	**1.25 DMIPS**/MHz
Power Consumption	0.28mW/MHz	**0.19mW**/MHz
Area	0.62mm^2 (Core only)	0.86mm^2 (core + peripherals)

Pipeline Branch Speculation



- 3-stage + branch speculation

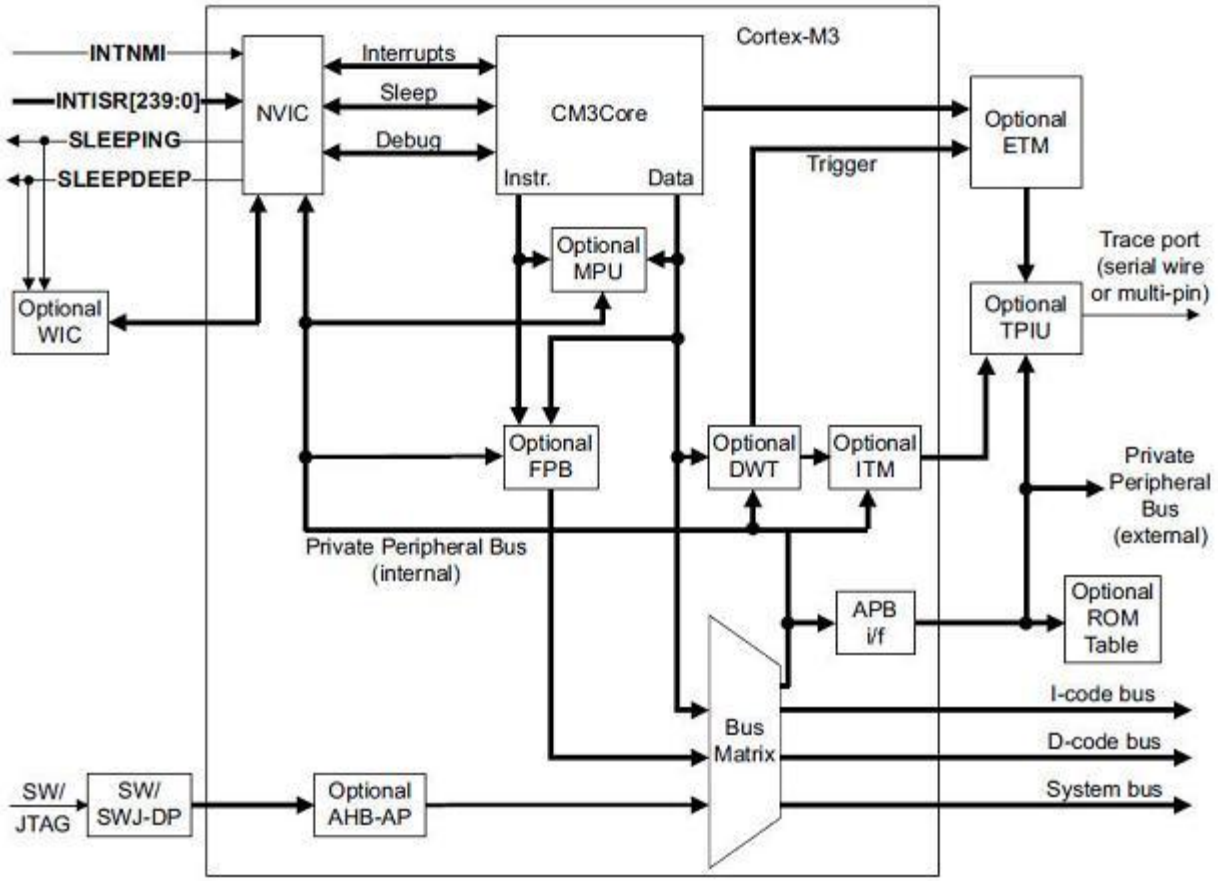
Pipeline Branch Speculation




```
do {
    execution A
    execution B
    C --;
} while (C > 0);
execution D
execution E
```

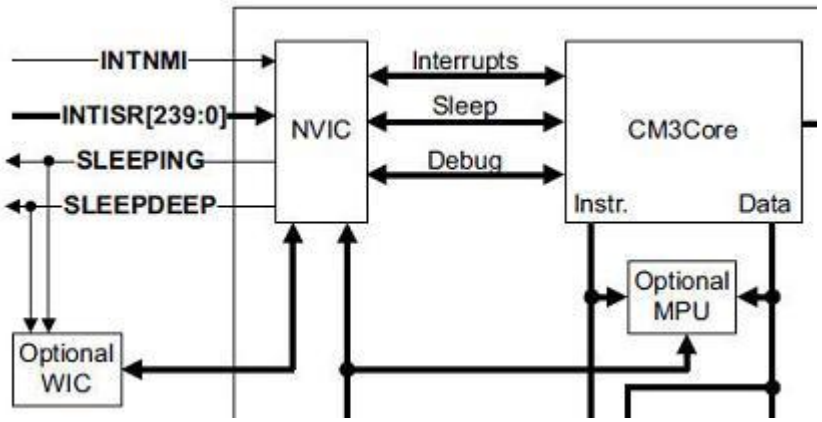
```
Front:
    execution A
    execution B
    C --;
    If(C > 0) goto Front
    execution D
    execution E
```

Cortex-M3 block diagram



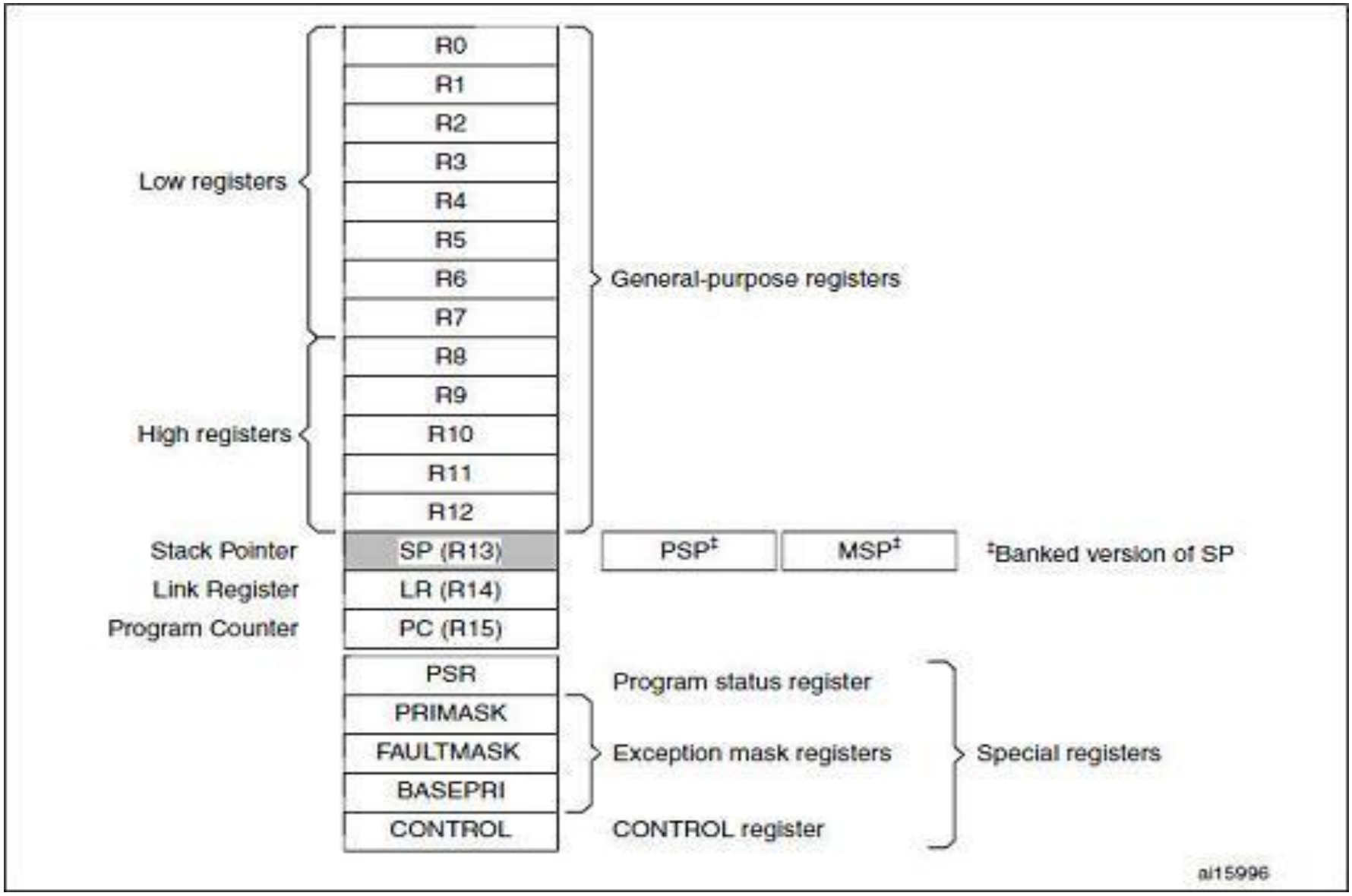
- DWT, ITM, ETM 등 debugging, trace 관련 부분 보강
- STM32의 경우 매우 작고 저렴한 가격을 맞추기 위해서 위의 Optional 부분은 대부분은 들어 있지 않다.

Cortex-M3 interrupt



- interrupt를 발생시키는 소스가 획기적으로 늘어났다.
  - 시스템에서 관장하는 **16**개의 인터럽트를 포함해서 외부 인터럽트로 **240**개까지 할당할 수 있게 되었다. 칩을 만드는 제조사에서 자유롭게 변경 가능한 부분이 될 것이다.
- 가장 중요한 변화는 **Interrupt Latency**의 감소를 얘기할 수 있다.

Registers


R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
SP (R13)
LR (R14)
PC (R15)
PSR
PRIMASK
FAULTMASK
BASEPRI
CONTROL
Low registers
High registers
General-purpose registers
Stack Pointer
Link Register
Program Counter
PSP‡
MSP‡
‡Banked version of SP
Program status register
Exception mask registers
CONTROL register
Special registers
ai15996

General-purpose registers & Stack pointer

<General-purpose registers>

- 범용 레지스터. 대부분의 data 처리 명령어에서 사용.
- Low 레지스터: R0~R7은 범용 레지스터를 사용할수 있는 모든 명령어에서 접근 가능하다.
- High 레지스터: R8~R12는 범용 레지스터를 사용할수 있는 모든 32비트 명령에서 접근 가능하다. **16비트 명령은 접근할수 없다.**

<Stack pointer>

- Stack Pointer (SP)는 register R13이다.
- Thread mode에서 CONTROL register의 bit[1]은 사용될 stack pointer를 아래와 같이 지정한다.
  - 0 = Main Stack Pointer (MSP). 초기 reset value이다.
  - 1 = Process Stack Pointer (PSP).
- **Reset시 processor는 MSP로 address 0x00000000으로부터 값을 load**한다.

Mode Privilege

Processor mode	Used to execute	Privilege level for software execution	Stack used
Thread	Applications	Privileged or unprivileged(1)	Main stack or process stack(1)
Handler	Exception handlers	Always privileged	Main stack

- Processor mode는 **Thread mode와 Handler mode**로 나뉩니다.
  - 프로세서가 보통의 Application Program을 실행하는지 - Thread mode
  - Exception handler를 실행하는지를 결정 - Handler mode
  - 처음 **Reset**이 되었을 때 프로세서는 **Thread mode**에 진입
  - Exception 처리를 끝내면 Thread mode로 되돌아 간다.

- Privilege level은 특권 레벨과 사용자 레벨
  - **Privilege level과 Unprivileged level**로 구분 - 보호를 목적
  - 사용자의 프로그램이 뜻하지 않은 동작으로 중요한 부분에 침해가 발생할 수 있는 가능성을 미연에 방지하려는 것
  - Critical한 영역으로의 메모리 접근을 보호하기 위한 방법 제공

Link register & Program counter

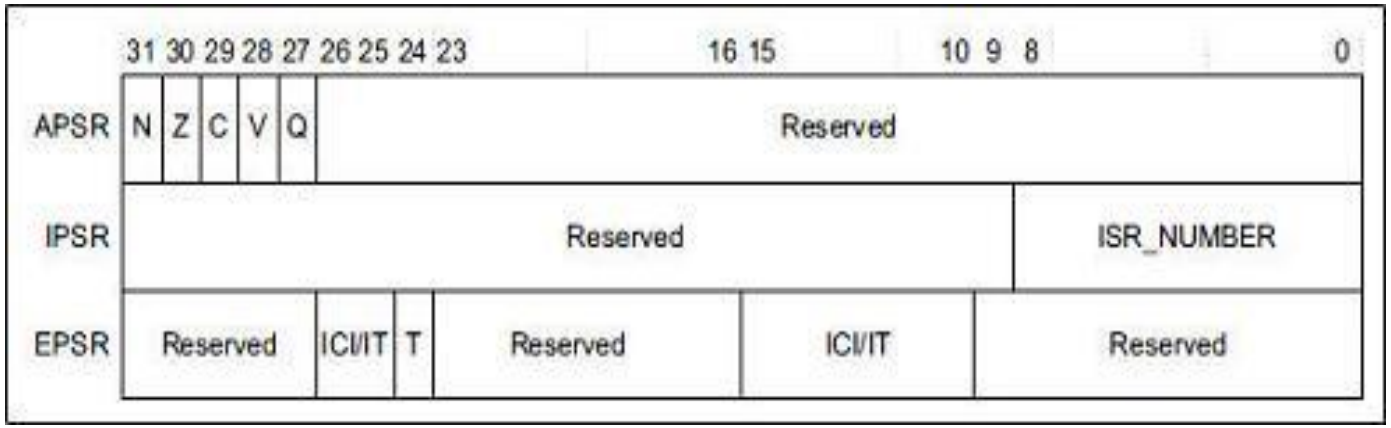
<Link register>

- Link Register (LR)는 register R14이다.
- subroutines, function calls이나 exceptions에 대한 return 정보를 저장하고 있다.

<Program counter>

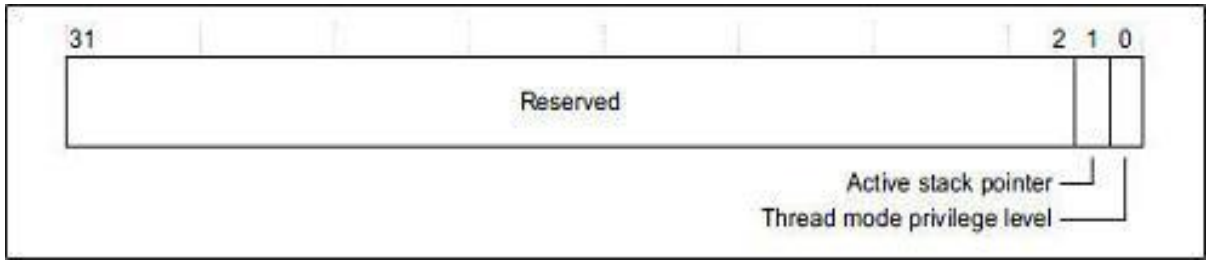
- Program Counter (PC)는 register R15이다.
- 현재의 program 주소값을 가지고 있다.
- instruction fetch가 반드시 halfword aligned되어 있어야 하기 때문에 Bit[0]는 항상 1이다.
- Reset시 processor는 **reset vector의 값을 PC로 load한다. 이때의 주소값은 0x00000004**이다.

Program status register



- Program Status Register (PSR)은 다음으로 구성된다.
  - Application Program Status Register (APSR)
  - Interrupt Program Status Register (IPSR):
    현재 Interrupt Service Routine (ISR)의 exception type 번호 저장
  - Execution Program Status Register (EPSR)
- 3개의 레지스터들은 32-bit PSR에 비트 영역이 서로 겹치지 않는다.
- 이들 레지스터들에 대한 **접근은 개별적으로도 가능하고 둘이나 혹은 모든 3개의 레지스터를 한꺼번에 조합해서도 접근이 가능**하다. 사용하는 명령에 argument로 각각의 이름을 주면된다.

Control Register



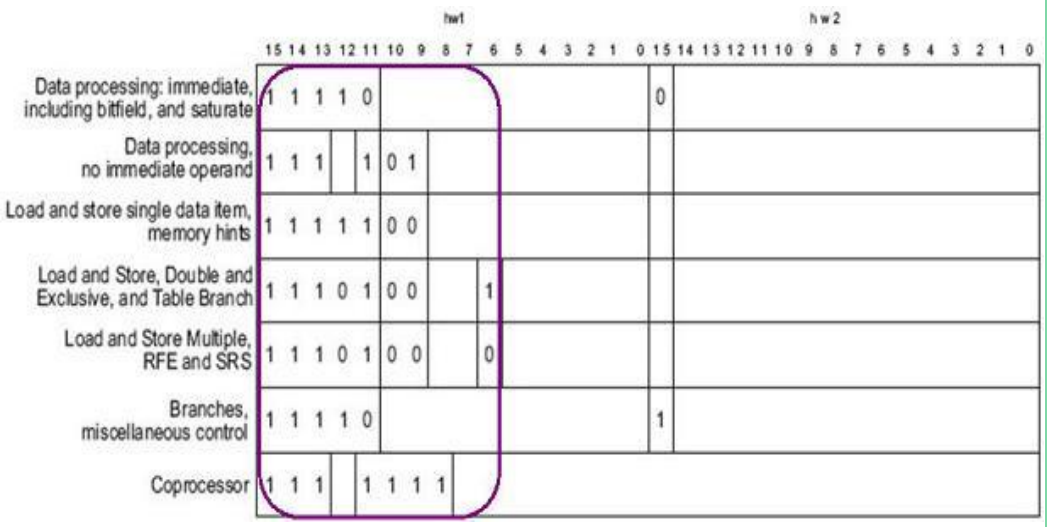
- stack과 Thread mode privilege level을 control한다.
- Bit 1 ASPSEL: Active stack pointer selection, 현재 stack 선택
  - 현재 stack pointer - 0: MSP, 1: PSP
  - Handler mode에서 읽으면 0이고 write는 무시 (항상 MSP 사용하기 때문)
- Bit 0 TPL: Thread mode privilege level
  - 0: Privileged, 1: Unprivileged.
- OS 환경에서, Thread mode에서 동작하는 thread는 process stack 사용하고, kernel과 exception handlers는 main stack을 사용
- Thread mode는 MSP를 사용한다. PSP로 바꾸려고 하면 MSR 명령을 통해서 ASPSEL을 1로 바꾸어야 한다. stack pointer를 바꿀 때 MSR 다음에 **ISB(Instruction Synchronization Barrier)**을 사용해야 한다. 이것은 ISB 이후 명령들이 새로운 stack pointer를 사용해서 수행하도록 해준다. **ISB는 pipeline을 flush한다**.

Exception Mask Register

PRIMASK	nonmaskable interrupt (NMI)와 Hard-Fault를 제외한 모든 인터럽트 mask
FAULTMASK	nonmaskable interrupt (NMI)를 제외한 모든 인터럽트가 발생하지 못하게 한다.
BASEPRI	여기에 적힌 priority level과 같은 것을 포함해서 이보다 큰 값(낮은 priority)을 가진 모든 인터럽트 mask

Thumb-2

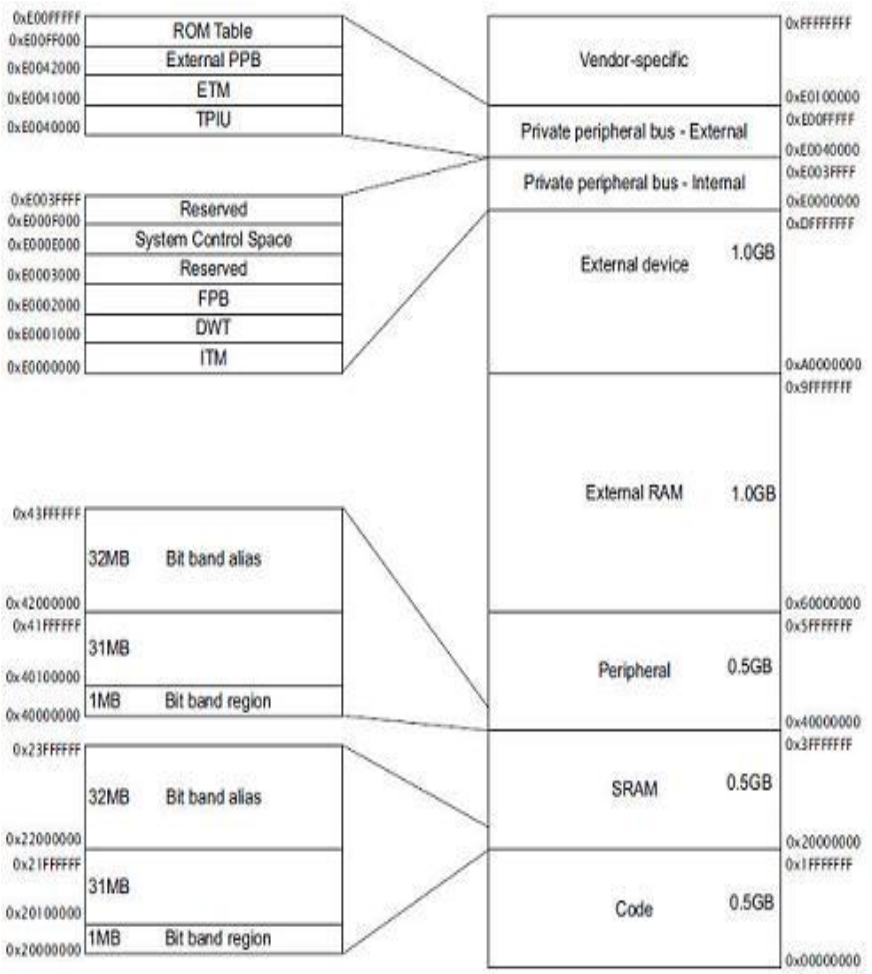
32-bit Thumb instruction, hw1				32-bit Thumb instruction, hw2			
15	8	7	0	15	8	7	0
Byte at Address A+1		Byte at Address A		Byte at Address A+3		Byte at Address A+2	



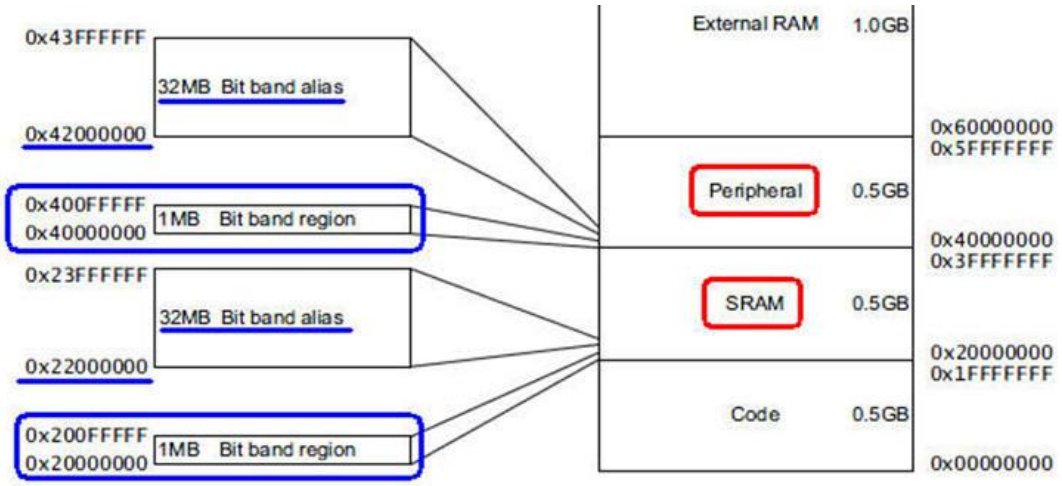
- Thumb-2에 대해 처음 소개된 것은 ARMv6T2에서이다.
- M profile에서 새로운 모델 시도 - 오직 Thumb mode에서만 동작
- 모든 instructions에 대해서 16-bit alignment를 강제한다.
- Thumb-2는 **16 비트 명령과 32 비트 명령을 모두 수행**하게 된다.
- hw1이 low address이고 hw2가 high address가 된다.
  - hw1,2 각각 16 비트 명령이거나, 둘을 합쳐 하나의 32 비트 명령
- 맨 상위 비트가 111인 경우는, 아래 16 비트를 다르게 처리
  - 하나의 예외가 상위 비트가 111이고 그 다음 비트 2개가 00이면 Unconditional branch로 처리하는 16 비트 명령이다.

Memory Map

- 미리 정의된 **Memory Map**
- 단순하고 고정된 memory map을 4 Giga bytes의 주소 공간을 할당해서 미리 정의를 내려놓고 있는 것이다.
- Code (code space), SRAM (memory space), external memories/devices와 internal/external peripherals까지 모든 영역에 대해서 고정된 주소 영역이 할당 되어 있다.
- 칩을 만드는 제조사에서 업체에서 특별히 관리해 주어야 하는 영역에 사용할 수 있도록 특별한 영역인 **vendor specific** 부분도 존재한다.

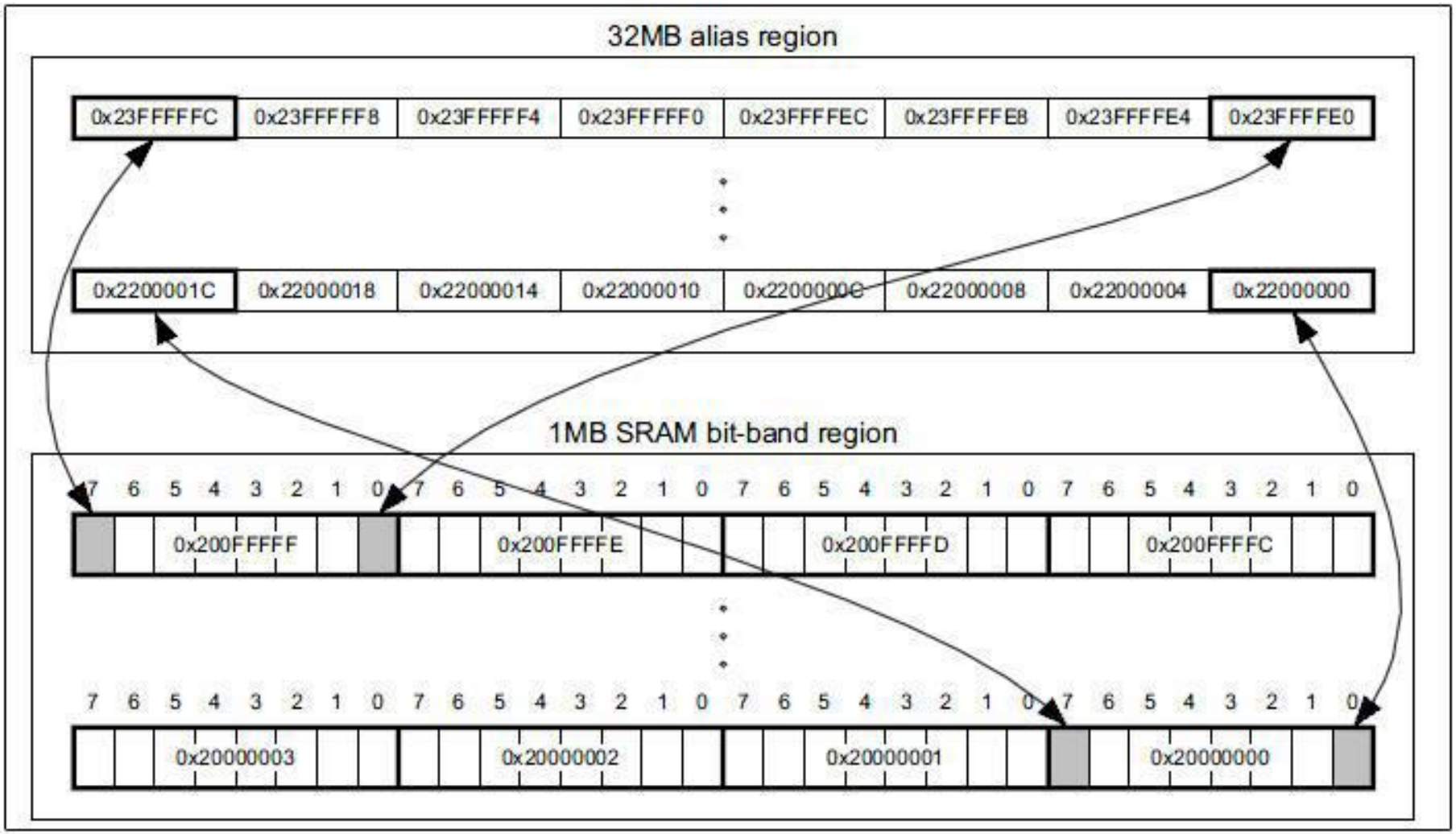


Bit banding



- bit operation을 빠르게 해줄 수 있는 방법 제공
  - ➢ 하드웨어적으로 특정 비트를 각각 특정 어드레스에 할당 해놓고 있다.
- **32 MB alias 영역에 쓰는 행위는 bit-band 영역 특정 비트에 영향**
- 0x20000000 번지의 1번 비트 - 0x22000004 번지에 할당
  - ➢ 32 비트 (4 바이트)로 증가되기 때문에 0x22000004 번지로 4 만큼 증가
  - ➢ 0x20000004 번지의 0번 비트 alias 영역? 4 X 32 만큼 떨어진 곳. 4 X 32 = 128, 즉 0x80. 0x22000080 번지가 0x20000004 번지의 0번 비트
- bit_word_addr = bit_band_base + (byte_offset x 32)
  + (bit_number × 4)

Bit banding


32MB alias region
0x23FFFFFC
0x23FFFFF8
0x23FFFFF4
0x23FFFFF0
0x23FFFFEC
0x23FFFFE8
0x23FFFFE4
0x23FFFFE0
0x2200001C
0x22000018
0x22000014
0x22000010
0x2200000C
0x22000008
0x22000004
0x22000000
1MB SRAM bit-band region
7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
0x200FFFFF
0x200FFFFE
0x200FFFFD
0x200FFFFC
7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
0x20000003
0x20000002
0x20000001
0x20000000

Exception 종류

위치	Exception type	Priority	설명
0	**-**	-	리셋 시 vector table의 첫 entry
1	**Reset**	–3 (highest)	Power up과 Warm reset시.
2	**Non-maskable Interrupt (NMI)**	-2	pre-empted되지 않는다.
3	**Hard Fault**	-1	모든 종류의 Fault
4	**Memory Management**	Configurable	MPU mismatch
5	**Bus Fault**	Configurable	Pre-fetch, memory fault
6	**Usage Fault**	Configurable	Undefined instruction
7~10	**-**	-	Reserved
11	**SVCall**	Configurable	System service 호출
12	**Debug Monitor**	Configurable	not halting Debug monitor
13	**-**	-	Reserved
14	**PendSV**	Configurable	Pendable request
15	**SysTick**	Configurable	System tick timer.
16~255	**External Interrupt**	Configurable	core 밖 외부 interrupt (0~239)

Exception Priority & Nested 처리

- NVIC(Nested Vectored Interrupt Controller): 각종 인터럽트를 효율적으로 제어 - 모든 Exception의 우선순위를 결정하고 처리한다.
- **Nested 처리** - 하나의 인터럽트가 처리되고 있는 중에 보다 높은 우선순위의 인터럽트 발생시 처리되던 인터럽트는 잠시 멈추고 높은 우선순위의 인터럽트가 처리될 수 있다.
- **Interrupt 번호가 낮을 수록 높은 Priority**를 가진다. Reset은 가장 높은 Priority를 가진다. 우선순위 -3, -2, -1을 가지는 Reset, Non-maskable Interrupt (NMI), Hard Fault는 우선 순위를 변경할 수 없다. 고정된 우선 순위를 가지고 있고, 가장 높은 우선순위를 가진다.
- 하나의 우선 순위를 공유할 수도 있다. 이를 위해 **Preempting Priority와 Subpriority**를 두어서 그룹으로 만들어서 관리하는 방법을 제시한다.
- 15개의 exception을 포함해서 사용자 Interrupt가 240개까지 지원되기 때문에 ARM7이나 ARM9에서 사용하던 Supervisor 모드 및 다양한 모드들이 2가지 모드로 줄었다. Privileged Mode와 Non-Privileged 모드 두 개 뿐이다.

Preempting Priority & Subpriority

PRIGROUP[2:0]	Binary point position	Pre-emption field	Subpriority field	Number of pre-emption priorities	Number of subpriorities
b000	bxxxxxxx.y	[7:1]	[0]	128	2
b001	bxxxxxx.yy	[7:2]	[1:0]	64	4
b010	bxxxxx.yyy	[7:3]	[2:0]	32	8
b011	bxxxx.yyyy	[7:4]	[3:0]	16	16
b100	bxxx.yyyyy	[7:5]	[4:0]	8	32
b101	bxx.yyyyyy	[7:6]	[5:0]	4	64
b110	bx.yyyyyyy	[7]	[6:0]	2	128
b111	b.yyyyyyyy	None	[7:0]	0	256

PRIGROUP [2:0]	Interrupt priority level value, PRI_*M*[7:4]			Number of	
	Binary point(1)	Group priority bits	Subpriority bits	Group priorities	Sub priorities
0b100	0bxxx.y	[7:5]	[4]	8	2
0b101	0bxx.yy	[7:6]	[5:4]	4	4
0b110	0bx.yyy	[7]	[6:4]	2	8
0b111	0b.yyyy	None	[7:4]	1	16

Vector Table은 Address

- ARM7, ARM9의 경우 Vector Table은 명령어이었다. 하지만 Cortex-M3의 경우는 명령이 아닌 주소 값이다.
- Vector Table에 명령이 존재하게 될 경우에의 문제점은 interrupt의 지연 시간이다. 인터럽트가 발생했을 때 해당 테이블에 존재하는 명령을 fetch해서 가져오면 거기에 다시 분기 명령이 들어있게 되고, 분기하는 처리를 다시 해야 하기 때문에 상당한 지연시간이 발생하게 된다.
- Cortex-M3에서는 직접 주소값을 넣어두고 그것을 PC 값으로 바로 복사함으로써 분기에 따른 지연시간이 상당 부분 감소하게 된다. 과거에도 VIC (Vectored Interrupt Controller)라는 개념으로 이것을 구현한 경우도 있었다. 이를 하드웨어적으로 기본으로 탑재하게 하면서 nested 기능까지 추가한 것이라고 보면 된다.
- Reset에 대한 **Exception Address가 0x0번지가 아니고 0x4번지**이다. **0x0번 번지에는 초기 Stack Address가 들어간다**.
- Cortex-M3는 **재설정 가능한 (re-locatable) vector table** 사용.
  - SCB 레지스터를 이용해서 Vector Table이 저장될 주소와 위치 변경 가능
  - RAM 공간에 넣을 수도 있고, Flash 영역에 저장할 수도 있다.

Wait For Interrupt (WFI), Wait For Event (WFE)

- NVIC는 Cortex-M3 processor의 power-management scheme을 위한 부분도 구현하고 있다.
- Wait For Interrupt (WFI)나 Wait For Event (WFE) instructions이 호출되면 Core는 low-power 상태로 되면서 exception이 발생하기를 기다리게 된다. 다른 exception이 발생할 때까지 Core는 Sleep 상태에 있게 된다.

Symbol	Parameter	Conditions	Typ(1)		Max		Unit
			V_{DD}/V_{BAT} = 2.4 V	V_{DD}/V_{BAT} = 3.3 V	T_A = 85 °C	T_A = 105 °C	
I_{DD}	Supply current in Stop mode	Regulator in Run mode, low-speed and high-speed internal RC oscillators and high-speed oscillator OFF (no independent watchdog)	23.5	24	200	370	µA
		Regulator in Low Power mode, low-speed and high-speed internal RC oscillators and high-speed oscillator OFF (no independent watchdog)	13.5	14	180	340	
	Supply current in Standby mode	Low-speed internal RC oscillator and independent watchdog ON	2.6	3.4	-	-	
		Low-speed internal RC oscillator ON, independent watchdog OFF	2.4	3.2	-	-	
		Low-speed internal RC oscillator and independent watchdog OFF, low-speed oscillator and RTC OFF	1.7	2	4	5	
I_{DD_VBAT}	Backup domain supply current	Low-speed oscillator and RTC ON	1.1	1.4	1.9(2)	2.2	

Hardware Stack 처리

- ARM7 프로세서의 경우 여러가지 exception mode가 존재하고 각각의 경우마다 banked 레지스터들이 존재했었다. 하지만 Cortex-M3에서는 이러한 부분들을 거의 없애버리고 단순화시켜서 gate count를 줄이면서 시스템의 복잡도도 줄였다. 대신 이러한 방식을 대신하는 방법으로 **stack base의 exception 모델**을 제시하고 있는 것이다.
- exception이 발생했을 때 프로세서는 자동으로 하드웨어적으로 **Program Counter, Program Status Register, Link Register** 그리고 **R0-R3, R12**의 일반 레지스터를 스텍에 저장한다.
- 이러한 저장 과정은 instruction bus가 vector table로부터 exception vector를 인지하고 exception code의 첫 번째 instruction을 fetch하는 동안에, data bus가 레지스터를 stack에 저장하게 된다.
- 동시에 작업을 진행함으로 인해서 **interrupt**가 발생하였을 때의 지연시간을 극소화시키는 것이다.
- hardware가 stack의 처리에 대한 것을 담당함으로써 전통적인 C 함수에서 ISR routine의 경우에 특별히 처리해주던 stack 처리와 관련한 assembler wrappers를 추가하던 부분이 Cortex-M3 processor에서는 필요 없어진 것이다.

ARM7,9 ISR routine 선언 예

```
#ifdef __GNUC__ /* gcc */
#ifdef TEST_FIQ
void Timer1_InterruptServiceRoutine(void) __attribute__ ((interrupt("FIQ")));
#else
void Timer1_InterruptServiceRoutine(void) __attribute__ ((interrupt("IRQ")));
#endif
#endif

#ifdef __ARMCC_VERSION /* armcc (for ADS 1.2) */
void __irq Timer1_InterruptServiceRoutine (void)
#else // __GNUC__ /* gcc */
void Timer1_InterruptServiceRoutine (void)
#endif
```

- ADS V1.2에서는 __irq라는 컴파일러 지시어를 사용하고 GCC는 __attribute__ ((interrupt("IRQ")))와 같은 지시어를 사용한다.
- 지시어는 컴파일러로 하여금 이것이 interrupt service routine이라는 것을 알수 있게 하는 것이다. 일반 함수들이 호출되는 과정과 ISR에서의 stack에 대한 처리를 구분해서 컴파일 할 수 있도록 해야 한다.
- Cortex-M3에서는 하드웨어가 stack에 대한 부분을 처리해주기 때문에 이러한 지시어가 필요 없어진 것이다.

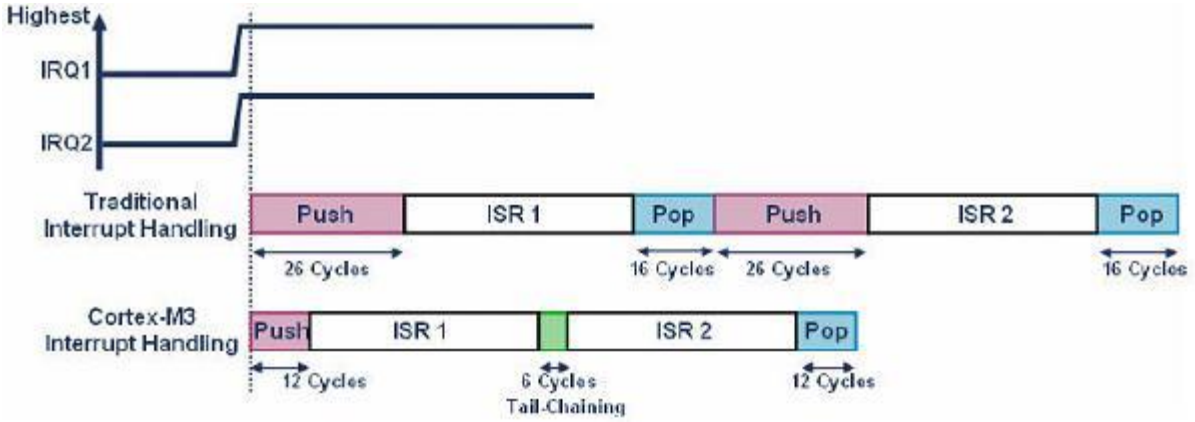
Interrupt Latency 감소 3가지 방법

- **Tail chaining, late arrival, pop pre-emption**

- Nested에 대한 지원으로 인해서 이미 처리되고 있는 interrupt가 보다 더 높은 우선순위를 갖는 interrupt에 의해서 언제든 중단될 수 있다.
  - 우선 순위에 대한 것이 영구적으로 고정된 것이 아니라 소프트웨어적으로 변경하는 것도 가능하다.

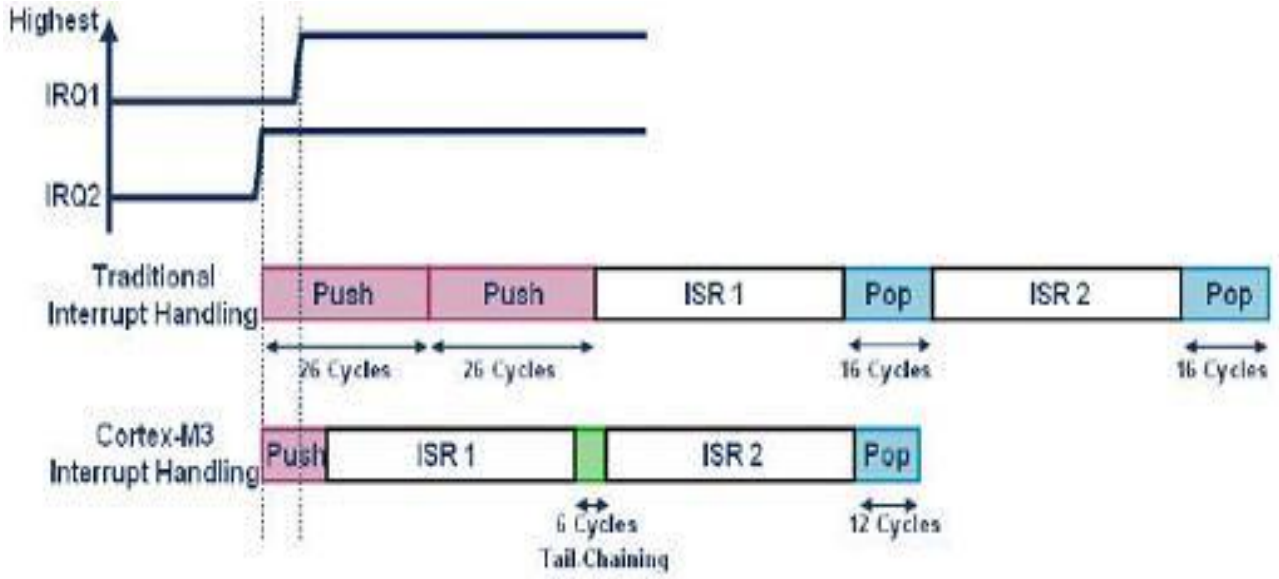
- 인터럽트가 현재 처리 중인 상태에서 보다 높은 우선순위의 인터럽트에 의해서 중단되는 경우는 어쩔 수 없이 지금 수행 중이던 인터럽트의 상태를 보존하기 위한 Stack push와 높은 우선순위의 인터럽트 종료 후에 Stack pop을 수행할 수밖에 없다. 하지만 **그런 상황이 아닌 경우에 타이밍 상으로 보다 빠르게 인터럽트를 처리할 수 있는 방안들**이 여러 개 있을 수 있다.

Tail Chaining



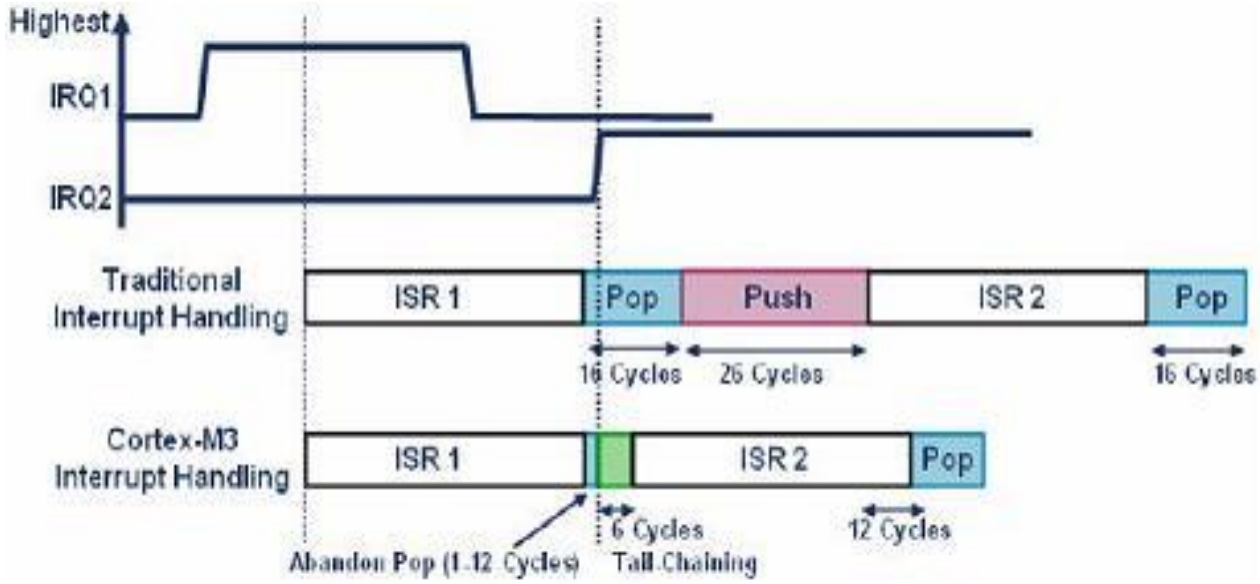
- back-to-back interrupts: 동시에 여러 인터럽트가 쌓여 있는 경우
- 전통적인 방식에서는 Stack push pop이 2회 발생하게 되지만 tail-chaining 방법을 이용해서 이 횟수를 한번으로 줄이는 것이다.
  - 전통적인 방식에서 두 인터럽트의 처리 부분에 대한 실행시간을 제외하고 오버헤드로 붙는 클럭 수가 총 84 cycle이 된다. 하지만 tail-chining을 사용하는 Cortex-M3에서는 30 cycle로 무려 **54 cycle을 절약**하게 되는 것이다.
- NVIC hardware 내에 구현되어 있어서 보다 빠르고 쉽게 사용할수 있도록 해주고 있는 것이다.
- 100 MHz 이하의 성능을 가지는 micro-processor 시장에서 이 부분은 획기적인 성능 향상을 가져온다.

Late Arrival



- 높은 우선순위를 가지는 인터럽트가 더 늦게 도착한 경우
- 낮은 우선순위를 가지는 IRQ2가 먼저 발생을 했고 이를 위해서 현재의 프로세서 상태를 저장하기 위한 Push 작업을 수행하고 있는 동안 **(즉, 아직까지 IRQ2의 ISR2가 실행되지 않은 상태인 것이 중요하다)** 이때 보다 높은 우선순위를 가진 IRQ1이 발생하게 되면, 이전에 수행하던 Push 작업은 그대로 수행을 하면서 Push 작업이 끝난 이후에 먼저 발생한 ISR2를 수행하는 것이 아니고 ISR1을 먼저 수행한다.
- 이후의 동작은 tail-chaining과 같이 ISR1에 대한 수행을 이어서 한다.

Pop pre-emption



- ISR 동작을 종료한 이후 이전 상태를 복구하기 위한 **Pop을 수행하고 있는 시점에 다른 IRQ**가 발생하는 경우
- 만약 **Pop을 종료하지 않은 상황**이라면 이전의 Pop이 되고 있던 상황 자체를 무시하고 Stack pointer도 원래의 자리로 옮기고, 새로 들어온 interrupt를 처리하게 된다.
- 이 경우 지연 시간 부분에서 Pop을 수행하던 cycle이 얼마나 진행되고 있었는가에 따라서 추가되는 cycle의 수는 달라질 수 있다.

기타 특징

- H/W divide와 single cycle multiply
  - 32 비트 곱셈을 한 cycle에 수행할 수 있도록 제공
  - Hardware divide ALU 제공 – 2~12 cycle 내에 처리 가능한 divider 포함
- 완전히 C로만 구현하는 Firmware
  - 이유는 Stack에 대한 주소가 0x0번지에 들어가 있게 됨으로써 ARM7, ARM9에서의 초기화 작업이 C 언어로도 가능해졌기 때문이다.
  - Reset 상태에서 Stack 주소가 초기화 되어있지 않기 때문에 Stack을 초기화하고 Startup과 관련한 code를 실행해야 하고 (이것을 어셈블리 언어로 작성해야 했다) 비로소 C로 작성된 main 함수로 진입할수 있었다. Stack이 초기화되지 않고서는 C로 구현된 내용이 처리될 수 없기 때문이다.
  - Cortex-M3의 경우 **0x0번지에서 32 비트의 값을 가져와서 Stack 주소를 초기화 한 이후에 Reset을 하기 때문**에 이것이 가능해진 것이다.
- Unaligned data access
  - ARM7에서 align이 맞지 않으면 Abort exception이 발생
  - Cortex-M3에서는 4 바이트로 align이 되어있지 않은 데이타 구조에서도 access가 가능하도록 설계되어 있다.
  - 물론 align이 맞지 않는 접근을 하면 속도에서의 손실을 볼수 밖에 없다.