

ARM 프로세서의 이해

목 차

1 Micro Processor Overview

- 1. 1 ALU, CPU, Peripheral
- 1. 2 Bus system
- 1. 3 Processor
- 1. 4 Cache Memory, write buffer
- 1. 5 System On a Chip

2 Arm Processor Overview

- 2. 1 ARM cpu란 무엇인가?
- 2. 2 CISC vs RISC
- 2. 3 Pipeline
- 2. 4 Samsung ARM ROADMAP

3 Programming Model

- 3. 1 ARM/THUMB Mode
- 3. 2 Little/Big Endian
- 3. 3 Registers, CPSR, SPSR
- 3. 4 Exceptions
- 3. 5 Operating modes

4 Instructions

- 4. 1 ARM & THUMB Instruction
- 4. 2 Data processing instruction
 - 4. 2. 1 Arithmetic
 - 4. 2. 2 Bit-wise logical
 - 4. 2. 3 Register Movement
 - 4. 2. 4 Comparison
 - 4. 2. 5 Immediate
 - 4. 2. 6 Shifted register
 - 4. 2. 7 Multiplies
- 4. 3 Data transfer instruction
 - 4. 3 .1 Single register load/store
 - 4. 3. 2 Multiple register data transfer
 - 4. 3. 3 Addressing Mode
 - 4. 3. 4 Stack control
 - 4. 3. 5 Swp
- 4. 4 Flow control instruction
 - 4. 4. 1 Branch
 - 4. 4. 2 Conditional branch
 - 4. 4. 3 Branch and link
 - 4. 4. 4 Supervisor call
- 4. 5 기타 명령어

5 Psedo Instruction

- 5. 1 Psedo Instruction
- 5. 2 Directive
- 5. 3 Data Definition
- 5. 4 Macro
- 5. 5 Symbol Definnition

1장 Micro Processor Overview



개요

1장에서는 ARM 프로세서의 학습에 도움이 되는 기본적인 내용 중에서 중요한 사항 몇 가지를 설명합니다. 이 장에서 학습하는 일반적인 프로세서에서의 동작과 ARM 프로세서의 동작을 비교해서 이해하면 보다 깊이있고 폭넓은 학습을 도모하리라 생각합니다.

학습목표

내장형 시스템(Embedded System)에 필요한 기반지식을 습득합니다.

학습내용

1. ALU, CPU, Peripheral
2. Bus System
3. Processor
4. Cache, Write Buffer
5. System On a Chip

1.1 ALU, CPU, Peripheral

●ALU(Arithmetic Logical Unit)

: ALU는 프로세서 내부 회로 중 산술, 논리연산처리를 수행하는 장치입니다.

산술연산은 덧셈, 뺄셈 기타 곱셈, 나눗셈 연산을 말합니다. ARM 프로세서에서는 대다수의 DSP와 같이 나눗셈 명령어가 없습니다. 논리 연산은 AND, OR, XOR, NOT 등의 연산을 말합니다.

●중앙처리장치(CPU)

: 디지털 컴퓨터에서 가장 핵심적인 구성요소로, 목적은 메모리에서 전송된 명령어를 디코드하고, 내부 레지스터나 메모리 또는 I/O 인터페이스 장치에 저장된 데이터를 전송하고, 산술, 논리연산과 제어를 수행함에 있습니다.

CPU가 외부에 있는 다른 구성요소들과 명령어나 데이터, 제어정보를 주고 받기 위해서는 1개 이상의 버스를 사용합니다.

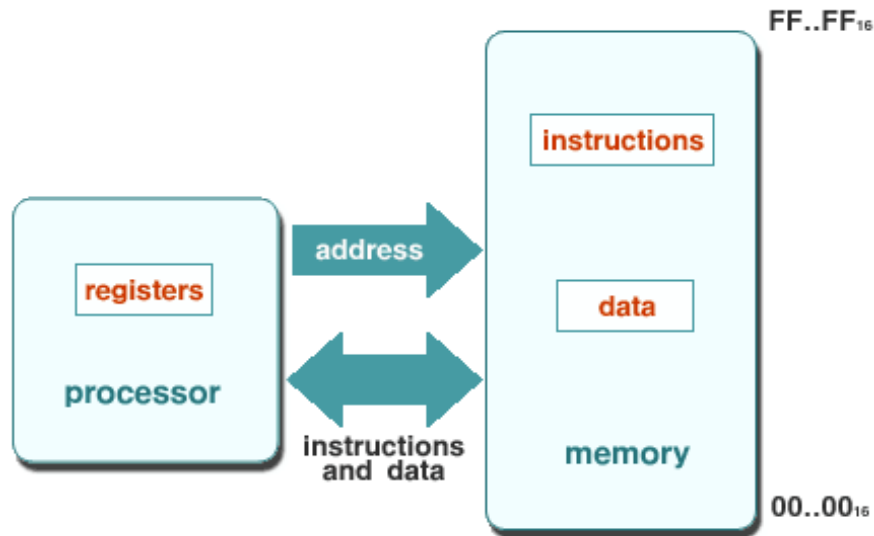
●주변장치(peripherals)

: CPU가 직접 제어하는 디바이스를 온라인으로 연결되었다고 말하는데, 이 디바이스가 CPU와 직접 통신하거나 CPU의 명령에 따라서 메모리로 2진 정보를 주고 받습니다. 이때 컴퓨터에 온라인으로 연결된 I/O 디바이스를 주변장치라고 합니다.

1.2 Bus System

●버스 시스템(Bus System)

: 버스 시스템은 어드레스 버스, 데이터 버스, 제어 신호로 구성됩니다. 해당 주소와 데이터를 버스에 싣고 제어 신호를 가하면, CPU 외부에 있는 소자의 해당 주소에 데이터를 읽거나 쓸 수 있습니다. 버스 시스템의 구성에 따라 시스템의 동작여부와 성능이 결정되므로, 아주 중요한 요소라 할 수 있습니다.

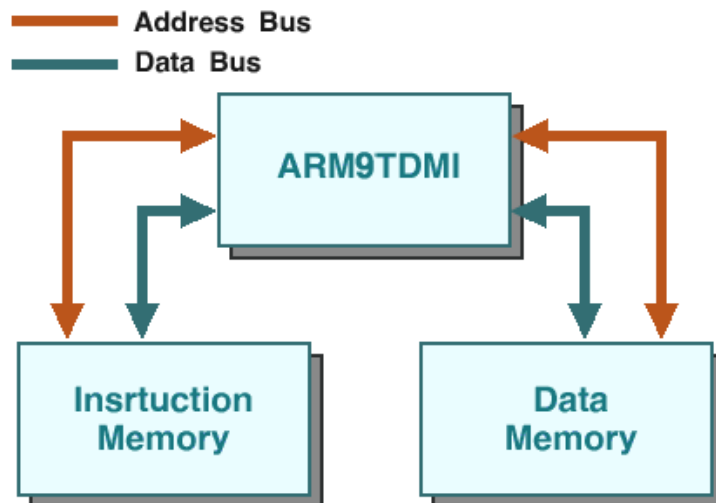


● 하버드 아키텍처

: 하버드 아키텍처는 명령어와 데이터 영역이 분리되어 있는 형태입니다. 즉 명령어를 위한 어드레스, 데이터 버스와 데이터를 위한 어드레스, 데이터 버스가 분리되어 있는 버스 시스템을 사용합니다.

명령어와 데이터의 참조를 동시에 수행할 수 있으므로, 성능 면에서 우수합니다. 단점은 버스 시스템이 복잡하며, 외부 메모리가 2세트가 있어야 한다는 점입니다.

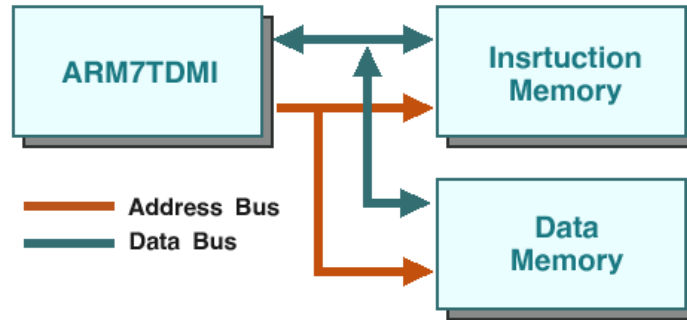
ARM9은 고성능을 얻기 위하여 하버드 아키텍처를 채택하였습니다.



● 폰노이만 아키텍처

: 폰 노이만 아키텍처는 명령어와 데이터를 위한 어드레스, 데이터 버스가 하나입니다.

즉 외부 메모리 공간에 명령어와 데이터가 혼재해 있다는 것을 의미합니다. 개인용 컴퓨터는 폰노이만 아키텍처를 사용하고 있으며, ARM7 또한 폰노이만 아키텍처를 채택하고 있습니다.



1.3 Processor

●마이크로 컨트롤러(Micro-controller)

: 제한된 특수한 용도로 동작시키기 위해서 마이크로 컨트롤러라고 불리는 작고 비교적 간단한 컴퓨터가 표준 컴퓨터나 다른 디지털 시스템에 사용됩니다.

예를 들면, 어떤 마이크로 컨트롤러는 컴퓨터에 있는 키보드나 모니터에서만 사용됩니다.

마이크로 컨트롤러는 명령어의 길이도 보통 4-8비트로 짧으며, 레지스터의 수도 적고 명령어 세트도 한정되어 있습니다.

상대적으로 표현하면 성능은 낮으나 특정한 목적에 알맞는 성능을 갖고 있다고 할 수 있습니다.

인텔이 최초로 출시한 것으로, 기존의 시스템은 CPU와 메모리(ROM, RAM 등), 기타 주변장치가 각각 별도로 되어있었으나, 이를 하나로 통합한 것. 전자식 계산기를 개발하는 일본업체에 납품하기 위해 개발되었으나 일반 목적으로 사용하도록 수정하여 출시한 것이 그 효시라고 할 수 있습니다.

●마이크로 프로세서(Micro Processor)

: 마이크로 컨트롤러 보다는 상대적으로 동작속도도 빠르고 복잡한 CPU를 의미합니다.

통상 마이크로 컨트롤러와 같이 많은 주변 장치들을 내장하지는 않으며, 인터페이스할 수 있는 외부 메모리의 용량이 큼니다.

특정목적보다는 일반적인 용도로 널리 사용될 수 있는 것을 의미합니다.

●내장형 프로세서(Embedded Processor)

: 개인용 컴퓨터가 사용되는 분야가 아닌 제어, 통신, 가전제품 등에 사용되는 프로세서를 내장형 프로세서라고 부릅니다. 기존의 마이크로 컨트롤러, 마이크로 프로세서 등을 모두 포함해 통칭하는 경향이 큼니다. 반도체 기술의 발전에 따라 마이크로 컨트롤러보다 높은 집적도를 보이는 "SoC" 형태가 대다수이며, 성능 면에서는 기존의 마이크로 프로세서에 필적합니다.

●DSP(Digital Signal Processor)

: 군사용으로 개발되었던 것이 이후 신호처리를 위한 특수 목적의 전용 프로세서로 발전한 것입니다.

신호처리를 위한 연산능력이 뛰어나며, 개인용 컴퓨터용 프로세서, 내장형 프로세서보다 훨씬 앞서 GHz 대역의 성능의 제품이 출시되었습니다. 현재는 멀티미디어, 통신 등의 분야에 많이 쓰이고 있습니다. ARM 프로세서에서는 DSP의 대표적인 특징인 MAC(Multiply and Accumulate) 명령어를 지원합니다.

1.4 Cache Memory, Write Buffer

● 캐시 메모리

: 캐시 메모리는 ARM 프로세서 내부에 있는 특수 목적의 메모리로, **가장 최근에 참조되었던 혹은 참조될 가능성이 있는 명령어, 데이터를 저장하는데 사용합니다.** CPU는 읽고자 하는 데이터가 캐시 메모리에 있는 경우, 외부 메모리를 참조하지 않고 캐시에서 데이터를 읽어 오게 됩니다. 캐시 메모리의 속도는 CPU와 동일하게 빠른 속도로 동작하므로, 속도가 상대적으로 느린 외부 메모리를 참조하는 경우보다 더욱 빠르게 동작합니다. 결과적으로 캐시 메모리를 참조하는 경우가 많을수록 명령어, 데이터를 읽는 동작속도는 최고성능을 발휘하게 되는 것입니다.

● 라이트 쓰루(Write Through)

: CPU가 특정 주소에 데이터를 쓰기 동작하는 경우를 고려해 봅시다. 캐시 메모리 안에 이미 해당 데이터가 존재하는 경우, 라이트 쓰루 방식은 캐시와 외부 메모리에 모두 데이터를 쓰기 동작합니다. 즉, 쓰기 동작 시 캐시는 성능향상에 아무런 도움이 되지 못합니다.

● 라이트 백(Write Back)

: CPU가 특정 주소에 데이터를 쓰기 동작하는 경우를 고려합니다. 캐시 메모리 안에 이미 해당 데이터가 존재하는 경우, 라이트 백은 캐시에만 데이터 쓰기 동작을 수행합니다. 따라서 캐시와 외부 메모리에 있는 데이터의 내용이 달라 불일치가 발생합니다. 이후 해당 데이터를 읽는 동작을 할 경우에는 캐시의 데이터를 참조합니다. 캐시에 쓰인 데이터는 캐시에서 제거될 때 외부 메모리에 쓰여지게 되어, 이전에 발생했던 캐시와 외부 메모리의 데이터 내용 불일치를 해결하게 됩니다.

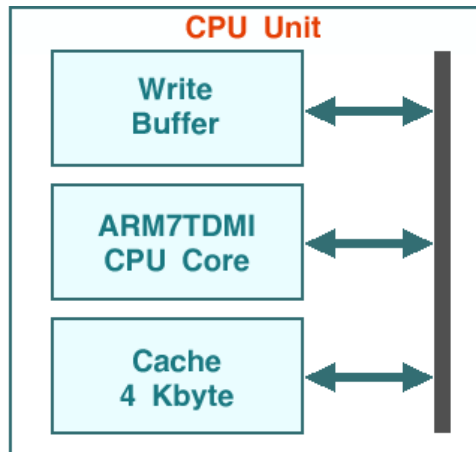
● 캐시 일치(Cache Coherency)

: 캐시는 읽기 동작이 수행될 경우를 고려한 장치입니다. 쓰기 동작이 수행이 될 경우, 데이터의 내용을 외부 장치에 쓰는 동작은 캐시 메모리의 종류에 따라 다릅니다. 특정상황 하에서는 동일한 주소의 데이터가 캐시와 외부 메모리에 존재할 수 있는데, 이때 캐시에 저장된 데이터의 내용과 외부 메모리에 저장된 데이터의 내용이 불일치할 경우를 캐시 불일치라 부릅니다. 데이터를 참조하는 CPU가 이를 잘못하여 참조하면, 다른 내용의 데이터를 참조할 수 있으므로 이를 해결하기 위한 방법을 강구해야 합니다.

: Samsung ARM의 경우 라이트 쓰루 방식을 사용하므로, 특별히 캐시 일치문제를 고려할 필요는 없습니다.

● 라이트 버퍼(Write Buffer)

: 캐시 메모리는 명령어, 데이터를 읽을 경우의 성능 향상을 도모하는 것이 목적인 반면, 쓰는 경우의 성능을 향상 시키고자 하는 것이 **라이트 버퍼**입니다. ARM7 프로세서의 내부에는 통상 32bit X 4의 크기를 갖는 라이트 버퍼를 내장하고 있습니다. CPU가 쓰기 동작을 하는 동안에도 다른 처리를 계속 할 수 있도록, 어드레스와 데이터는 외부 장치가 아닌 라이트 버퍼에 저장됩니다. 이후 CPU는 라이트 버퍼의 쓰기가 끝날 때 까지 기다릴 필요 없이, 계속 명령어를 수행할 수도 있으므로 성능향상을 꾀할 수 있고, 라이트 버퍼의 내용은 버스의 사용권한이 라이트 버퍼에 주어졌을 경우, 라이트 버퍼의 내용이 외부 장치에 쓰여지게 됩니다.



1.5 System On a Chip

● SoC(System On a Chip)

: 디지털용 반도체 소자는 집적도 면에서 계속 발전하여 왔습니다. 집적도에 따른 용어는 아래와 같이 우측으로 갈수록 높은 집적도를 표현하였습니다.

SSI=>MSI=>LSI=>VLSI

그러나 반도체 기술의 빠른 발전에 따라 위의 용어로 표현하기에는 한계에 온 것 같습니다. 집적도 측면에서 요즘은 "SoC"라는 용어를 쉽게 들을 수 있습니다. 중요한 반도체 소자 몇 개를 하나로 집적하는 수준이 아닌, **시스템 (CPU, memory, peripherals) 전체를 하나의 소자로 집적**하고자 하는 것입니다. 이것이 의미하는 바는 단순히 집적도가 높다는 것만은 아닙니다. 시스템의 소형화, 저가격, 고성능은 부가적으로 얻을 수 있습니다. 최근에는 반도체 특성상 차이를 보이던, 아날로그 소자 및 통신소자까지의 하나의 반도체 소자로 집적한 제품이 선을 보이고 있습니다. ARM 프로세서도 "SoC" 소자의 한 예로서, ARM 프로세서, 타이머, DMA, I/O 장치, 인터럽트 컨트롤러 등의 하나의 칩으로 집적되었습니다.

학습정리

● 내장형 프로세서(Embedded Processor)

: 최근 내장형 프로세서는 주로 32비트 급이며, 성능 면에서는 개인용 컴퓨터에 필적하는 성능을 발휘합니다.

● Cache/Write Buffer

: 메모리와 프로세서 사이의 속도차를 해소하고, 버스 사용 효율을 높임으로서 프로세서의 성능을 최대한 발휘하도록 합니다.

● SoC(System On a Chip)

: 최근 반도체 집적도를 반영하는 단어로 기존보다 고집적된 형태로 시스템 전체를 하나의 반도체 칩에 내장하고자 지향하여 고성능, 저전력, 저가격의 부가가치를 창출합니다.

● 하버드 아키텍처

: 폰 노이만 아키텍처와는 달리, 인스트럭션과 데이터 버스가 분리되어 있는 버스시스템을 갖습니다. 따라서 메모리가 이중으로 필요하게 된다는 단점은 있으나 버스의 효율적인 사용이 가능해 동일한 버스 스피드에서 더욱 높은 성능을 발휘할 수 있습니다.

2장 Arm processor Overview



개요

2장에서는 ARM 프로세서의 기본적인 특징과 장점을 살펴봅니다.
또한 다양한 종류의 ARM 프로세서들을 살펴보고, 제품개발을 위한 적절한 선택기준과 앞으로 개발될 개발방향을 살펴봅니다.

학습목표

ARM 프로세서의 기능과 특징을 이해하고, 적절한 소자의 선택능력을 배양합니다

학습내용

1. ARM CPU란 무엇인가?
2. CISC vs RISC
3. 파이프라인이란 무엇인가?
4. Samsung ARM roadmap
5. ARM Selection Guide

2.1 ARM CPU란 무엇인가?

ARM CPU란 무엇인가? 처음 접하시는 분들이 흔히 하시는 질문입니다. 단순히 ARM이라는 회사에서 만드는 프로세서라고 답변하기에는 부족한 부분이 너무 많습니다. 자세한 특징들은 각 장을 통해 살펴보도록 하고, 간단한 질문과 특징들을 통해 ARM 프로세서의 장점을 살펴보도록 하겠습니다.

ARM(Advanced RISC Machines)라는 회사는 ARM 프로세서 IC를 생산하지 않습니다. 이상하다고 생각하시거나 오해하시는 분들이 많습니다. ARM사에서는 설계만 할뿐, 생산을 하지 않습니다. 실질적인 프로세서 IC를 생산하는 업체는 유수의 반도체 업체로, 아래의 그림에서 볼 수 있는 것처럼 대다수의 반도체 업체가 모두 생산하고 있습니다.

각 반도체 업체에서 생산하는 ARM 프로세서는 모두 동일한가?

아닙니다. 여러 반도체 업체에서 동일한 ARM 프로세서를 생산한다면, 경쟁이 굉장히 치열할겁니다. 사실 동일한 프로세서를 그림에서처럼 많은 업체에서 생산할 필요는 없겠지요.

하지만, 조금씩 그 기능이 다른 ARM 프로세서를 많은 반도체 업체에서 생산하고 있습니다.

ARM사에서는 프로세서(혹은 코어)라고 불리우는 핵심부분만을 설계해서 제공합니다.

각 반도체업체에서는 코어에 여러 주변장치(흔히 IP 형태로 존재함)들을 추가해서 IC를 제작, 생산합니다.

이것은 사용용도 및 목적에 따라 다양화될 수 있으며, 집적도가 증가함에 따라 이러한 IC들을 SoC(System On a Chip)이라고 합니다. 1장에서 살펴본 것처럼, SoC는 장점이 참 많습니다.

자, 이제 ARM 프로세서가 무엇을 목표로 설계되었는지 살펴봅시다.

고성능(Very High Performance)

: 기존 임베디드 시장에서는 주로 8/16비트의 프로세서가 주류를 이루고 있었습니다.

하지만 시장에서 요구되는 응용분야가 점차 **음성, 영상, 통신 등의 고성능 처리**가 필요하게 됨에 따라 **32비트 고성능 프로세서**가 등장하게 되었습니다.

기존에 존재하던 32비트 프로세서는 성능은 우수한 반면에 가격, 시스템의 크기, 전력소모 면에서는 8/16비트 프로세서에 비해 불리한 면이 많았습니다. ARM 프로세서는 이러한 단점을 최소화하면서도 고성능의 처리능력을 가지도록 설계되어, 많은 분야에서 그 수요가 급증하고 있습니다.

● 저전력(Very Low Power Consumption)

: 개인용 컴퓨터의 분야에서는 전력소모보다는 처리속도가 더 우선시 되었습니다.

처리속도를 증가시키기 위한 각종 기술이 추가되었으나, 이에 비례하여 전력소모나 발열이 더욱 커지게 되었지요. 하지만 요즘과 같은 **모발(mobil) 시대**에서 요구되는 특징은 처리속도도 빠르면 좋겠지만, 이보다는 **전력소모가 적은 것이 더 우선시**되고 있습니다.

핸드폰, PDA, e-book 등은 모두 배터리로 동작되고 있습니다. 처리속도도 중요하지만, 배터리의 동작시간을 연장시키려면 전력소모를 줄이는 방법이 최선입니다.

ARM 프로세서는 전력소모를 줄이기 위해, 전력소모면에서 최적화된 설계를 지향하고 있습니다. 일부 기능을 보면, 처리속도가 감소되는데도 불구하고 전력소모를 줄이기 위해 선택한 부분도 발견할 수 있습니다.

● 적은 크기의 다이 사이즈(Very Small Die Size)

: IC를 만들기 위해서는 **원형의 실리콘 웨이퍼상에 회로를 구성**합니다.

웨이퍼의 크기가 고정되어 있으므로, IC를 구현하기 위한 크기가 작을수록 많은 IC들을 만들 수가 있습니다. 즉 저가격의 IC를 만들 수 있다는 의미입니다.

IC를 구현하는 부분(die)를 작게 하기 위해서는 당연히 회로가 간단해야 합니다.

이 문제는 처리속도, 저전력, 저가격과 맞물려 있습니다.

IC를 구성하는 회로가 간단할수록, 다이의 크기가 작아집니다.

이에 따라 가격, 전력소모도 비례하여 작아지겠죠? 반면에 처리속도는 감소하게 됩니다.

ARM 프로세서는 최적화된 회로를 통해 저가격, 저전력을 구현하면서도 처리속도는 크게 떨어지지 않도록 설계되었습니다.

● 적은 크기의 코드 사이즈(Very Small Code Size or Good Code Density)

: 동일한 명령을 수행하는데 필요한 코드의 길이가 짧다는 것을 의미합니다.

다르게 표현하자면, 일정한 기능을 수행하는 프로그램을 작성하였을 경우 코드의 길이가 짧다면 전체 프로그램의 크기도 짧다는 의미입니다.

이것은 단순히 짧다는 의미를 넘어 시스템 전체의 저전력, 저가격, 고성능과 맞물려 있는 문제입니다.

ARM 프로세서에서는 ARM 명령어 이외에, **THUMB**이라는 **압축된 형태의 16비트 명령어**를 지원합니다.

THUMB 명령어의 코드길이의 타 프로세서의 동일 명령어에 비해 상당히 짧은 특징을 가지고 있습니다.

📌 ARM 프로세서의 장점은 무엇인가?

이에 대한 대답은 이미 ARM 프로세서의 특징을 통해 찾으실 수 있을 것입니다.

1. 멀티미디어, 통신분야에서 요구되는 **고성능의 구현**
2. 휴대용 제품에서 요구되는 **저전력의 구현**
3. 작은 다이 사이즈를 통한 **저가격의 구현**

이에 덧붙여 유수 반도체업체에서 각기 다른 형태의 ARM 프로세서를 생산하는 것도 큰 장점입니다.

경쟁에 따른 가격의 하락, 호환성에 의한 제품개발의 용이성, 단종에 의한 문제점 해결 등을 들 수 있습니다.

2.2 CISC vs RISC

❏ CISC(Complex Instruction Set Computer)

이것은 **명령어 형태가 말 그대로 복잡하다**는 것을 의미합니다.

특정한 기능을 소프트웨어가 아닌 하드웨어적으로 구현함으로써 **처리속도를 높이려는 것**이 목적이었습니다. 따라서 기능이 추가됨에 따라 하드웨어 구현이 되는 명령어들도 추가되어 복잡하게 되었습니다. 명령어의 복잡성은 IC를 설계하는 회로의 복잡성과 비례하므로, 발열과 더불어 속도증가에 제한을 가지게 되었습니다.

❏ RISC(Reduced Instruction Set Computer)

많은 명령어 중에서 반복적으로 많은 쓰이는 명령어는 하드웨어적으로 구현하고, 다른 명령어들은 하드웨어적으로 구현된 명령어들을 여러 개 사용해서 소프트웨어적으로 구현하고자 만든 형태입니다.

명령어의 수를 줄이므로, 하드웨어가 간단하게 되고 이에 따라 동작속도가 증가되었습니다.

소프트웨어적으로 구현된 명령어에 의한 처리속도의 저하는 동작속도에 의해 보완되고, 전체적으로(확률적으로) 보았을 경우, 처리속도의 향상을 꾀할 수 있다는 것이 목적이었습니다.

명령어 수가 적어진 것과 아울러 명령어의 길이를 동일하게 할 수 있으므로, **파이프라인의 구현이 용이하게** 되었습니다.

ARM 프로세서는 2.1장에서 살펴본 특징들을 위해 RISC 형태로 구현되었습니다.

2.3 Pipeline

❏ 프로세서에서 명령어를 실행하기 위해서는 크게 3단계의 과정을 거칩니다.

1단계: **페치(fetch)단계**에서는 명령어를 읽어 옵니다.

2단계: **디코드(decode) 단계**에서는 읽어 온 명령어를 분석합니다.

3단계: **실행(execute) 단계**에서는 분석한 명령어를 실행합니다.

하나의 명령어를 실행하기 위해서는 이러한 3단계, 혹은 3 클럭이 필요합니다만, 아래의 그림처럼 파이프라인 형태로 3가지 작업이 병행되어 실행이 되면 결국 1개의 명령어가 1클럭에 실행이 되는 것과 같은 효과가 있습니다. 실행속도를 향상시키는 한 가지 방법입니다.



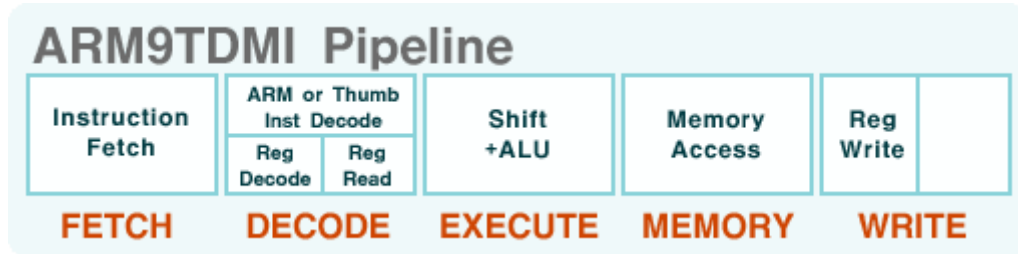
❏ ARM7의 명령어 중 LDR/STR와 같은 명령어는 데이터를 읽거나 쓰는 과정이 있으므로, 3단 파이프라인이 동작이 지연될 수 있습니다.

❏ RISC에서는 한 명령어를 실행하는데 일정한 길이(단계)를 가지기 때문에 위의 파이프라인의 구현이 쉽고, 파이프라인이 동작이 잘 깨어지지 않습니다. CISC의 경우에는 파이프라인의

구현은 가능합니다만, 구현이 복잡할 뿐 아니라 실행되는 명령어의 순서에 따라 파이프라인의 동작이 잘 깨어질 수 있습니다.

ARM9의 경우에는 파이프라인의 단계를 더욱 세분화하여 5단계로 구성되어 있습니다.

Fetch=>Decode=>Execute=>Memory=>Register Write-back



ARM(Advanced RISC Machines)

: ARM 프로세서는 다양한 형태로 수많은 반도체 업체에서 생산되는 프로세서입니다. 하지만, 코어는 동일하므로, 통일된 개발환경과 개발툴들을 사용하는 편의성이 있습니다.

RISC(Reduced Instruction Set Computer)

: 최근 전자제품들을 휴대성과 멀티미디어 성능이 갈수록 증대되어 갑니다. RISC 타입의 프로세서는 저전력 동작특성을 가지면서도 높은 성능을 얻을 수 있어, 이러한 요구에 부합될 수 있으며 각광받는 이유입니다.

Pipeline

: 명령어 처리속도를 높이기 위하여, 명령어 처리단계를 세부화한 뒤 이를 병렬로 동시에 수행이 되도록 만든 구조입니다. 따라서 파이프라인의 단계가 많을수록 처리속도는 더욱 빨라진다고 생각할 수 있습니다.

ARM7은 3단 파이프라인을 ARM9의 경우에는 5단 파이프라인을 사용합니다.

3장 Programming Model



개요

3장에서는 ARM 프로세서가 가지고 있는 약간은 특별하다고 할 수 있는 내용들에 대해서 학습합니다. 특별히 **Exception**과 **Operating Mode**에 대해서 자세히 살펴보도록 합시다.

학습목표

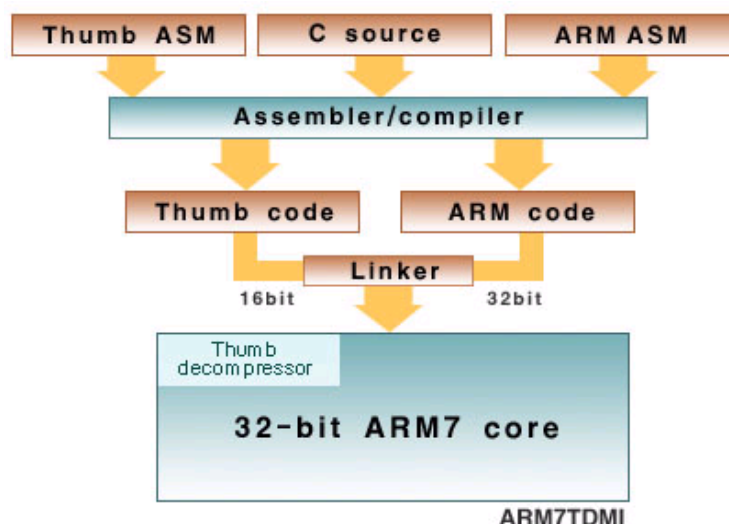
1. Exception의 이해
2. Operating Mode의 이해

학습내용

1. ARM/THUMB Mode
2. Little/Big Endian
3. Registers, CPSR,SPSR
4. Exception
5. Operating Mode

3.1 ARM & THUMB Mode

ARM 프로세서는 16/32bit 프로세서라고들 합니다. 동일한 프로세서에서 16/32bit 명령어를 수행할 수 있기 때문입니다. 32bit로 구성된 명령어를 (1)ARM 명령어, 16bit로 구성된 명령어를 (2)THUMB 명령어라고 합니다. ARM 명령어가 수행되는 모드를 ARM 모드, THUMB 명령어가 수행되는 모드를 THUMB 모드라고 합니다.



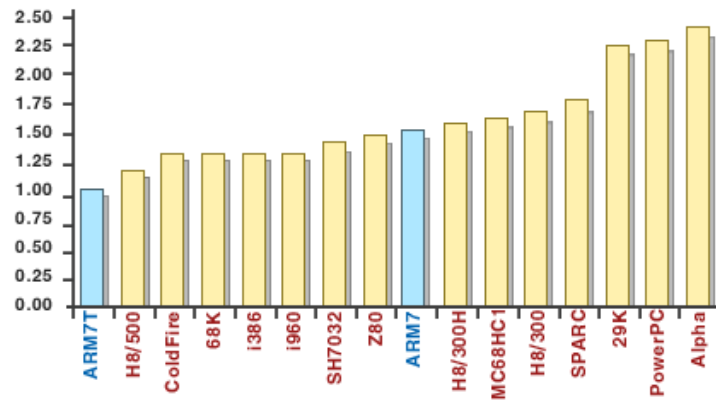
위의 그림에서 보는 것처럼, 프로그램 작성시 코딩은 ARM/THUMB 명령어를 혼용해서 작성할 수 있습니다.

각각의 컴파일 과정을 거친 후, 해당 코드를 생성합니다. ARM 프로세서에서 명령어가 실행이 될 때에는 ARM 명령어인 경우는 바로 실행이 이루어지고, THUMB 명령어인 경우에는 THUMB

Decompressor를 거쳐 ARM 명령어로 확장이 된 후 실행이 됩니다. 이 과정은 하드웨어적으로 수행이 되기 때문에 실행시 추가적인 성능 지연이 일어나지는 않습니다.

● THUMB 명령어는 16bit 명령어로 압축이 되어있는 형태라고 설명 드렸습니다. 압축이 많이 될수록 명령어의 길이가 상대적으로 짧다고 할 수 있는데, 이로 인해 부수적으로 얻을 수 있는 장점이 있습니다.

아래의 그림은 ARM7의 THUMB의 명령어의 길이를 1로 했을 때, 타사 프로세서의 명령어 길이와 비교한 그림입니다. 상대적으로 가장 짧다고 혹은 압축이 많이 되었다고 생각할 수 있겠습니다.



3.2 Little/Big Endian

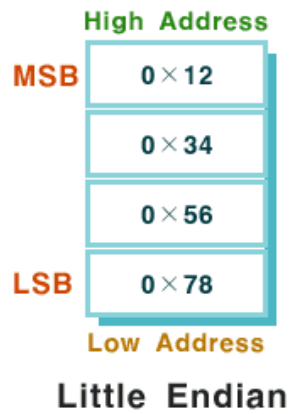
● Little/Big Endian은 8bit 이상의 데이터에서 바이트 단위의 정렬방식에 따른 차이점을 가지고 있습니다. PC의 경우에는 Little Endian방식을 따릅니다. ARM7을 생산하는 타사의 경우, 대부분이 Little Endian만을 지원하는 반면, 삼성 반도체의 경우에는 초기에는 Big Endian 만을 지원했었고 최근에는 Little/Big Endian을 동일 칩에서 모두 지원하도록 되어있습니다.



● Little Endian

Little Endian은 상위 어드레스에 상위 데이터, 하위 어드레스에 하위 데이터가 저장되는 형태입니다.

예를 들어 32bit 데이터 "0x12345678"이 위와 같이 있다고 합시다. 상위 8bit 데이터가 "0x12", 하위 데이터가 "0x78"입니다. 이 데이터를 메모리 공간상에 표현하면, 아래의 그림과 같습니다. 즉 상위 어드레스에 상위 데이터 "0x12", 하위 어드레스에 하위 데이터 "0x78"이 저장됩니다.



● Big Endian

Big Endian은 Little Endian과 반대로 상위 어드레스에 하위 데이터, 하위 어드레스에 상위 데이터가 저장되는 형태입니다.

Little Endian과 동일한 예를 들어 설명합니다. 즉 상위 어드레스에 하위 데이터 "0x78", 하위 어드레스에 상위 데이터 "0x12"가 저장됩니다.



3.3 Registers, CPSR/SPSR Register

● ARM&THUMB 모드시 레지스터 관계

(1) ARM Mode

(2) THUMB Mode

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15
CPSR
SPSR

ARM Mode

R0
R1
R2
R3
R4
R5
R6
R7
R13
R14
R15
CPSR
SPSR

THUMB Mode

● 동작모드별 레지스터

(3) User&System Mode

(4) FIQ Mode

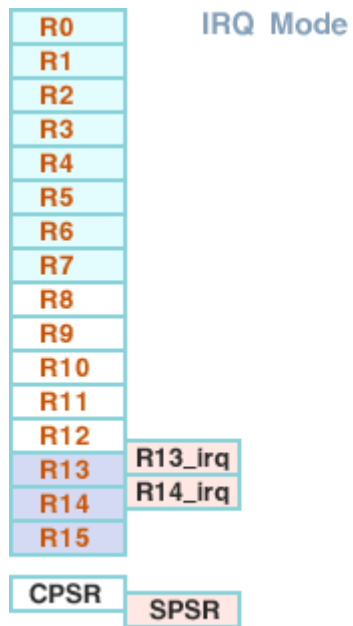
R0
R1
R2
R3
R4
R5
R6
R7
R8_fiq
R9_fiq
R10_fiq
R11_fiq
R12_fiq
R13_fiq
R14_fiq
R15
CPSR
SPSR

FIQ Mode

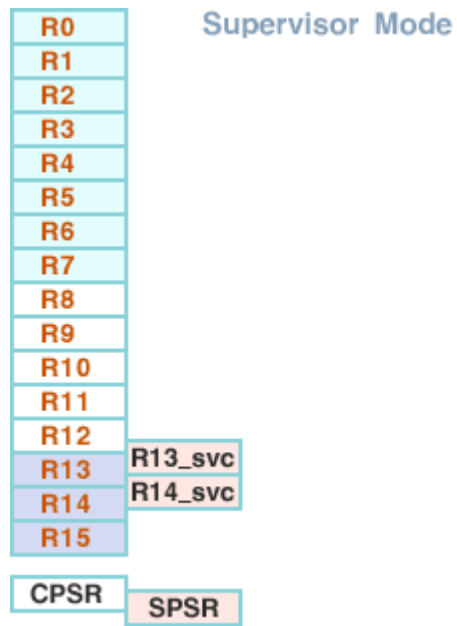
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15
CPSR

User&System Mode

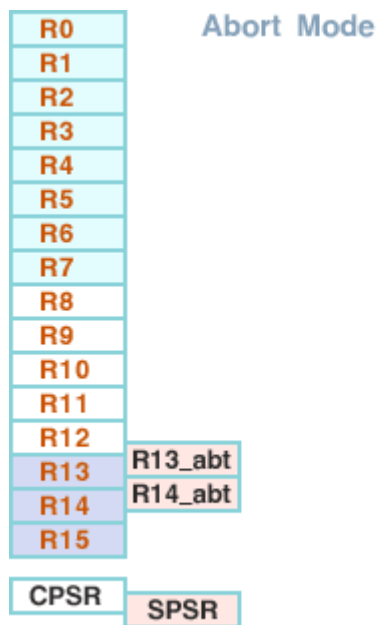
(5) IRQ Mode



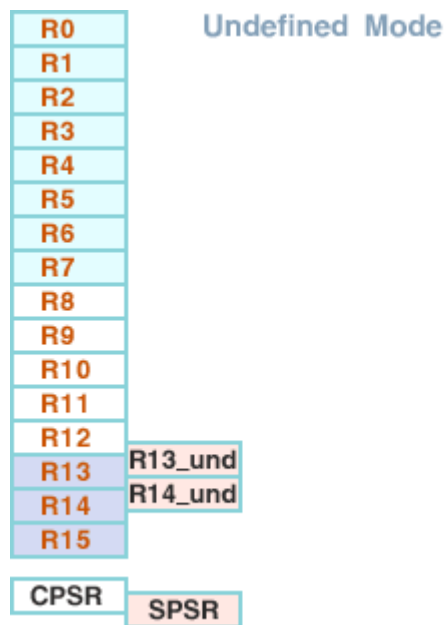
(6) Supervisor Mode



(7) Abort Mode



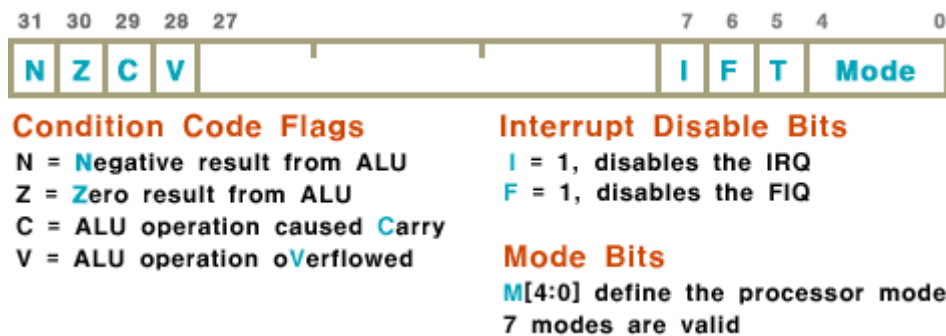
(8) Undefined Mode



● 기본 레지스터

- ARM은 **RISC 프로세서**입니다. 대부분의 RISC 프로세서는 연산에 사용될 데이터를 레지스터에 저장한 후, 레지스터 기반으로 연산을 수행합니다.
- ARM은 일반 목적의 레지스터 16개(R0-15)와 **CPSR(Current Program Status Register)**, **SPSR(Saved Program Status Register)**을 가지고 있습니다.
- R13은 SP(Stack Pointer)로 사용됩니다. 스택을 사용하지 않을 경우, 일반 레지스터로 사용 가능합니다.
- R14는 LR(Link Register)로 사용됩니다. LR 레지스터는 PC값을 임시로 저장하는데 사용하며, 명령어 중 "BL"의 실행시 사용됩니다. 사용하지 않을 경우, 일반 레지스터로 사용 가능합니다.
- R15는 PC(Program Counter)입니다. PC는 다음에 수행이 되어야 할 명령어의 주소를 가르킵니다.
프로그램 작성시 R15 혹은 PC로 혼용해서 사용하여도 아무런 지장없이 컴파일됩니다.
- 앞서 ARM/THUMB Mode를 살펴보았죠?
ARM Mode시에는 SP/LR/PC를 제외하고, R0-R12를 일반 목적의 레지스터로 사용 가능합니다. THUMB Mode의 경우는 ARM Mode와는 달리 R0-R7까지만 사용이 가능합니다. 레지스터 5개를 덜 쓰게 되는 셈이죠

● CPSR(Current Program Status Register)



● Condition Code Flags

CPSR 레지스터는 위의 그림에서 보는 바와 같이 32bit 레지스터로, 상태 플래그와 인터럽트 관련기능, 프로세서 모드에 관련된 정보들을 포함하고 있습니다.

최상위 비트에 위치하고 있는 N,Z,C,V 4개의 비트는 연산결과에 따른 상태 플래그들입니다.

- N : 연산결과가 음수인 경우, 이 비트가 1로 설정됩니다.
- Z : 연산결과가 0인 경우, 1로 설정됩니다.
- C : 캐리가 발생한 경우, 1로 설정됩니다.

유의할 점은 상태 플래그에 Borrow 비트가 없다는 점입니다. Borrow 상태를 위해

캐리 비트를 사용합니다. 의미는 반대입니다. 즉 Borrow가 발생할 경우, C는 0으로 설정됩니다
V : 오버플로우가 발생한 경우, 1로 설정됩니다.

3.4 Exception

● Interrupt Disable Bits

ARM 프로세서의 경우, 인터럽트와 관련된 익셉션(Exception)은 IRQ와 FIQ 두가지가 있습니다.
자세한 내용은 3.4장을 참조하시기 바랍니다.

IRQ/FIQ Exception을 동작하지 않도록 하기 위한 비트들입니다.

I : IRQ Exception을 동작하지 않도록 합니다.

F : FIQ Exception을 동작하지 않도록 합니다.

● Mode Bits

ARM 프로세서에는 7가지 동작 모드(Operating Mode)가 있습니다.

자세한 내용은 3.5장을 참조하시기 바랍니다.

하위 5bit(0-4)는 현재 동작 모드를 나타냅니다.

현재 동작모드가 어떤 모드인지 알고 싶을 때, 하위 5bit를 읽어 확인할 수 있습니다.

동작모드를 변경하고자 할 때, 하위 5bit 값을 변경하면 모드가 변경됩니다.

단, 현재 USER 모드일 때에는 M[4:0] 비트값을 변경할 수 없습니다.

동작모드별 비트값은 아래와 같습니다.

10000	USER
10001	FIQ
10010	IRQ
10011	SUPERVISOR
10111	ABORT
11011	UNDEFINED
11111	SYSTEM

▶▶ 심화학습

User 모드에서는 모드비트를 이용해서 다른 모드로 변경이 불가능합니다.

다른 모드로의 전환을 위해서는 어떻게 해야 할까요?



Tip

User 모드에서 다른 모드로의 전환을 원하는 경우, SWI 명령어를 사용합니다.

SWI 명령어를 사용하면, SWI exception이 발생하며 supervisor 모드로 전환됩니다.

모드 전환 후 적절한 처리를 통해 다른 모드에의 전환을 모색합니다

● Exception

ARM 프로세서에는 총 7개의 Exception이 존재합니다. 일반 프로세서에는 보지 못하는 기능입니다.

인터럽트보다 큰 개념이라고 생각하시면 될 것 같습니다.

아래의 테이블은 우선순위(priority)를 기준으로 정렬한 것입니다.

(1) Priority / Vector Address

Exception	Priority	Mode	Vector
Reset	1	Supervisor	0x00000000
Data Abort	2	Abort	0x00000010
FIQ	3	FIQ	0x0000001c
IRQ	4	IRQ	0x00000018
Prefetch Abort	5	Abort	0x0000000c
Undefined Instruction	6	Undef	0x00000004
SWI	7	Supervisor	0x00000008

- Exception 발생시, 해당 모드로 변경과 아울러 벡터주소 (Vector Address)로 분기합니다.

이와 병행하여 자동적으로 수행되는 동작은 오른쪽과 같습니다.

LR(R14) := PC
SPSR := CPSR
CPSR update

- Exception과 관련된 동작이 모두 끝난 후, 리턴될 경우에 수행되는 동작은 오른쪽과 같습니다.

CPSR := SPSR
PC := LR(R14)

- Exception에 따라 세부적인 내용이 차이가 있습니다. 하나씩 구분하여 살펴봅니다.
- Exception을 Vector Address에 따라 재정렬한 그림은 아래와 같습니다. 최초 리셋 시 Reset Exception이 수행되어야 하므로 0x00번지에 위치하고 있으며, FIQ Exception의 Service Routine은 0x1c번지 바로 아래에 위치할 수 있도록 가장 아래부분에 위치하고 있습니다. 이것은 점프로 인한 지연까지도 제거하기 위한 고려라고 이해하시면 되겠습니다. 0x14번지는 미래를 위해서 현재 사용되지 않는 예약되어진 번지입니다.

0x00	Branch to Handler	Reset
0x04	Branch to Handler	Undefined Instruction
0x08	Branch to Handler	Software Interrupt
0x0c	Branch to Handler	Prefetch Memory Abort
0x10	Branch to Handler	Data Memory Abort
0x14	Branch to Handler	Reserved
0x18	Branch to Handler	Normal Interrupt
0x1c	Branch to Handler	Fast Interrupt

Interrupt Handler

심화학습

FIQ가 가장 긴급히 처리되어야 하는데도 불구하고, 왜 data abort가 FIQ보다 우선순위가 높은가요?



Tip

Case 1)

Data abort와 FIQ가 동시에 발생한 경우, 만약 FIQ가 먼저 처리된다면:

FIQ 처리후 data abort가 발생한 명령 다음 명령어를 수행하기 때문에 data abort가 발생했었다는 것을 잊게 됩니다.

Case 2)

Data abort와 FIQ가 동시에 발생한 경우, 만약 data abort가 먼저 처리된다면:

일단 data abort 처리 루틴(0x10)으로 branch한 후 data abort에서는 FIQ가 금지 되지 않았으므로, FIQ 처리루틴이 다시 바로 수행됩니다.
 그후 FIQ처리를 끝낸후 다시 data abort처리루틴으로 복귀하므로 data abort가 정상적으로 처리가 이루어 집니다.

● Reset Exception


전원이 최초 공급이 된 상황, 혹은 하드웨어적으로 리셋이 걸리는 경우 Reset Exception이 발생합니다. 이때, 특정 레지스터의 내용은 아래 그림과 같이 설정됩니다.

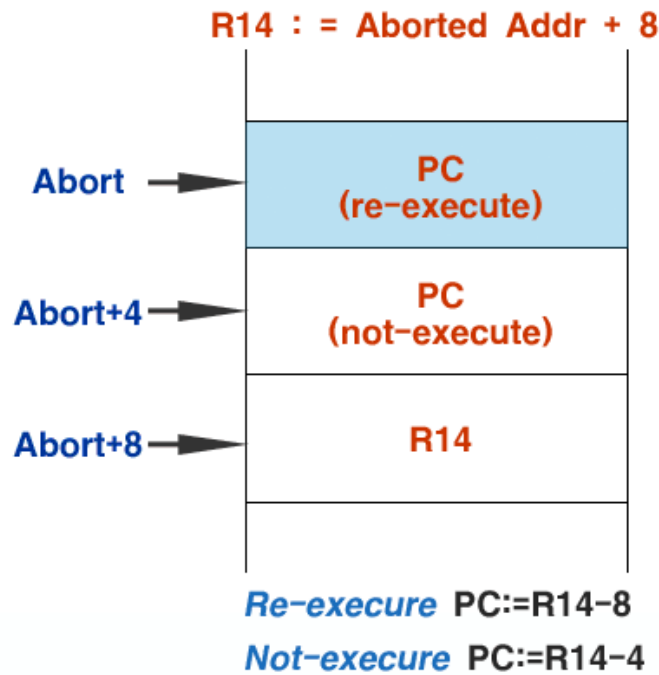
- R14_svc : Supervisor모드에서의 레지스터 R14를 의미합니다. Reset Exception시 예측할 수 없는 더미값이 저장되게 됩니다.
- SPSR_svc : Supervisor모드에서의 레지스터 SPSR를 의미합니다. CPSR값과 동일한 값이 저장됩니다.
- CPSR[4:0] : 프로세서 모드는 Supervisor Mode로 설정됩니다.
- CPSR[5] : T=0, ARM 모드로 설정됩니다.
- CPSR[7:6] : IRQ/FIQ 인터럽트 모두 사용이 금지됩니다.
- PC : ARM 프로세서는 0x00번지로 점프합니다.

R14_svc	unpredictable value
SPSR_svc	CPSR
CPSR[5:0]	Supervisor mode
CPSR[7:6]	I = 1 , F = 1
PC	0 x 0

● Data Abort Exception

버스를 통해 명령어가 아닌 데이터를 읽으려고 합니다. 예를 들면, LDR/STR 명령어가 실행되는 경우를 들 수 있습니다. 이 동작이 실패하였을 경우, Data Abort Exception이 발생합니다.

- R14_a bt : 데이터 LDR/STR 동작이 실패한 주소 + 8
- SPSR_a bt : Exception이 일어나기 전의 CPSR값이 저장됩니다.
- CPSR[4:0] : Abort 모드로 전환됩니다.
- CPSR[5] : T=0, ARM 모드로 설정됩니다.
- CPSR[7:6] : IRQ Disable, FIQ는 변경되지 않음
- PC : 0x10번지로 점프
- Return Sequence ➡  **Tip**



Data Abort 동작에서 문제가 발생하였습니다. R14(LR) 레지스터에는 데이터를 읽는 명령어의 주소+8의 값을 저장하였습니다. 따라서 데이터를 다시 읽고자 할 경우에는 (R14-8)의 주소로 분기해야하며, 이를 포기하고 다음 명령어를 계속 진행하고자 할 경우에는 (R14-4)의 주소로 분기합니다.

R14_abt	address of an aborted instruction+8
SPSR_abt	CPSR
CPSR[5:0]	Abort mode
CPSR[7:6]	I = 1 , F = unchanged
PC	0 x 10
To return	SUBS PC, R14, #4 (not re-executed) SUBSS PC, R14, #8 (re-executed)

SUB 명령어는 산술연산 명령어로 뺄셈 기능을 합니다.

SUBS에서 S는 CPSR의 상태 플래그에 영향을 주도록 하는 접미사입니다.

SUBS PC, ... 와 같이 프로그램을 작성할 경우, 즉 명령어는 SUBS이고 결과가 저장되는 레지스터가 PC(R15)인 경우 SPSR값이 CPSR로 복사됩니다.

이때 CPSR[5:0]이 변경되어 원래 프로세서 모드로 되돌아오게 됩니다.


FIQ Exception

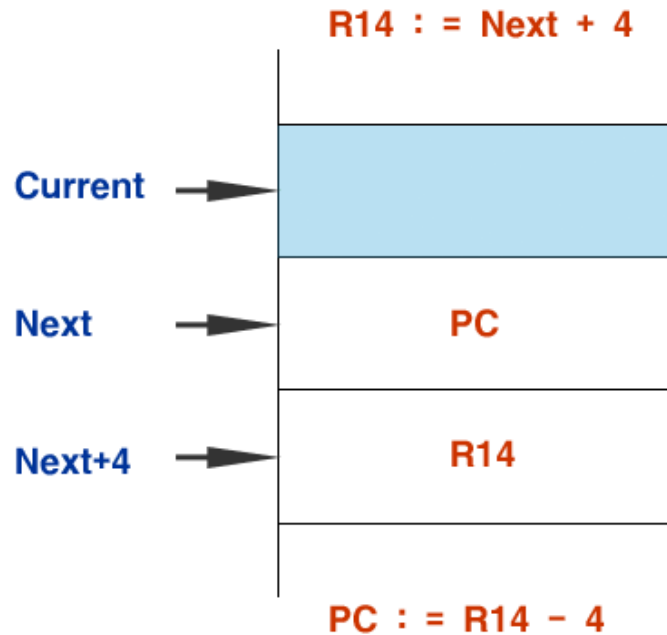
매우 긴급하게 처리해야 할 인터럽트의 처리를 위해 만들어졌습니다.

OS에서 Context Switch시 오버헤드를 줄이기 위해 전용 레지스터들을 가지고 있습니다.

- R14_fiq : 다음 수행할 명령어 번지+4
- SPSR_fiq : Exception이 일어나기 전의 CPSR값이 저장됩니다.
- CPSR[4:0] : FIQ 모드로 전환됩니다.

- CPSR[5] : T=0, ARM 모드로 설정됩니다.
- CPSR[7:6] : IRQ Disable, FIQ Disable
- PC : 0x1C번지로 점프

● Return Sequence   **Tip**




R14_fiq	address of a next instruction +4
SPSR_fiq	CPSR
CPSR[5:0]	FIQ mode
CPSR[7:6]	I = 1 , F = 1
PC	0 x 1c
To return	SUBS PC, R14, #4


SUB 명령어는 산술연산 명령어로 뺄셈 기능을 합니다. SUBS에서 S는 CPSR의 상태 플래그에 영향을 주도록 하는 점미사입니다. 4.1.1장을 참조하세요.SUBS PC, ... 와 같이 프로그램을 작성할 경우, 즉 명령어는 SUBS이고 결과가 저장되는 레지스터가 PC(R15)인 경우 SPSR값이 CPSR로 복사됩니다. 이때 CPSR[5:0]이 변경되어 원래 프로세서 모드로 되돌아오게 됩니다.

● IRQ Exception

FIQ가 아닌 일반 인터럽트를 처리할 때 사용됩니다.

 심화학습



FIQ와 IRQ Exception이 동시에 발생하였을 때, 동작을 설명하시오

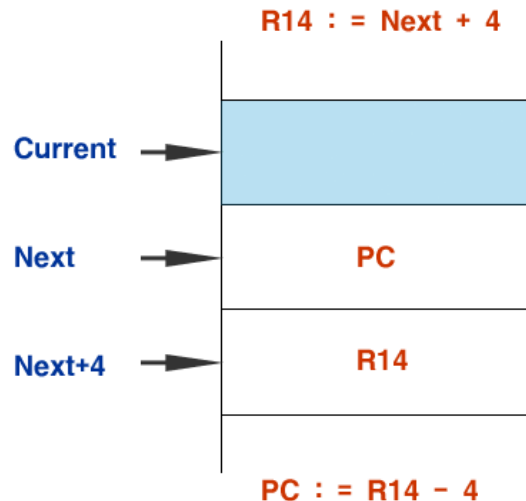
 **Tip**

▶ FIQ 보다 IRQ의 우선순위가 낮기 때문에 동시에 발생할 경우,
FIQ가 먼저 처리된 후, 원래 동작모드로 돌아온다. 다음 IRQ가 처리됩니다.
만일 FIQ 처리루틴에서 소프트웨어적으로 IRQ Enable시키면,
원래모드=>FIQ=>IRQ순으로 Exception이 발생합니다.
단, 이럴 경우에는 원래 동작상태로 돌아오는 것이 불가능하게 됩니다.

이를 방지하기 위해서 FIQ Exception시 FIQ/IRQ를 모두 disable시킵니다.

- R14_irq : 다음 수행할 명령어 번지+ 4
- SPSR_irq : Exception이 일어나기 전의 CPSR값이 저장됩니다.
- CPSR[4:0] : IRQ 모드로 전환됩니다.
- CPSR[5] : T=0, ARM 모드로 설정됩니다.
- CPSR[7:6] : IRQ Disable, FIQ 이전상태값
- PC : 0x18번지로 점프

● Return Sequence   **Tip**




R14_irq	address of a next instruction +4
SPSR_irq	CPSR
CPSR[5:0]	IRQ mode
CPSR[7:6]	I = 1 , F = unchanged
PC	0 x 18
To return	SUBS PC, R14, #4

● Prefetch Abort Exception


CPU는 명령어를 읽으려고(prefetch) 하는데, 시스템이 메모리에서 명령어 코드를 읽어 올 수 없다고 하는 경우에 발생하는 exception입니다.

명령어를 읽었으나, 수행하지 못하는 명령어인 경우는 Undefined Instruction입니다. 착오없으시기 바랍니다.

 심화학습

ARM7은 3-stage pipeline, ARM9은 5-stage pipeline을 사용합니다.

Prefetch Abort Exception은 현재 파이프라인 싸이클이 종료된 후 발생합니다.

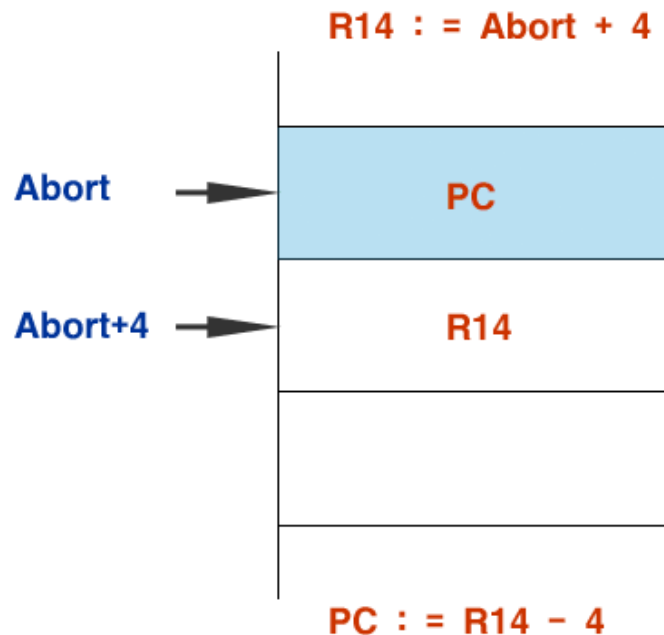
 **Tip**



위의 그림은 3단 구조를 가지고 있는 ARM7의 파이프라인입니다. 첫 번째 명령어가 페치되고 디코드될 때, 두 번째 명령어가 페치되고 있습니다. (적색으로 된 페치부분)

이때 문제가 발생한다고 가정하면, prefetch abort exception이 발생합니다. 즉각 exception을 수행하면, 첫 번째 명령어가 실행(execute)되지 않은 상황이므로 문제가 발생할 수 있습니다. 따라서 이러한 문제가 발생하지 않도록 첫 번째 명령어가 실행이 모두 된 후 exception이 발생합니다. 또한 두 번째 명령어와 세 번째 명령어는 위 그림의 파이프라인의 형태와 같이 수행되지 않고 뒤로 밀리게 됩니다. 즉, 파이프라인이 깨지는 상황입니다.

- R14_abt : Prefetch가 실패한 명령어 번지+4
- SPSR_abt : Exception이 일어나기 전의 CPSR값이 저장됩니다.
- CPSR[4:0] : Abort 모드로 전환됩니다.
- CPSR[5] : T=0, ARM 모드로 설정됩니다.
- CPSR[7:6] : IRQ Disable, FIQ는 이전값
- PC : 0x0C번지로 점프
- Return Sequence **Tip**



Abort 위치에서 명령어를 실행하지 못했으므로, Abort 위치에서부터 다시 명령어를 실행해야 합니다.

따라서 R14에 저장되어있는 주소에서 4를 감소한 위치, 즉 (R14-4)를 PC에 저장합니다.

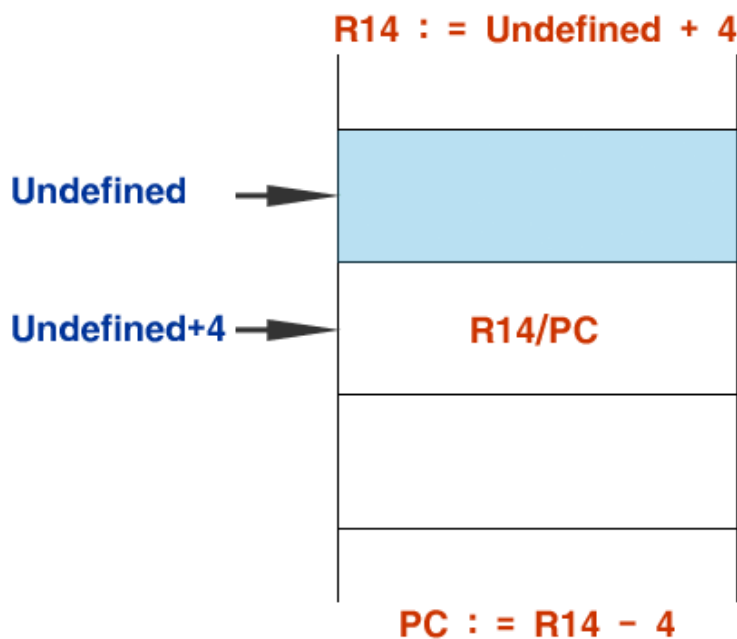
R14_abt	address of a aborted instruction +4
SPSR_abt	CPSR
CPSR[5:0]	Abort mode
CPSR[7:6]	I = 1 , F = unchanged
PC	0 x c
To return	SUBS PC, R14, #4

● Undefined Instruction Exception

명령어를 읽었으나, ARM이 수행할 수 없는 명령어인 경우 발생하는 exception입니다.

- R14_und : Undefined 명령어 번지+ 4
- SPSR_und : Exception이 일어나기 전의 CPSR값이 저장됩니다.
- CPSR[4:0] : Undefined 모드로 전환됩니다.
- CPSR[5] : T=0, ARM 모드로 설정됩니다.
- CPSR[7:6] : IRQ Disable, FIQ는 이전값
- PC : 0x04번지로 점프

● Return Sequence ➡ Tip



R14_und	address of undefined instruction +4
SPSR_und	CPSR
CPSR[5:0]	Undefined mode
CPSR[7:6]	I = 1 , F = unchanged
PC	0 x 4
To return	MOVS PC, R14

▶ 심화학습

지금까지는 SUBS 명령어를 사용했는데, 왜 MOVS 명령어를 사용할까요?



Tip

대체적으로 undefined instruction은 현재 ARM7TDMI에서 지원되지 않는 명령어를 S/W적으로 emulation하기 위해 사용되는데, 이경우 S/W에 의해 undefined instruction이 이미 처리되었으므로, undefined instruction이 재수행될 필요가 없습니다. 혹은 지원되지 않는 명령어이므로, 실행하지 않고 지나쳐야겠지요? 따라서 R14에 저장되어 있는 주소에서부터 명령어가 실행되어야 하므로, SUBS 명령어를 사용할 필요가 없습니다. 이러한 이유로 MOVS 명령어를 사용하는 것이죠.

만일 SUBS 명령어를 굳이 사용하고자 한다면, SUBS PC, R14, #0 라고 사용해도 되겠지요

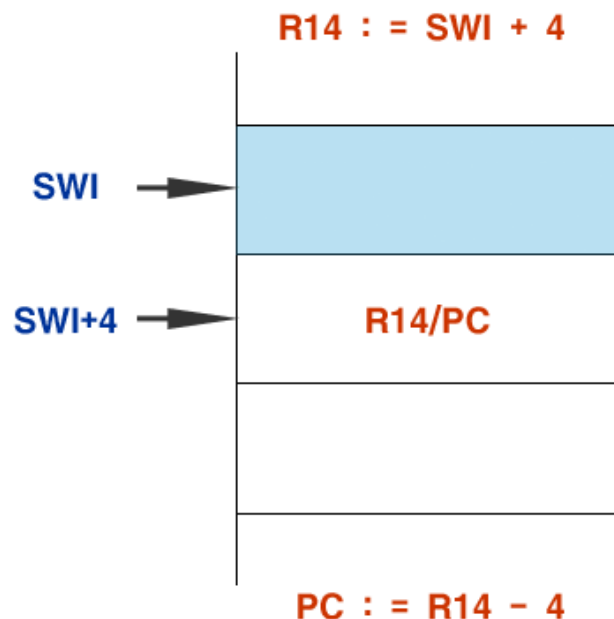
● SWI(Software Interrupt) Exception

OS상의 여러 프로그램들이 공유해서 사용하게 되는 루틴을 소프트웨어 인터럽트를 이용하여 주로 사용합니다. 이를 위해 지원되는 것입니다.

OS가 통상 supervisor 모드에서 동작하므로, SWI 발생시 이 모드로 전환됩니다.

- R14_svc : SWI 명령어 번지 + 4
- SPSR_svc : Exception이 일어나기 전의 CPSR값이 저장됩니다.
- CPSR[4:0] : Supervisor 모드로 전환됩니다.
- CPSR[5] : T=0, ARM 모드로 설정됩니다.
- CPSR[7:6] : IRQ Disable, FIQ는 이전값
- PC : 0x08번지로 점프

● Return Sequence ▶▶ Tip



SWI 명령어 주소에 있는 명령어는 이미 실행이 되었으므로, SWI+4 주소가 저장된 R14가 PC로 전송되면 됩니다.

R14_svc	address of SWI instruction +4
SPSR_svc	CPSR
CPSR[5:0]	Supervisor mode
CPSR[7:6]	I = 1 , F = unchanged
PC	0 x 8
To return	MOVS PC, R14

3.5 Operation Mode

ARM에는 총 7개의 동작 모드가 있습니다. 이렇게 여러 개의 모드를 준비한 것은 멀티태스킹 운영체제를 위한 것입니다. Exception과 같이 7개일 뿐 아니라, 동작 시 연관된 부분이 있어 오해하기 쉬운 부분입니다. Exception과 Operating Mode를 잘 구분하여 이해하도록 합시다.

1. User : 일반적인 프로그램이 동작하는 모드입니다.
2. System : 주로 OS kernel에서 사용됩니다. 단, User와 System 모드는 동일한 레지스터 뱅크를 사용하므로, 스택을 이용하여야 원활한 동작이 이루어집니다.
3. FIQ : 인터럽트 중에서 IRQ보다 더 빨리 수행이 되어야 하는 부분을 FIQ로 설정하면 됩니다. 가장 우선순위가 높은 인터럽트로 이해하시면 되겠습니다. 단지 차이점이라면 빠른 처리를 위해 하드웨어적인 지원이 된다고 보시면 되겠지요.
4. IRQ : FIQ보다 우선순위가 낮은 인터럽트입니다. 일반 프로세서에서의 인터럽트와 동일하다고 보시면 됩니다. 소프트웨어로 여러 개의 IRQ를 처리할 수 있습니다. 자세한 내용은 7장의 Isr_Irq 부분을 참조하시기 바랍니다.
5. Supervisor: OS를 위한 모드입니다. SWI exception 발생시 진입합니다. 최초 RESET exception 시에도 이 모드로 진입합니다.
6. Abort : 데이터를 읽거나 쓰는 동작시, 메모리 시스템이 읽거나 쓸 수 없다고 하는 경우의 모드입니다.
7. Undef : 명령어를 페치했으나, ARM7TDMI가 모르는 명령어인 경우의 모드입니다.

4장 Instructions



개요

4장에서는 32bit ARM 명령어들을 공부하고 동작과정을 이해 할 수 있습니다.
또한 학습한 명령어들을 이용하여 프로그램 작성 시 활용능력을 배양할 수 있도록 예제를 중심으로 학습합니다.

학습목표

1. ARM 명령어의 이해
2. 명령어 활용능력 배양

학습내용

1. ARM&THUMB Instruction
2. Data Processing Instruction
3. Data Transfer Instruction
4. Flow Control Instruction
5. 기타 명령어

4.1 ARM & THUMB Instruction

ARM 프로세서는 16/32bit 프로세서라고들 합니다. 동일한 프로세서에서 16/32bit 명령어를 수행할 수 있기 때문입니다. 32bit로 구성된 명령어를 ARM 명령어, 16bit로 구성된 명령어를 THUMB 명령어라고 합니다.

ARM 명령어를 사용하는 경우는 다음과 같습니다.

프로세서의 **처리속도가 우선시** 되어야 하는 프로그램 작성 시에는 THUMB 명령어보다는 ARM 명령어의 사용이 유리합니다. 프로세서에 인터페이스되어있는 메모리가 32bit 버스로 구성되어있는 경우에는 굳이 16bit인 THUMB 명령어를 사용할 해도 특별한 장점을 얻지 못합니다. 즉, 32bit를 한번 읽어서 16bit THUMB 명령어 2개를 실행시키지는 않습니다.

THUMB 명령어를 사용하는 경우는 다음과 같습니다.

전력소모를 줄여야 하는 휴대용 시스템의 개발을 위해서, 메모리를 16bit로 구성해야하는 경우, 이때는 ARM 명령어보다 THUMB 명령어를 사용하는 것이 통상 30%정도 처리속도 향상이 되므로 유리합니다.

단, 캐시가 없는 시스템이거나 캐시 실패(cache miss)한 경우에 한합니다.

캐쉬가 있는 시스템(cache hit의 경우)에서 메모리가 16bit이더라도 ARM 명령어로 작성하는것이 THUMB 명령어보다 여전히 빠릅니다.

프로그램이 저장되는 프로그램용 메모리의 크기를 줄여야 하는 경우, 이때에도 THUMB 명령어를 사용하는 것이 유리합니다. 통상 ARM 명령어로 작성된 프로그램을 THUMB 명령어로 작성하면 70%정도로 크기가 줄어듭니다.


4.2 Data Processing Instruction

● 기본적으로 설명은 ARM 명령어를 사용합니다.

따라서 명령어는 32bit의 길이를 가집니다.

● 일반적으로 명령어에는 2개의 입력 데이터와 1개의 결과 데이터로 구성됩니다.

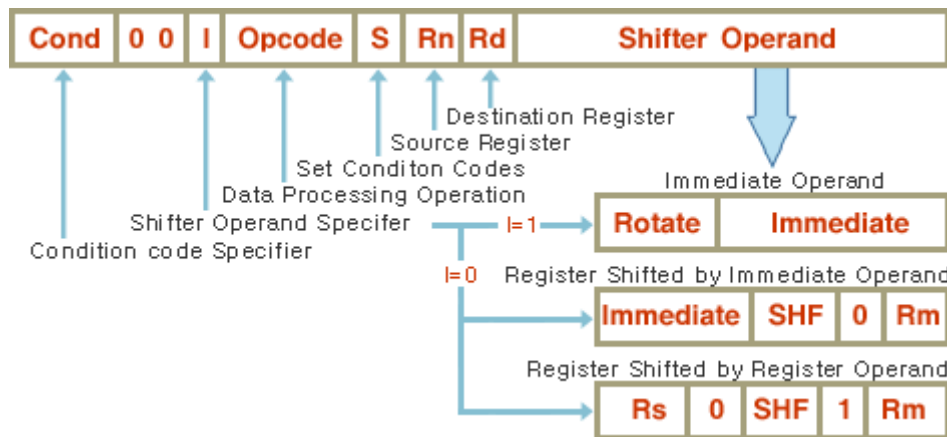
● Data Processing Instruction은 데이터의 처리에 관련된 명령어들로 산술, 논리, 비교

등에 사용됩니다.  **Tip**

▶ OP-CODE별 Data Processing Instruction

Mnemonic	Operation	Opcode
MOV	Move	1101
MVN	Move Not	1111
CMP	Compare	1010
CMN	Compare Negated	1011
TST	Test	1000
TEQ	Test Equivalence	1001
ADD	Add	0100
ADC	Add with Carry	0101
SUB	Subtract	0010
SBC	Subtract with Carry	0110
RSB	Reverse Subtract	0011
RSC	Reverse Subtract with Carry	0111
AND	Logical AND	0000
EOR	Logical Exclusive OR	0001
ORR	Logical (inclusive) OR	1100
BIC	Bit Clear	1110

Mnemonic	Action
MOV	Rd := operand2 (operand1 is ignored)
MVN	Rd := NOT operand2 (operand1 is ignored)
CMP	update flags after operand1 - operand2
CMN	update flags after operand1 + operand2
TST	update flags after operand1 AND operand2
TEQ	update flags after operand1 EOR operand2
ADD	Rd := operand1 + operand2
ADC	Rd := operand1 + operand2 + Carry Flag
SUB	Rd := operand1 - operand2
SBC	Rd := operand1 - operand2 - NOT(Carry Flag)
RSB	Rd := operand2 - operand1
RSC	Rd := operand2 - operand1 - NOT(Carry Flag)
AND	Rd := operand1 AND operand2
EOR	Rd := operand1 EOR operand2
ORR	Rd := operand1 OR operand2
BIC	Rd := operand1 AND NOT operand2



4.2.1 Arithmetic Operation

● 덧셈 명령어

- ADD reg1, reg2, reg3 => reg1:=reg2+reg3 :

reg2와 reg3를 더한 후, 그 결과를 reg1에 저장합니다. 캐리비트는 사용하지 않습니다.

Tip

<Syntax>

ADD<cond>S <Rd>, <Rn>, <shifter_operand>

<Operation>

if ConditionPassed(cond) then

Rd = Rn + shifter_operand

if S == 1 and Rd == R15 then

CPSR = SPSR

else if S == 1 then

N Flag = Rd[31]

Z Flag = if Rd == 0 then 1 else 0

C Flag = CarryFrom(Rn + shifter_operand)

V Flag = OverflowFrom(Rn + shifter_operand)

<Description>

덧셈 명령어인 "ADD" 명령어로서 32비트 덧셈 기능을 한다. CPSR 레지스터의 상태 플래그(N,Z,C,V)에 따른 조건연산을 하고자 할 때에는, ADD명령어 뒤에 조건연산자를 추가한다.

예) ADDNE, ADDEQ, ADDGT 등

"ADD" 명령어는 캐리를 고려하지 않고, 단순히 32비트 덧셈기능만을 수행한다.

캐리를 고려할 경우에는 "ADC" 명령어를 사용한다.

기본적으로 덧셈 연산결과는 CPSR 레지스터의 상태 플래그(N,Z,C,V)에 반영되지 않는다. 이를 반영하기 위해서는 명령어 뒤에 'S'를 추가한다. 조건연산자와 함께 쓰일 때에는 조건연산자뒤에 'S'를 추가한다.

예) ADDS r0, r1, r2, ADDNES r0, r1, r2

쉬프트한 데이터를 입력으로 사용하고자 할 경우에는, 제일 뒷부분에 해당 데이터를 위치한다.

예) ADD r0, r1, r2 LSL #2

<Example>

1. 32비트 데이터 덧셈(캐리고려하지 않음, 상태플래그에 반영하지 않음)

MOV r0, #100

MOV r1, #200

ADD r2, r0, r1

Result : r2 = 300, C=unaffected

2. 32비트 데이터 덧셈(캐리고려하지 않음, 상태플래그에 반영)

LDR r0, =0xf0000000


LDR r1, =0xffffffff

ADDS r2, r0, r1

Result : r2 = 0x0eff0000, C=1

■ ADC reg1, reg2, reg3 => reg1:=reg2+reg3+C :

C(캐리)비트와 함께 reg2와 reg3를 더한 후, 그 결과를 reg1에 저장합니다.

 **Tip**

ADC(Add with Carry)

<Syntax>

ADD<cond>S <Rd>, <Rn>, <shifter_operand>

<Operation>

if ConditionPassed(cond) then

Rd = Rn + shifter_operand + C Flag

if S == 1 and Rd == R15 then

CPSR = SPSR

else if S == 1 then

N Flag = Rd[31]

Z Flag = if Rd == 0 then 1 else 0

C Flag = CarryFrom(Rn + shifter_operand + C Flag)

V Flag = OverflowFrom(Rn + shifter_operand + C Flag)

<Description>

덧셈 명령어인 "ADD" 명령어와 동일하게 32비트 덧셈 기능을 하지만, 덧셈 연산시 CPSR 레지스터에 있는 'C' 플래그를 반영한다. 32비트 이상의 큰 수를 더할 경우, 하위 32비트는 "ADD" 명령어를 상위 데이터를 더할 때에는 캐리를 고려하는 "ADC" 명령어를 사용한다.

CPSR 레지스터의 상태 플래그(N,Z,C,V)에 따른 조건연산을 하고자 할 때에는, ADC명령어 뒤에 조건연산자를 추가한다.

예) ADCNE, ADCEQ, ADCGT 등

기본적으로 덧셈 연산결과는 CPSR 레지스터의 상태 플래그(N,Z,C,V)에 반영되지 않는다. 이를 반영하기 위해서는 명령어 뒤에 'S'를 추가한다. 조건연산자와 함께 쓰일 때에는 조건연산자뒤에 'S'를 추가한다.

예) ADCS r0, r1, r2, ADCNES r0, r1, r2

쉬프트한 데이터를 입력으로 사용하고자 할 경우에는, 제일 뒷부분에 해당 데이터를 위치한다.

예) ADC r0, r1, r2 LSL #2

Example

1. 32비트 데이터 덧셈(캐리고려하지 않음, 상태플래그에 반영하지 않음)

Condition : C=1

MOV r0, #100

MOV r1, #200

ADD r2, r0, r1

Result : r2 = 301, C=1

2. 128비트 데이터 덧셈

Condition : C=1

MOV r0, #0x78 ;LSB

MOV r1, #0x56

MOV r2, #0x34

r3, #0x12 ;MSB

MOV r4, #0xff ;LSB

MOV r5, #0x00

MOV r6, #0xff

MOV r7, #0x00 ;MSB

ADDS r8, r0, r4 ; LSB 캐리고려할 필요없고, 상태 플래그 반영해야함

ADCS r9, r1, r5 ; 캐리고려하고, 상태 플래그 반영해야함

ADCS r10, r2, r6 ; 캐리고려하고, 상태 플래그 반영해야함

ADC r11, r3, r7 ; MSB 캐리고려하고, 상태 플래그 반영할 필요없음

■ SUB reg1, reg2, reg3 => reg1:=reg2-reg3 :

reg2에서 reg3를 뺀 결과를 reg1에 저장합니다.



Tip

SUB(Subtract)

<Syntax>

ADD<cond>S <Rd>, <Rn>, <shifter_operand>

<Operation>


```

if ConditionPassed(cond) then
    Rd = Rn + shifter_operand
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(Rn - shifter_operand)
        V Flag = OverflowFrom(Rn - shifter_operand)

```

<Description>

뺄셈 명령어로서 32비트 뺄셈 기능을 한다. 빌려오기(Borrow) 플래그는 반영하지 않는다.

CPSR 레지스터의 상태 플래그(N,Z,C,V)에 따른 조건연산을 하고자 할 때에는, SUB 명령어 뒤에 조건연산자를 추가한다.

예) SUBNE, SUBEQ, SUBGT 등

기본적으로 뺄셈 연산결과는 CPSR 레지스터의 상태 플래그(N,Z,C,V)에 반영되지 않는다. 이를 반영하기 위해서는 명령어 뒤에 'S'를 추가한다. 조건연산자와 함께 쓰일 때에는 조건연산자 뒤에 'S'를 추가한다.

예) SUBS r0, r1, r2, SUBNES r0, r1, r2

쉬프트한 데이터를 입력으로 사용하고자 할 경우에는, 제일 뒷부분에 해당 데이터를 위치한다.

예) SUB r0, r1, r2 LSL #2

<Example>

1. 32비트 데이터 뺄셈(캐리고려하지 않음, 상태플래그에 반영)

```

r0, #100
r1, #200
r2, r0, r1

```

Result : r2 = -100(0xfffff9c), C=0(B=1)

2. 32비트 데이터 뺄셈(캐리고려하지 않음, 상태플래그에 반영)

```

r0, #100
MOV    r1, #200
SUBS    r2, r1, r0

```

Result : r2 = 100(0x00000064), C=1(B=0)

- SBC reg1, reg2, reg3 => reg1:=reg2-reg3+(C-1) :
 reg2에서 reg3를 뺀 결과를 (C-1)를 더한 결과를 reg1에 저장합니다.
 C 비트의 의미가 not borrow이므로 (C-1)=B로 이해하면 됩니다.



Tip

SBC(Subtract with Carry)

<Syntax>

SBC<cond>S <Rd>, <Rn>, <shifter_operand>

<Operation>

if ConditionPassed(cond) then

Rd = Rn - shifter_operand - NOT(C Flag)

if S == 1 and Rd == R15 then

CPSR = SPSR

else if S == 1 then

N Flag = Rd[31]

Z Flag = if Rd == 0 then 1 else 0

C Flag = NOT BorrowFrom(Rn - shifter_operand - NOT(C Flag))

V Flag = OverflowFrom(Rn - shifter_operand - NOT(C Flag))

<Description>

뺄셈 명령어로서 32비트 뺄셈 기능을 한다. 빌려오기(Borrow) 플래그를 반영한다. CPSR 레지스터의 상태 플래그는 'B' 플래그가 없다. 따라서 "NOT(C Flag)"의 형태로 이를 사용한다.

32비트 이상의 큰 수를 뺄 경우, 하위 32비트는 "SUB" 명령어를 사용하고 상위 데이터를 뺄 때에는 빌려오기 비트를 고려하는 "SBC" 명령어를 사용한다.

- RSB reg1, reg2, reg3 => reg1:=reg3-reg2 :

SUB 명령어와 동일하나 reg2와 reg3의 순서가 바뀌었다고(reverse) 이해하면 됩니다.



Tip

RSB(Reverse Subtract)

<Syntax>

RSB<cond>S <Rd>, <Rn>, <shifter_operand>

<Operation>

if ConditionPassed(cond) then

Rd = shifter_operand - Rn

if S == 1 and Rd == R15 then

CPSR = SPSR

else if S == 1 then

N Flag = Rd[31]

Z Flag = if Rd == 0 then 1 else 0

C Flag = NOT BorrowFrom(shifter_operand-Rn)

V Flag = OverflowFrom(shifter_operand-Rn)


<Description>

32비트 뺄셈 연산기능을 수행하는 산술연산 명령어이다.

"SUB" 명령어와 입력 레지스터의 순서가 반전된 것 외에는 동일하다.

예) SUB r0, r1, r2, RSB r0, r2, r1은 동일하다.

- RSC reg1, reg2, reg3 => reg1:=reg3-reg2+(C-1) :
SBC 명령어와 동일하나 reg2와 reg3의 순서가 바뀌었다고 이해하면 됩니다.

 **Tip**

RSC(Reverse Subtract with Carry)

<Syntax>

RSC<cond>S <Rd>, <Rn>, <shifter_operand>

<Operation>

if ConditionPassed(cond) then

Rd = shifter_operand - Rn - NOT(C Flag)

if S == 1 and Rd == R15 then

CPSR = SPSR

else if S == 1 then

N Flag = Rd[31]

Z Flag = if Rd == 0 then 1 else 0

C Flag = NOT BorrowFrom(shifter_operand-Rn-NOT(C Flag))

V Flag = OverflowFrom(shifter_operand-Rn-NOT(C Flag))

<Description>

32비트 뺄셈 연산기능을 수행하는 산술연산 명령어이다.

"SBC" 명령어와 입력 레지스터의 순서가 반전된 것 외에는 동일하다.

예) SBC r0, r1, r2, RSC r0, r2, r1은 동일하다.

- reg1, reg2, reg3에는 레지스터가 위치할 수 있습니다.
레지스터의 ARM 명령어를 사용하는 경우 r0-r15까지 총 16개가 있으며, 레지스터 각각은 32bit 크기를 가지고 있습니다.

- reg1, reg2, reg3는 동일 레지스터를 사용하여도 무방합니다.

예) ADD r0,r0,r1

심화학습

SUB 명령어와 반전된 형태의 Reverse SUB명령어는 존재하는데, ADD 명령어와 반대가 되는 명령어는 왜 없는 것일까?

 **Tip**

☞ 당연한 얘기가 되겠지만, ADD 명령어 실행시 reg2+reg3 혹은 reg3+reg2의 결과가 동일하므로 필요가 없기 때문이죠.


사실 RSB 명령어도 필요성은 없어 보입니다. 아마도 32bit로 명령어를 표현하는데 있어 충분한 여유가 있어 만들어진 것이 아닌가 싶습니다.

4.2.2 Bit-wise Logical Operation

● 논리연산 명령어

- AND reg1, reg2, reg3 => reg1:=reg2 AND reg3 :

reg2와 reg3를 논리곱(AND) 연산 후, 그 결과를 reg1에 저장합니다.

 **Tip**

AND

<Syntax>

AND<cond>S <Rd>, <Rn>, <shifter_operand>

<Operation>

```
if ConditionPassed(cond) then
    Rd = Rn AND shifter_operand
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

<Description>


논리연산 명령어인 "AND" 로써 32비트 논리곱을 구한다. 상단에서 볼 수 있듯이 Rn과 shifter_operand와의 논리곱을 Rd에 저장한다. 연산결과를 CPSR레지스터에 있는 상태 플래그에 반영하기 위해서는 명령어 뒤에 'S'를 추가한다.

<Example>

```
LDR r0, =0x12345678
LDR r1, =0x0000ffff
AND r2, r0, r1
```

Result : r2 = 0x0000567

- ORR reg1, reg2, reg3 => reg1:=reg2 OR reg3 :
reg2와 reg3를 논리합(OR) 연산 후, 그 결과를 reg1에 저장합니다.

 **Tip**

ORR

<Syntax>

ORR<cond>S <Rd>, <Rn>, <shifter_operand>

<Operation>

```
if ConditionPassed(cond) then
    Rd = Rd OR shifter_operand
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
```

V Flag = unaffected

<Description>

논리연산 명령어인 "OR"로서 32비트 논리합을 구한다. 상단에서 볼 수 있듯이 Rn과 shifter_operand와의 논리합을 Rd에 저장한다. 연산결과를 CPSR레지스터에 있는 상태 플래그에 반영하기 위해서는 명령어 뒤에 'S'를 추가한다.

<Example>


```
LDR r0, =0x12345678
```

```
LDR r1, =0x0000ffff
```

```
ORR r2, r0, r1
```

Result : r2 = 0x1234ffff

- EOR reg1, reg2, reg3 => reg1:=reg2 XOR reg3 :
reg2와 reg3를 배타합(XOR) 연산 후, 그 결과를 reg1에 저장합니다.

 **Tip**

EOR

<Syntax>

EOR<cond>S <Rd>, <Rn>, <shifter_operand>

<Operation>

if ConditionPassed(cond) then

Rd = Rn EOR shifter_operand

if S == 1 and Rd == R15 then

CPSR = SPSR

else if S == 1 then

N Flag = Rd[31]

Z Flag = if Rd == 0 then 1 else 0

C Flag = CarryFrom(Rn + shifter_operand)

V Flag = OverflowFrom(Rn + shifter_operand)

<Description>

논리연산 명령어인 "XOR"로서 32비트 배타합을 구한다. 상단에서 볼 수 있듯이 Rn과 shifter_operand와의 배타합을 Rd에 저장한다. 연산결과를 CPSR레지스터에 있는 상태 플래그에 반영하기 위해서는 명령어 뒤에 'S'를 추가한다.


<Example>


```
LDR r0, =0xffffffff
```

```
LDR r1, =0x0000ffff
```

```
EOR r2, r0, r1
```

Result : r2 = 0xffff0000

-  BIC reg1, reg2, reg3 => reg1:=reg2 AND (NOT reg3) :
 reg2와 reg3의 반전된 값을 논리곱(AND) 연산 후, 그 결과를 reg1에 저장합니다.
 즉 reg3 레지스터 비트 중에서 '1'의 값을 갖는 위치의 reg2 비트값을 '0'으로 만듭니다.

 **Tip**

BIC(Bit Clear)

<Syntax>

BIC<cond>S <Rd>, <Rn>, <shifter_operand>

<Operation>

if ConditionPassed(cond) then

Rd = Rn AND (NOT shifter_operand)

if S == 1 and Rd == R15 then

CPSR = SPSR

else if S == 1 then

N Flag = Rd[31]

Z Flag = if Rd == 0 then 1 else 0

C Flag = shifter_carryout

V Flag = unaffected

<Description>

사용자가 원하는 특정 비트들만을 '0'으로 클리어하는 명령어입니다.

이와 반대의 기능을 하는 "BIS" 명령어는 없습니다.

일반적으로 인텔사의 펜티엄과 같은 32비트 프로세서에서는 비트

제어명령을 잘 지원하지 않습니다. 8051 혹은 PIC과 같은 8비트 마이크로 컨트롤러 등에서 이를 잘 지원하지요.

ARM 프로세서는 32비트 프로세서이지만, 제어분야 등에서 유용하게 사용될 수 있도록 비트 제어 명령을 지원합니다.

<Example>


LDR r0, =0x12345678

LDR r1, =0x0000ffff

BIC r2, r0, r1

Result : r2 = 0x12340000

-  BIC r0,r1,r2와 동일한 기능을 하되, MVN과 AND를 사용하여 재작성하시오.


 **Tip**

 MVN r2,r2 AND r0,r1,r2

▶ 심화학습

BIC(Bit Clear)명령어와 반대가 되는 BIS(Bit Set) 명령어는 없습니다. 필요가 없기 때문인데, 이럴 경우 어느 명령어를 사용해야 할까요?

 **Tip**


 BIS는 특정 비트만을 1로 만드는 기능을 수행할 것입니다.

이 기능은 ORR 명령어를 사용해서 구현할 수 있습니다.

4.2.3 Register Movement Operation

- 레지스터가 데이터 이동을 위한 명령어입니다. 레지스터는 r0-r15까지 16개라고 했지요?
16개 레지스터간의 데이터 이동을 의미하는 것입니다.

- MOV reg1, reg2 => reg1:=reg2 :
reg2의 내용을 reg1으로 복사합니다.

 **Tip**

MOV

<Syntax>

MOV<cond>S <Rd>, <shifter_operand>

<Operation>

```
if ConditionPassed(cond) then
  Rd = shifter_operand
  if S == 1 and Rd == R15 then
    CPSR = SPSR
  else if S == 1 then
    N Flag = Rd[31]
    Z Flag = if Rd == 0 then 1 else 0
    C Flag = shifter_carry_out
    V Flag = unaffected
```

<Description>

ARM 코어 레지스터간 데이터의 이동시 사용하는 명령어입니다. 데이터의 이동이지만, 외부 메모리와 레지스터간의 데이터 이동이 아니므로, 이 명령어는 "data processing instruction"에 해당합니다.

<Example>

- 의사 명령어 "NOP"를 대치할 경우

MOV r0, r0, r0


- 레지스터의 내용을 단순히 쉬프트만 하고자 할 경우

MOV r0, r0, LSL #2

- 레지스터의 특정비트의 값이 '1' 인지 '0'인지를 확인하고자 할 경우
(예는 0번 비트)

```
MOVS r0, r0, LSR #1
BCS LOOP_BIT1
BCC LOOP_BIT0
```

- MVN reg1, reg2 => reg1:=NOT reg2 :
reg2의 반전된 내용을 reg1으로 복사합니다.

 **Tip**

MVN

<Syntax>

MVN<cond>S <Rd>, <shifter_operand>

<Operation>

```
if ConditionPassed(cond) then
    Rd = NOT shifter_operand
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

<Description>

ARM 코어 레지스터간의 데이터 이동에 사용된다. 단, 레지스터의 논리 반전된 결과를 저장한다. 주의할 점은 양수, 음수와 같이 부호가 반전되는 것이 아니라는 것이다.

<Example>

LDR r1, =0x0000ffff


MVN r0, r1

Result : r0 = 0xffff0000

4.2.4 Comparison Operation

- 다른 산술명령어와는 달리 결과값을 출력하지 않는 명령어입니다.
따라서 2개의 데이터 입력값만이 존재합니다.
결과값을 출력하지는 않지만, 결과에 따른 상태플래그(N,Z,C,V)에는 영향을 줍니다.

- CMP reg1, reg2 => reg1-reg2 :
reg1에서 reg2를 뺀 결과값에 따른 상태플래그를 반영합니다.
reg1과 reg2중 어느 값이 큰지 혹은 작은지 등을 비교하는데 사용합니다.

 **Tip**

CMP

<Syntax>

CMP<cond> <Rn>, <shifter_operand>

<Operation>

```
if ConditionPassed(cond) then
    alu_out = Rn - shifter_operand
N Flag = alu_out[31]
Z Flag = if alu_out == 0 then 1 else 0
C Flag = CarryFrom(Rn + shifter_operand)
```


V Flag = OverflowFrom(Rn + shifter_operand)

<Description>

두 수를 비교하는 명령어로 2개의 입력 데이터를 뺄셈 처리한 후,
그 결과를 CPSR 레지스터에 있는 상태 플래그(N,Z,C,V)에만 영향을 미친다.

<Example>


LDR r0, =0x100

LDR r1, =0x200

Result : C=0(B=1)

- CMN reg1, reg2 => reg1+reg2 :

reg1에서 reg2를 더한 결과값에 따른 상태플래그를 반영합니다.

 **Tip**

CMN

<Syntax>

CMN<cond> <Rn>, <shifter_operand>

<Operation>

if ConditionPassed(cond) then

alu_out = Rn + shifter_operand

N Flag = alu_out[31]

Z Flag = if alu_out == 0 then 1 else 0

C Flag = CarryFrom(Rn + shifter_operand)

V Flag = OverflowFrom(Rn + shifter_operand)

<Description>

두 수를 비교하는 명령어로 2개의 입력 데이터를 덧셈 처리한 후,
그 결과를 CPSR 레지스터에 있는 상태 플래그(N,Z,C,V)에만 영향을 미친다.

<Example>


LDR r0, =0x100

LDR r1, =0x200

Result : C=0(B=1)

- TST reg1, reg2 => reg1 AND reg2 :

reg1와 reg2의 논리곱(AND) 연산을 한 결과값에 따른 상태플래그를 반영합니다.

 **Tip**

TST

<Syntax>

TST<cond> <Rn>, <shifter_operand>

<Operation>

```
if ConditionPassed(cond) then
    alu_out = Rn AND shifter_operand
    N Flag = alu_out[31]
    Z Flag = if alu_out == 0 then 1 else 0
    C Flag = shifter_carry_out
    V Flag = unaffected
```

<Description>


32비트 2개의 데이터를 비교하는 명령어이다. 연산기능은 "AND"를 사용한다.

<Example>

```
LDR r0, =0x12345678
LDR r1, =0x000
```

Result : Z=1

- TEQ reg1, reg2 => reg1 XOR reg2 :
reg1와 reg2의 배타합의 결과값에 따른 상태플래그를 반영합니다.
reg1과 reg2가 같은 값을 갖는지를 비교하는데 사용합니다.

 **Tip**

TEQ

<Syntax>

TEQ<cond> <Rn>, <shifter_operand>

<Operation>

```
if ConditionPassed(cond) then
    alu_out = Rn EOR shifter_operand
    N Flag = alu_out[31]
    Z Flag = if alu_out == 0 then 1 else 0
    C Flag = shifter_carry_out
    V Flag = unaffected
```

<Description>

32비트 2개의 데이터가 동일한지를 확인하는데 사용되는 비교명령어이다. 연산기능은 "XOR"를 사용하며, CPSR 레지스터의 'Z' 플래그를 이용한다.

<Example>

```
LDR r0, =0x100
LDR r1, =0x100
TEQ r0, r1
```

BEQ TEST ; 통상 같은지를 비교할 때 사용한다.

Result : Z=1

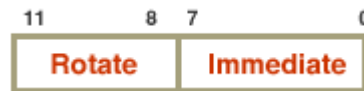
4.2.5 Immediate Operand

- 산술연산 과정에서 상수값을 사용하고자 할 경우, 어떤 방법이 가능한지 알아봅니다.

명령어 reg1, reg2, reg3

- reg1, reg2, reg3과 같은 레지스터 대신에 상수값을 사용할 수 있습니다.
- 상수 데이터를 의미하기 위해서 '#' 기호를 사용합니다.

예) ADD r3, r3, #4



- 상수 데이터는 총 12bit로, 4bit의 로테이트와 8bit의 상수로 구성되어있습니다.
- 따라서 로테이트를 하지 않을 경우 상수 데이터의 범위는 0-255(8bit)입니다.
- 상수 데이터의 범위는 0-4095가 아닌 $(0 \rightarrow 255) \times 2^{2n}$ 입니다. ($0 \leq n \leq 12$)

심화학습

다음 상수 데이터 중 바른 데이터 값과 그렇지 않은 값들을 구별하시오.

0x101, 0x104, 0xff1, 0xff00, 0xff000, 0xff, 0x102, 0xff0, 0xff04, 0xff003



- Valid Constants : 0xff, 0x102, 0x104, 0xff0, 0xff00, 0xff000
- Invalid Constants : 0x101, 0xff1, 0xff04, 0xff003

- 10진수 데이터를 사용할 경우, '#'뒤에 십진수 데이터 값을 적습니다.
- 16진수 데이터를 사용할 경우, '#0x123' 혹은 '#&123'과 같이 '0x', '&'기호를 이용해 16진수 데이터임을 표시해 줍니다.

4.2.6 Shifted Register Operand


- 일반 프로세서에서와는 달리, ARM 명령어에서 쉬프트 명령어는 다른 명령어와의 조합을 통해서만 사용이 가능합니다. 어떤 관점에서 보면 불편하다는 생각이 들 수도 있지만, 쉬프트 명령어와 다른 명령어가 동시 실행이 가능하도록 하기 위한 것입니다.
- ADD 명령어와 쉬프트 명령어들이 동시에 수행되는 상황을 고려해 봅니다.

ADD reg1, reg2, reg3

- ADD reg1, reg2, reg3, LSL #2 => reg1:=reg2 + (reg3<<2) :**
reg3를 왼쪽으로 2비트 쉬프트한 후, reg2와 덧셈을 합니다. 덧셈한 최종 결과를 reg1에 저장합니다.
- ADD reg1, reg2, reg3, LSL reg4 => reg1:=reg2 + (reg3<<reg4) :**
reg3를 왼쪽으로 reg4에 저장된 값만큼 쉬프트한 후, reg2와 덧셈을 합니다. 덧셈한 최종 결과를 reg1에 저장합니다.

심화학습

단순히 r0 레지스터의 내용을 쉬프트 시키고자 합니다. 독자적인 쉬프트 명령어가 존재하지 않는데, 어떻게 해야 할까요?

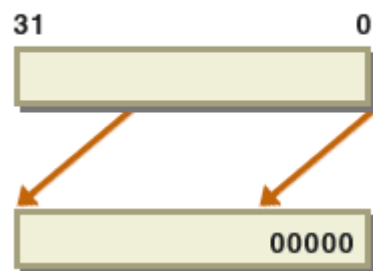
 **Tip**

☞ 쉬프트 명령어는 독자적으로 수행될 수 없습니다. 따라서 MOV 명령어와 조합해서 사용합니다. 예를 들어 r0 레지스터의 내용을 좌측으로 2비트 쉬프트하고자 할 경우, 아래와 같이 작성합니다.

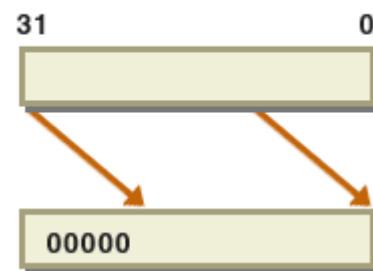
MOV r0, r0, LSL #2

● 위와 같이 수행된 쉬프트 명령어에는 다음과 같은 명령어들이 있습니다.

■ **LSL** (예 LSL #5)

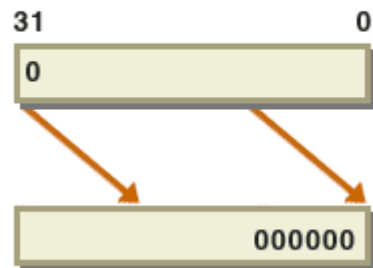


■ **LSR** (예 LSR #5)

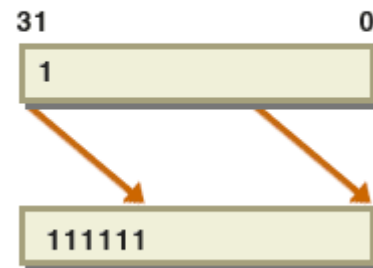


■ **ASL**

■ **ASR**

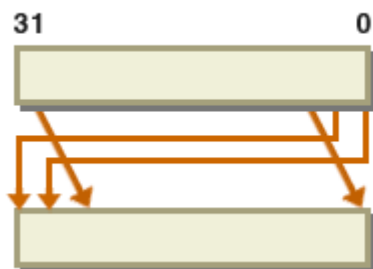


ASR #5, positive operand

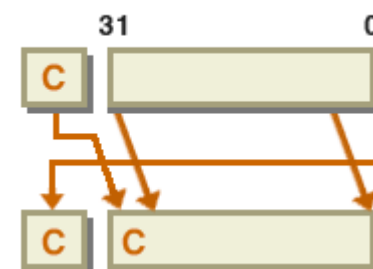


ASR #5, negative operand

■ **ROR** (예 ROR #5)



■ **RRX** (예 RRX)



4.2.7 Multiplies

● ARM7은 기본적으로 32X8 Booth Multiplier가 들어있습니다. 따라서 32bit 결과를 얻는 MUL/MLA 명령어의 경우 최대 4번 반복적으로 계산을 합니다. 64bit 결과를 얻는 SMULL/SMLAL, UMULL/UMLAL 명령어의 경우 최대 8번 반복 계산을 합니다

심화학습

r1=0x12345678, r2=0x12가 저장되어 있습니다.

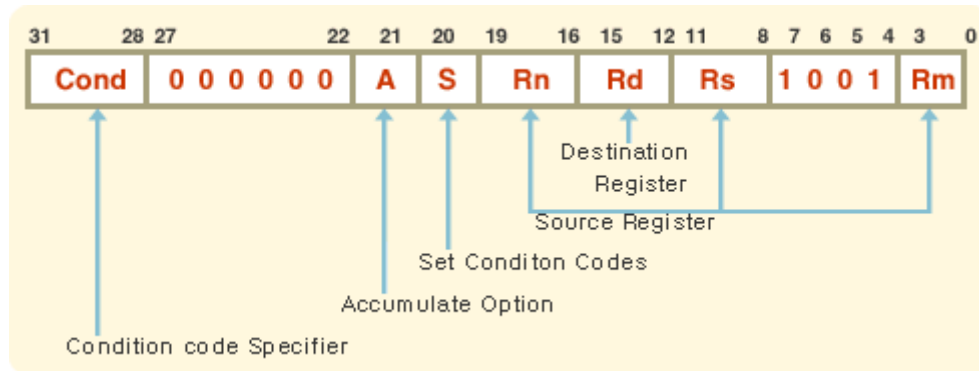
MUL r0, r1, r2로 작성한 것과 MUL r0, r2, r1으로 작성한 것 중 연산속도는 어느 것이

빠를까요? **Tip**

MUL r0, r1, r2의 경우에는 r2의 내용이 8bit 이므로 32X8 Booth Multiplier가 1번 실행합니다.

그러나 MUL r0, r2, r1의 경우에는 4번 실행되어야 하므로 전자의 경우가 3싸이클 빠릅니다.

- MUL(Multiply)/MLA(Multiply accumulate) :32X32의 연산결과 32bit를 구하는데 사용합니다.



- MUL reg1, reg2, reg3 => reg1:=reg2*reg3 :

reg3를 왼쪽으로 2비트 쉬프트한 후, reg2와 덧셈을 합니다.덧셈한 최종 결과를 reg1에 저장합니다.

Tip

MUL

<Syntax>

MUL{<cond>}S <Rd>, <Rm>, <Rs>

<Operation>

if ConditionPassed(cond) then

Rd = (Rm * Rd)[31:0]

if S == 1 then

N Flag = Rd[31]

Z Flag = if Rd == 0 then 1 else 0

C Flag = unaffected

V Flag = unaffected

< Description>

32비트 데이터의 곱셈 연산을 위해 사용된다. 연산결과 64비트 중 하위 32비트 만을 결과 레지스터에 저장한다.

<Example>

LDR r0, =0x123

LDR r1, =0x100

MUL r3, r0, r1

Result : r3=0x12300

- MLA reg1, reg2, reg3 => reg1:=reg2*reg3



Tip

MLA

<Syntax>

MLA<cond>S <Rd>, <Rm>, <Rs>, <Rn>

<Operation>

if ConditionPassed(cond) then

Rd = (Rm * Rs + Rn)[31:0]

if S == 1 then

N Flag = Rd[31]

Z Flag = if Rd == 0 then 1 else 0

C Flag = unaffected

V Flag = unaffected

<Description>

곱셈과 덧셈 연산을 동시에 수행하는 "MAC" 명령어입니다. 32비트 곱셈 연산과 덧셈연산을 수행한 후, 하위 32비트 데이터만을 레지스터에 저장합니다. 32*8bit Booth Multiplier를 사용하므로 실행시 여러 사이클을 소모합니다.

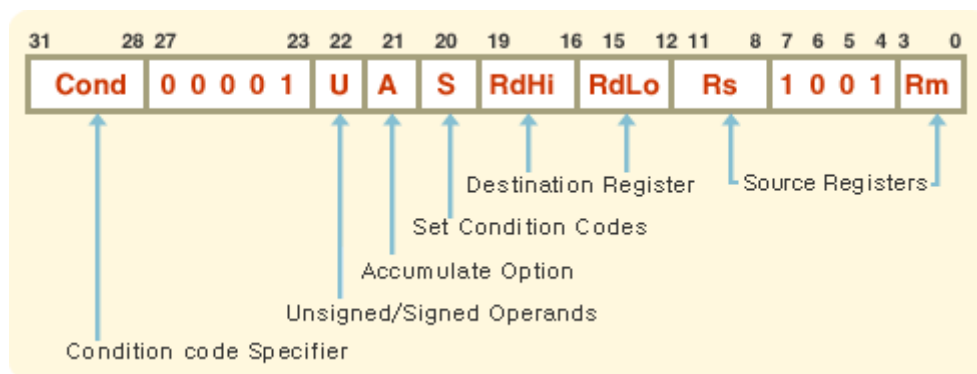
<Example>

LDR r0, =0x123

LDR r1, =0x100

LDR r2, =0x50

MUL r3, r0, r1, r2



- SMULL(Signed multiply long)/SMLAL(Signed multiply accumulate long) :

부호가 있는 32X32의 연산결과 64bit를 구하는데 사용합니다.

- SMULL reg1, reg2, reg3, reg4 => (reg1,reg2):=reg3*reg4



Tip

SMULL

<Syntax>

SMLAL<cond>S <RdLo>, <RdHi>, <Rm>, <Rs>

<Operation>

```
if ConditionPassed(cond) then
    RdLo = (Rm * Rs)[31:0]
    RdHi = (Rm * Rs)[63:32]
if S == 1 then
    N Flag = RdHi[31]
    Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
    C Flag = unaffected
    V Flag = unaffected
```

<Description>

"MUL" 명령어가 32비트 데이터 곱셈결과 64비트 중, 하위 32비트만을 처리하는데 반하여 "SMULL"은 64비트 결과를 모두 처리한다. 또한 32비트 결과를 얻을 경우에는 부호비트가 버려지나, 이 명령어는 입력 데이터의 부호를 고려해 결과를 출력한다.

● SMLAL reg1, reg2, reg3, reg4 => (reg1,reg2):=reg3*reg4+(reg1,reg2)



Tip

SMLAL

<Syntax>

SMLAL{<cond>}S <RdLo>, <RdHi>, <Rm>, <Rs>

<Operation>

```
if ConditionPassed(cond) then
    RdLo = (Rm * Rs)[31:0] + RdLo
    RdHi = (Rm * Rs)[63:32] + RdHi + CarryFrom(Rm * Rs)[31:0]
    + RdLo)

if S == 1 then
    N Flag = RdHi[31]
    Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
    C Flag = unaffected
    V Flag = unaffected
```

<Description>

부호가 있는 32비트 데이터를 곱셈연산 처리하여, 64비트 결과를 얻고자 할 경우 사용된다. "MUL" 명령어가 단지 곱셈연산만을 수행하는데 반하여 "MLA" 명령어가 곱셈과 덧셈을 수행하는 것과 동일하게 "SMLAL" 명령어도 64비트 곱셈결과와 64비트 레지스터의 내용을 덧셈하는 연산을 동시에 수행한다.

● UMULL(Unsigned multiply long)/UMLAL(Unsigned multiply accumulate long) :

부호가 없는 32X32의 연산결과 64bit를 구하는데 사용합니다.

● UMULL reg1, reg2, reg3, reg4 => (reg1,reg2):=reg3*reg4



Tip

UMULL

<Syntax>

UMULL{<cond>}S <RdLo>, <RdHi>, <Rm>, <Rs>

<Operation>

```

if ConditionPassed(cond) then
    RdHi = (Rm * Rs)[63:32]
    RdLo = (Rm * Rs)[31:0]
if S == 1 then
    N Flag = RdHi[31]
    Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
    C Flag = unaffected
    V Flag = unaffected

```

<Description>

"MUL" 명령어가 32비트 데이터 곱셈결과 64비트 중, 하위 32비트만을 처리하는데 반하여 "SMULL"은 64비트 결과를 모두 처리한다.

또한 32비트 결과를 얻을 경우에는 부호비트가 버려지나, 이 명령어는 입력 데이터의 부호를 고려해 결과를 출력한다.

UMLAL reg1, reg2, reg3, reg4 => (reg1,reg2):=reg3*reg4+(reg1,reg2)



Tip

UMLAL

<Syntax>

UMLAL{<cond>}S <RdLo>, <RdHi>, <Rm>, <Rs>

<Operation>

```

if ConditionPassed(cond) then
    RdLo = (Rm * Rs)[31:0] + RdLo
    RdHi = (Rm * Rs)[63:32] + RdHi + CarryFrom((Rm * Rs)[31:0]
    + RdLo)
if S == 1 then
    N Flag = RdHi[31]
    Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
    C Flag = unaffected
    V Flag = unaffected

```

<Description>

부호가 없는 32비트 데이터를 곱셈연산 처리하여, 64비트 결과를 얻고자 할 경우 사용된다. "MUL" 명령어가 단지 곱셈연산만을 수행하는데 반하여 "MLA" 명령어가 곱셈과 덧셈을 수행하는 것과 동일하게 "UMLAL" 명령어도 64비트 곱셈결과와 64비트 레지스터의 내용을 덧셈하는 연산을 동시에 수행한다.

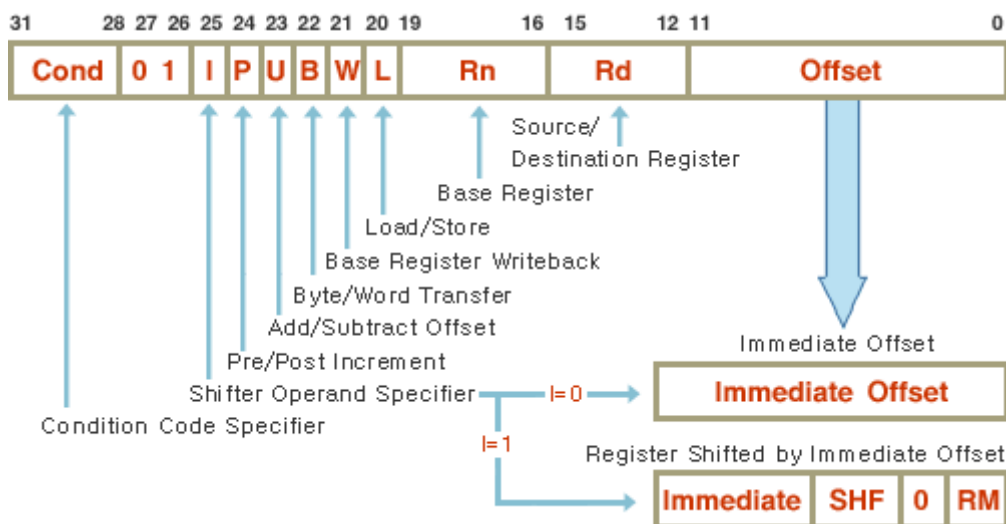
* 주의사항 ==> reg1 : Lower Data / reg2 : Higher Data

4.3 Data Transfer Instruction

- 데이터 전송을 위한 아래의 명령어들을 소개합니다.
- LDR
- STR
- LDMIA
- STMIA
- SWAP
- 데이터 전송은 ARM 프로세서 외부에 있는 메모리공간과 내부 레지스터간의 이동을 의미합니다.
- 데이터 전송명령 사용시 각 어드레싱 모드를 설명합니다.

4.3.1 Single Register Load/Store Instruction

- 32bit 데이터를 메모리 공간에서 읽거나 쓸때 사용하는 명령어 LDR/STR를 공부합니다.
LDR/STR 명령어는 아래의 설명에서 보듯이 reg1, [reg2]의 형태로 동일하나, 데이터의 이동방향이 반대인 것에 유의해야 합니다.



- **LDR reg1, [reg2] => reg1:=mem32[reg2] :**
reg2가 가르키는 메모리의 주소공간에서 32bit 데이터를 읽어와 reg1에 저장한다.
- **STR reg1, [reg2] => mem32[reg2]:=reg1 :**
reg1에 저장되어 있는 32bit 데이터를 reg2가 가르키는 메모리의 주소공간에 저장한다.
- LDR/STR 명령어 사용시 전송될 데이터의 크기는 아래와 같이 결정됩니다.
: 8bit => LDRB/STRB, LDRSB, LDRBT
: 16bit => LDRH/STRH, LDRSH
: 32bit => LDR/STR, LDRT

Tip

LDR

<Syntax>

LDR{<cond>} <Rd>, <addressing_mode>

<Operation>

```
if ConditionPassed(cond) then
  if address[1:0] == 0B00 then
    value = Memory[address,4]
  else if address[1:0] == 0b01 then
    value = Memory[address,4] Rotate_Right 8
  else if address[1:0] == 0b10 then
    value = Memory[address,4] Rotate_Right 16
  else /* if address[1:0] == 0b11 then */
    value = Memory[address,4] Rotate_Right 24
```

<Description>

외부 메모리로부터 ARM 코어 내부의 레지스터로 32비트의 데이터를 읽어오는데 사용한다. 다양한 어드레싱 모드를 이용하여 데이터를 읽어들이 수 있다.

<Example>

```
LDR r0, =0x100
LDR r1, =0x10
STR r1, [r0]
LDR r0, =0x104
LDR r1, =0x20
STR r1, [r0]
```

1.Pre-Indexed Addressing Mode

```
LDR r0, =0x100
LDR r2, [r0, #4]
```

Result : r0=0x100, r2=0x20

2.Post-Indexed Addressing Mode

```
LDR r0, =0x100
LDR r2, [r0], #4
```

Result : r0=0x104, r2=0x10

3.Auto-Indexed Addressing Mode

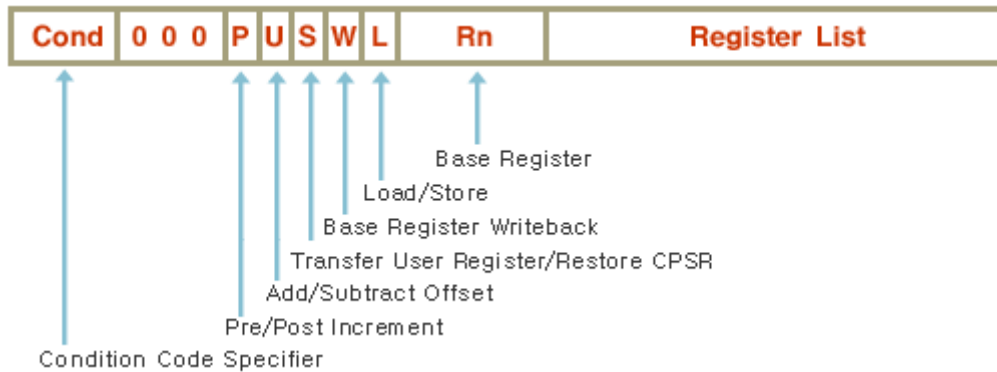
```
LDR r0, =0x100
LDR r2!, [r0, #4]
```

Result : r0=0x104, r2=0x2

4.3.2 Multiple Register Data Transfer

- 여러 개의 데이터를 읽거나 쓸때 사용하는 명령어가 Multiple Register Data Transfer 명령어입니다. Single Register Load/Store Instruction을 반복해서 사용하는 것보다 빠릅니다.

- LDR/STR 명령어를 사용할 때의 데이터 이동방향과 반대임에 유의합니다.
- {...} 안의 레지스터의 순서에 관계없이 낮은 주소의 데이터가 낮은 번호의 레지스터에 저장됩니다.



4.3.3 Addressing Mode

- 데이터 전송 시 사용되는 어드레싱 모드는 아래의 3가지가 있습니다.

◆ Pre-indexed Addressing Mode

LDR reg1, [reg2, #4] ; reg1:=mem32[reg2+4]

: (reg2+4)의 메모리 주소공간에서 데이터를 읽어와, reg1에 저장한다.

명령어의 실행이후 reg2의 내용은 변하지 않는다.

◆ Post-indexed Addressing

LDR reg1, [reg2], #4 ; reg1:=mem32[reg2]

; reg2:=reg2+4

: (reg2)의 메모리 주소공간에서 데이터를 읽어와, reg1에 저장한다.

명령어의 실행이후 reg2의 내용은 4 증가한다.

◆ Auto-indexing Addressing Mode

LDR reg1, [reg2, #4]! ; reg1:=mem32[reg2+4]

; reg2:=reg2+4

: (reg2+4)의 메모리 주소공간에서 데이터를 읽어와, reg1에 저장한다.

명령어의 실행이후 reg2의 내용은 4 증가한다.

4.3.4 Stack Control Instruction

- PC에서 스택에 데이터를 관리하는데 사용하는 명령어는 PUSH/POP 명령어이다.

ARM에서도 스택에 관련된 명령어가 있다. 모든 스택구조를 지원하기 위해 여러 개의 명령어가 존재한다. 스택전용으로만 사용되는 것이 아니라, 일반 데이터의 전송에도 사용한다. 어드레싱 모드에 따라 아래와 같은 적절한 OP CODE가 생성된다.

[LDM-1 / LDM-2 / LDM-3](#)

[STM-1 / STM-2](#)

* The mapping between the stack and block copy views of the load and store multiple instructions.

		Ascending		Descending	
		Full	Empty	Full	Empty
Increment	Before	STMIB STMFA			LDMIB LDMED
	After		STMIA STMEA	LDMIA LDMFD	
Decrement	Before		LDMDB LDMEA	STMDB STMFD	
	After	LMDMA LDMFA			STMDA STMED

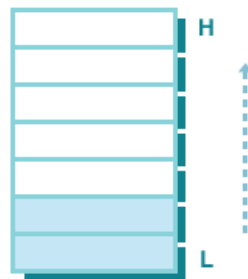
● 스택 주소에 따라 크게 4가지 형태가 존재한다.

- Full Ascending
- Empty Ascending
- Full Descending
- Empty Descending

● Ascending

: 스택에 데이터를 넣을 때마다, 스택의 증가방향이 어드레스가 증가하는 방향이다.

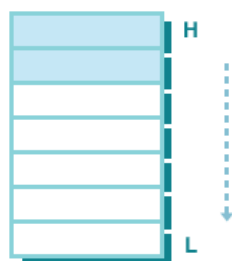
즉 하위 어드레스에서 상위 어드레스로 스택이 쌓이게 된다. 이 경우 스택의 시작 주소는 하위 어드레스 영역에서 시작한다.



● Descending

: 스택에 데이터를 넣을 때마다, 스택의 증가방향이 어드레스가 감소하는 방향이다.

즉 상위 어드레스에서 하위 어드레스로 스택이 쌓이게 된다. 이 경우 스택의 시작 주소는 상위 어드레스 영역에서 시작한다. ARM에서는 모든 스택 방식을 지원하지만, 삼성에서는 모두 이 방식을 사용한다.



● Full

: 스택에 데이터를 넣은 후, 스택 포인터가 가르치는 주소에 데이터가 차 있는 방식을 말한다.



● Empty

: 스택에 데이터를 넣은 후, 스택 포인터가 가르치는 주소에 데이터가 비어있는 방식을 말한다.



● 스택 명령어의 분류

: ARM에서 지원하는 스택 명령어는 앞의 표에서 보듯이 아주 다양합니다.

모든 스택 명령어는 명령어의 끝부분(접미사)를 통해 그 종류를 쉽게 구분할 수 있습니다.

IB (Increment & Before)

: 스택에 데이터를 넣기 전(before)에 어드레스를 증가(increment) 시킵니다.

IA (Increment & After)

: 스택에 데이터를 넣은 후(after)에 어드레스를 증가(increment) 시킵니다.

DB (Decrement & Before)

DA (Decrement & After)

EA (Empty & Ascending)

: 스택은 어드레스가 증가하는 방향으로 자라나므로, increment와 같습니다.

스택에 데이터를 넣은 후, 스택 포인터가 가르치는 곳은 비어있습니다.

즉, after와 같습니다.

ED (Empty & Descending)

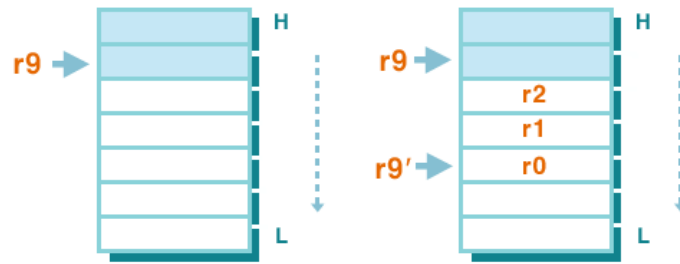
FA (Full & Ascending)

FD(Full & Descending)

● 스택에 많이 사용되는 STMFD/LDMFD 명령어에 대해 살펴봅니다.

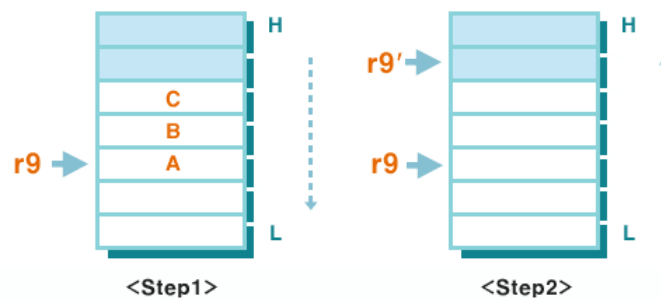
STMFD r9!, {r0, r1, r2}

: 이 명령어는 앞의 표에서 보면, full 방식입니다. 아래의 그림을 보면 마지막 스택 포인터가 가르치는 주소에 데이터가 있습니다. 또한 데이터를 넣기 전, 주소를 감소시키므로 decrement & before가 맞습니다. 스택이 하위 주소로 증가하므로, descending 방식입니다. 앞의 표에서의 의미를 살펴봅니다.



LDMFD r9!, {r0, r1, r2}

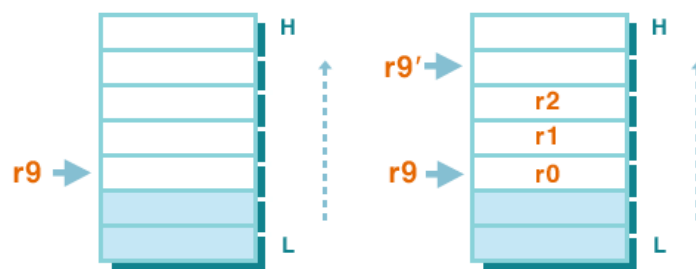
: 이 명령어는 앞의 표에서 보면, full 방식입니다. 아래의 그림을 보면 마지막 스택 포인터가 가르치는 주소에 데이터가 있습니다. 또한 데이터를 읽은 후, 주소를 증가시키므로 increment & after가 맞습니다. 스택이 하위 주소로 증가하므로, descending 방식입니다. 앞의 표에서의 의미를 살펴봅니다.



● 스택에 이용하는 것보다는 여러 개의 데이터를 읽거나 쓰는 경우에 주로 사용합니다. 특별히 "system management" 부분에서 많이 사용합니다.

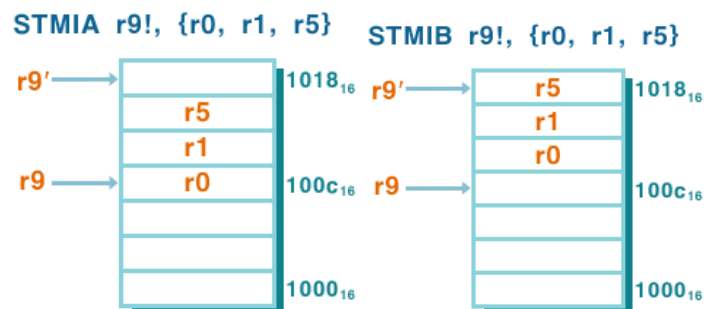
STMIA r9!, {r0, r1, r2}

: 이 명령어는 앞의 표에서 보면, empty 방식입니다. 아래의 그림을 보면 마지막 스택 포인터가 가르치는 주소에 데이터가 없습니다. 또한 데이터를 넣은 후, 주소를 감소시키므로 increment & after가 맞습니다. 스택이 상위 주소로 증가하므로, ascending 방식입니다. 앞의 표에서의 의미를 살펴봅니다.

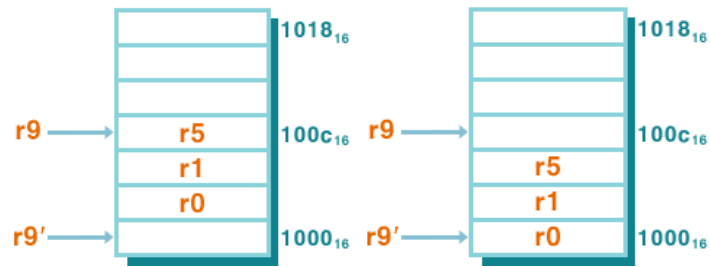


LDMIA r9!, {r0, r1, r2}

: 이 명령어는 LDMFD와 동일합니다. LDMFD 부분을 참조하시기 바랍니다.



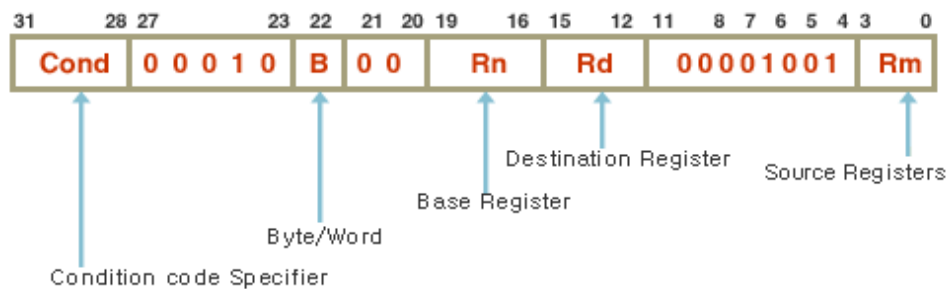
STMDA r9!, {r0, r1, r5} STMDB r9!, {r0, r1, r5}



4.3.5 Swp Instruction

- SWP 명령어는 데이터 교환을 위해 사용됩니다. 이터 정렬(sort)등에 널리 사용되지요. ARM 명령어에서의 "SWP" 명령어는 아래의 4가지 경우 중 2, 3번의 경우에 사용할 수 있습니다.

- 1) register-register swp 2) register-memory swp
- 3) memory-register swp 4) memory-memory swp



- SWP reg1, reg2, [reg3] => reg1:=mem32[reg3]
mem32[reg3]:=reg2

: 메모리에서 레지스터로, 레지스터에서 메모리로 데이터를 이동시킬 수 있습니다.

: reg1과 reg2가 같은 레지스터라면, "SWAP"의 기능으로 사용이 가능합니다.

Tip

SWP

<Syntax >

SWP{<cond>} <Rd>, <Rm>, [<Rd>]

< Operation>

```
if ConditionPassed(cond) then
  if Rn[1:0] == 0b00 then
    temp = Memory[Rn,4]
  else if Rn[1:0] == 0b01 then
    temp = Memory[Rn,4] Rotate_Right 8
  else if Rn[1:0] == 0b10 then
    temp = Memory[Rn,4] Rotate_Right 16
  else /* Rn[1:0] == 0b11 then */
```

temp = Memory[Rn,4] Rotate_Right 24

Memory[Rn,4] = Rm

Rd = temp

<Description>

이 명령어는 ARM 코어 레지스터의 데이터와 외부 메모리 상의 데이터를 교환하는데 사용한다. 혹은 이 명령어를 종종 "semaphore" 명령어로 부릅니다. 이는 멀티태스킹 OS상에서 자원에 대한 권한을 제어하는데 사용하기 때문입니다.

<Example>

LDR r0, =0x100

LDR r1, =0x200

LDR r2, =0x1000

STR r1, [r2]

SWP r0, r0, [r2]

Result : r0=0x200, [r2]=0x100

Result : r2 = 0x0eff0000, C=1

▣ 심화학습

SWP 명령어는 멀티태스킹 OS 운영체제에서 사용됩니다. 어떤 용도로 사용될까요?

Tip

☞ 멀티태스킹 OS에서 세마포어는 리소스에 대한 소유권을 할당하는데 사용합니다.

SWP 명령어를 사용해서 소유권을 여러 태스킹에서 번갈아 사용하는데 사용합니다.

LDR/STR 명령어를 사용할 경우, exception이나 interrupt가 발생하는 경우 문제가 발생할 수 있으므로 SWP 명령어를 사용합니다.

● SWPB reg1, reg2, [reg3] :

바이트(8bit) 단위의 데이터 이동시 사용됩니다.

Tip

SWPB

<Syntax >

SWPB{<cond>}B <Rd>, <Rm>, [<Rn>]

< Operation>

if ConditionPassed(cond) then

temp = Memory[Rn,1]

Memory[Rn,1] = Rm[7:0]

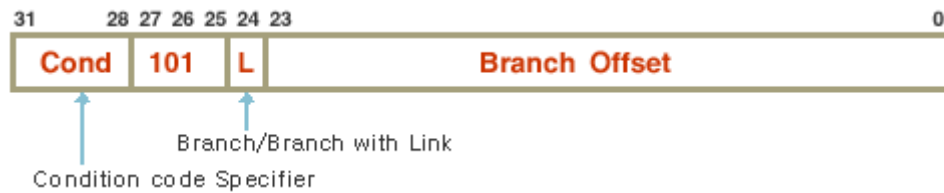
Rd = temp

< Description>

"SWP" 명령어와 동일하게 ARM 코어 레지스터와 외부 메모리간의 데이터 교환을 수행하되, 8비트 데이터를 처리한다.

4.4 Flow Control Instruction

이제 분기명령어들을 살펴봅니다.



- Branch Instruction
- Conditional Branch
- Branch and Link
- Supervisor Call

4.4.1 Branch Instruction

- 무조건 분기 명령어는 PC에서는 "JMP"입니다. ARM에서는 Branch의 약자 'B'입니다.
- 무조건 분기명령어 'B' 를 아래의 예를 들어 설명합니다.

Tip

B, BL

<Syntax >

BL<cond> <target_address>

< Operation>

if ConditionPassed(cond) then

if L == 1 then

LR = 분기명령어 다음 명령어의 주소(즉, 되돌아 올 주소)

PC = PC + (SignExtend(signed_immed_24) << 2)

< Description>

분기명령어에는 위에서 보는 바와 같이 B, BL 2개의 명령어가 있습니다. 'B' 명령어는 무조건 분기 명령어로 PC의 "JMP" 명령어와 동일한 기능을 합니다. 반면에 서브루틴을 호출하는 "CALL"에 해당하는 명령어가 "BL" 명령어입니다. "BL" 명령어를 호출하면 되돌아올 주소를 LR 레지스터에 보존합니다.

<Example>

1. 무조건 분기 명령어를 사용할 경우

B TEST

...

TEST

...

2. 조건 분기 명령어를 사용할 경우

LDR r0, =0x100

LDR r1, =0x200

CMP r0, r1

BGT TEST

...

TEST

...

3. 서브 루틴을 호출하는 경우

BL ARM_SUBR

...

SUBR

...

MOV PC, LR

- "B TEST" 명령어를 만나면, "TEST" 레이블이 있는 위치로 이동합니다. 동일한 이름이 레이블이 여러 개 있을 경우에는 에러가 발생합니다.

B TEST

...

TEST

...

...

- 무조건 분기명령어 "%B?", "%F?"를 설명합니다. 'B'는 back의 의미로 현재 명령어의 뒷부분에 있는 숫자 레이블 "01"의 위치로 이동합니다. 숫자 레이블 "01"이 여러 개 있는 경우, 가장 근접한 위치에 있는 숫자 레이블로 이동합니다. 즉, 동일한 이름의 레이블이 여러개 있을 경우에도 에러가 발생하지 않습니다. 'F'는 forward의 의미로 현재 명령어의 앞부분에 가장 근접해 있는 숫자 레이블 "02"의 위치로 이동합니다.

01

...

...

01

...

B %B01

B %F02

...

02

...

4.4.2 Conditional Branch

- ARM에서는 다양한 조건분기 명령어를 지원합니다.

Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BAL	Always	Always take this branch
BEQ	Equal	Comparison equal or non-zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLC	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out
BHS	Higher or 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 or equal	Signed integer comparison gave greater 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 same	Unsigned comparison gave lower or same

- 분기시 조건은 PSR(Program Status Register)의 Condition Flag(N,Z,C,V)값을 이용합니다.

▶ 심화학습

조건분기 명령어는 Condition Flag에 따라서 분기를 결정합니다.

N,Z,C,V에 따른 조건상태를 알아봅시다.



Tip

Suffix	Flags	Meaning
EQ	Z set	equal
NE	Z clear	not equal
CS/HS	C set	higher or same(Unsigned >=)
CC/LO	C clear	lower(Unsigned <)
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	higher(Unsigned >)
LS	C clear and Z set	lower or same(Unsigned <=)
GE	N and V the same	signed >=
LT	N and V differ	signed <
GT	Z clear, N and V the same	signed >
LE	Z set, N and V differ	signed <=

4.4.3 Branch and Link

- 무조건 분기 명령어인 "BL"을 설명합니다.

"BL" 명령어는 무조건 분기 명령어인 "B"와 동작은 동일합니다. 단, PC(r15)의 값을 R(r14) 레지스터에 저장하고 분기한다는 차이점이 있습니다. 아래의 예에서 보듯이 "BL SUBROUTINE"은 "T1"의 주소를 가르키는 "PC"의 값을 "LR"에 저장한 후, "SUBROUTINE" 레이블이 있는 위치로 분기합니다. 해당 서브루틴의 동작을 마친 후, 마지막에 있는 리턴 명령어 "MOV PC, LR"을 만나면 저장되어있던 리턴 주소를 "PC"로 복원하면서 "T1"으로 분기해 동작을 계속합니다.

```
SUBROUTINE
    ...
    MOV PC, LR
    ...
    BL SUBROUTINE
    ...
T1    ...
```

4.4.4 Supervisor Call

- 서브루틴의 필요성


서브루틴은 하나의 프로그램 안에서 재사용되는 코드들을 루틴으로 만들어 놓은 것입니다. 재사용율이 높을수록 효율적인 프로그램을 만들 수 있습니다.

- 소프트웨어 인터럽트에 의한 서비스 루틴

하나의 운영체제 하에서 여러 개의 프로그램이 동작하는 경우, 여러 프로그램에게서 재사용되는 코드들은 어떻게 작성해야 할까요? 운영체제와 프로그램에게서 재사용되는 코드들은 어떻게 작성해야 할까요? 이러한 경우에는 서브루틴의 작성으로는 해결할 수 없으므로, 각각의 루틴을 인터럽트 서비스 루틴으로 작성합니다. 작성된 루틴이 필요할 때, 이를 소프트웨어 인터럽트 형태로 호출하면, 위의 경우에 재사용이 가능합니다.

- 소프트웨어 인터럽트를 발생하는 명령어는 "SWI"입니다.

- SWI <service_routine_id>

 **Tip**

SWI

< Syntax >

SWI<cond> <immed_24>

< Operation>

if ConditionPassed(cond) then

R14_svc = "SWI" 명령어 다음 명령어의 주소

SPSR_svc = CPSR

CPSR[4:0] = 0b10011

CPSR[5] = 0

CPSR[7] = 1

if high vectors configured then

PC = 0xFFFFF0008

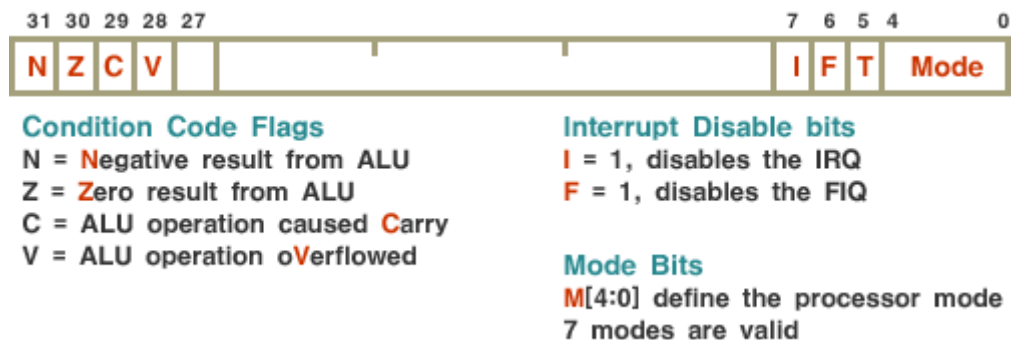
```
else
PC = 0x00000008
```

<Description>

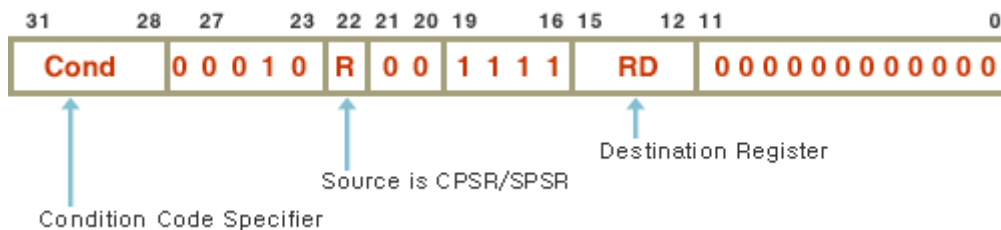
소프트웨어 인터럽트를 발생하는데 사용하는 명령어입니다. 실행이 되면, SWI exception이 발생합니다.

4.5 기타 명령어

● 일반용도의 레지스터인 r0~r15이외의 레지스터인 PSR(Program Status Register)를 읽거나 쓰기 위해서는 특별한 명령어를 사용하여야 합니다.



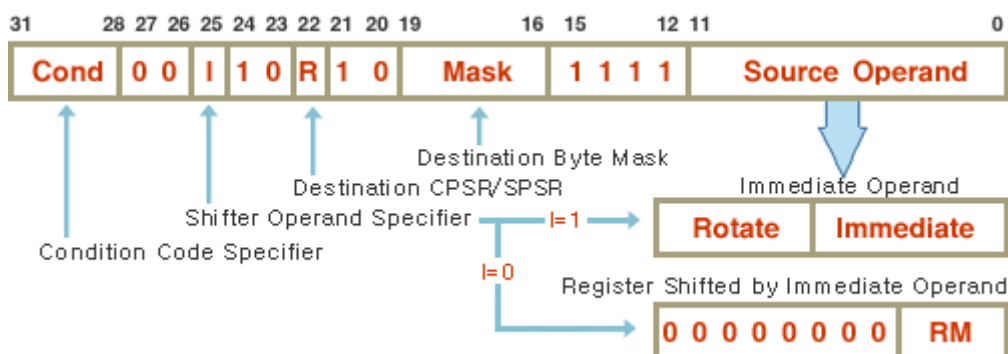
● MSR/MRS 명령어



● MRS reg1, CPSR/SPSR => reg1:=CPSR or SPSR :

CPSR(Current Program Status Register) 혹은 SPSR(Saved Program Status Register)의 내용을 일반용도의 레지스터(r0~r15)로 읽어 오는데 사용합니다.

단, 현재 프로세서 모드가 User 혹은 System 모드인 경우에는 SPSR 레지스터가 존재하지 않으므로 "MRS" 명령어를 사용하면 예측할 수 없는 더미 데이터가 읽혀집니다..





Tip

MRS(Move PSR to General-purpose Register)

<Syntax >

MRS<cond> <Rd>, CPSR

MRS<cond> <Rd>, SPSR

<Operation>

if ConditionPassed(cond) then

if R == 1 then

Rd = SPSR

else i

Rd = CPSR

< Description>

"MOV" 명령어는 ARM 코어 레지스터간의 데이터 이동시 사용하는 명령어입니다. 단, R0-R15의 레지스터만을 액세스할 수 있습니다. CPSR 혹은 SPSR 레지스터를 액세스하기 위해서는 특별한 명령어를 사용해야 하는 "MRS" 명령어는 그중 하나입니다. 이 명령어는 CPSR, SPSR로부터 데이터를 읽어오는데 사용합니다.

• MSR CPSR/SPSR, reg1/Immediate Data => CPSR/SPSR:=reg1/Immediate Data :

MSR(Move to Status register from ARM Register) 명령어는 "MRS" 명령어와는 반대로 특정 데이터를 CPSR/SPSR 레지스터에 저장하는데 사용합니다.

N/Z/C/V 컨디션 플래그를 변경하거나, 프로세서 모드를 변경하거나, IRQ/FIQ의 동작여부를 변경할 때 사용됩니다. CPSR/SPSR의 모든 비트를 변경하지 않고, 특정영역 만을 변경하고자 할 때에는, 특정 마스크 비트를 이용합니다.



Tip

MSR(Move to Status Register from ARM Register)

<Syntax >

MSR<cond> CPSR_<fields>, #<immediate>

MSR<cond> CPSR_<fields>, <Rm>

MSR<cond> SPSR_<fields>, #<immediate>

MSR<cond> SPSR_<fields>, <Rm>

< Operation>

if ConditionPassed(cond) then

if opcode[25] == 1

operand = 8_bit_immediate Rotate_Right (rotate_imm * 2)

else /* opcode[25] == 0 */

operand Rm

if R == 0 then

if field_mask[0] == 1 and InAPrivilegedMode() then

CPSR[7:0] = operand[7:0]

if field_mask[1] == 1 and InAPrivilegedMode() then

CPSR[15:8] = operand[15:8]

```

if field_mask[2] == 1 and InAPrivilegedMode() then
    CPSR[23:16] = operand[23:16]
if field_mask[3] == 1 then
    CPSR[31:24] = operand[31:24]
else /* R == 0 */
if field_mask[0] == 1 and CurrentModeHasSPSR() then
    CPSR[7:0] = operand[7:0]
if field_mask[1] == 1 and CurrentModeHasSPSR() then
    CPSR[15:8] = operand[15:8]
if field_mask[2] == 1 and CurrentModeHasSPSR() then
    CPSR[23:16] = operand[23:16]
if field_mask[3] == 1 then
    CPSR[31:24] = operand[31:24]
V Flag = OverflowFrom(Rn + shifter_operand)

```

< Description >

"MRS" 명령어와 데이터 이동의 방향이 반대입니다. ARM 코어 레지스터의 데이터 혹은 상수 데이터를 CPSR, SPSR 레지스터에 저장하는데 사용합니다.

- `_c` : sets the control field mask bit(bit 0)
: CPSR/SPSR 레지스터의 [0-7]bit만을 변경한다.
- `_x` : sets the extention field mask bit(bit 1)
: CPSR/SPSR 레지스터의 [8-15]bit만을 변경한다.
- `_s` : sets the status field mask bit(bit 2)
: CPSR/SPSR 레지스터의 [16-23]bit만을 변경한다.
- `_f` : sets the flags field mask bit(bit 3)
: CPSR/SPSR 레지스터의 [24-31]bit만을 변경한다.

▣ 심화학습

CPSR 혹은 SPSR 레지스터의 특정 플래그만을 변경하고자 할 경우 고려해 봅시다.


Tip

- `MSR CPSR, reg ; CPSR[31:0]:=reg[31:0]`
- `MSR CPSR_all, reg ; CPSR[31:28]:=reg[31:28]`
- `MSR CPSR_flg ; CPSR[31:28]:=reg[31:28]`
- `MSR CPSR_c ; CPSR[7:0]:=reg[7:0]`
- `MSR CPSR_x ; CPSR[15:8]:=reg[15:8]`
- `MSR CPSR_s ; CPSR[23:16]:=reg[23:16]`
- `MSR CPSR_f ; CPSR[31:24]:=reg[31:24]`
- `MSR CPSR_cxsf ; CPSR[31:0]:=reg[31:0]`

- Coprocessor를 ARM 프로세서에 추가했을 경우, 코프로세서,MMU 등에 데이터를 읽거나 저장할 때 사용되는 특별한 명령어가 MCR/MRC등 입니다.
코프로세서는 총 16개까지 지정할 수 있습니다.

- **LDC(Load Coprocessor Register)**

: 특정 어드레스 공간에 저장되어 있는 데이터를 코프로세서의 레지스터에 읽어오는 명령입니다.
아래의 예는 ARM 레지스터 R2가 가르치는 주소공간(정확히는 R2+ 4)에서 데이터를 읽어 코프로세서 6번(p6)안에 있는 레지스터 CR4(CP reg 4)에 로드하는 명령입니다.
example) LDC p6, CR4, [R2, #4]

 **Tip**

LDC

<Syntax >

```
LDC{<cond>}{L} <coproc>, <CRd>, <addressing_mode>
LDC2{L} <coproc>, <CRd>, <addressing_mode>
```

<Operation>

```
if ConditionPassed(cond) then
    address = start_address
    load Memory[address,4] for Coprocessor[cp_num]
    while (NotFinished(Coprocessor[cp_num]))
address = address + 4
assert address == end address
```

<Description>

외부 메모리와 ARM 코어 내부의 레지스터 사이에서 데이터 전송이 일어나는 경우, "LDR", "STR" 명령어를 사용합니다. 이와 동일하게 외부 메모리와 코프로세서 레지스터 사이에서 데이터 전송을 일어나는 경우, 특히 데이터를 읽어오는 경우 사용되는 명령어가 "LDC"입니다.

- **STC(Store Coprocessor Register)**

: 코프로세서의 레지스터에 저장된 데이터를 특정 어드레스 공간에 저장하는 명령입니다.
아래의 예는 코프로세서 8번(p8)안에 있는 레지스터 CR9(CP reg 9)의 데이터를 ARM 레지스터 R2가 가르치는 주소공간(정확히는 R2-16)에 저장하라는 명령입니다.
example) STC p8, CR9, [R2], #-16

 **Tip**

STC

<Syntax >

```
STC{<cond>}{L} <coproc>, <CRd>, <addressing_mode>
STC2{L} <coproc>, <CRd>, <addressing_mode>
```

< Operation>

```
if ConditionPassed(cond) then
    address = start_address
    Memory[address,4] = value from Coprocessor[cp_num]
```



```

while (NotFinished(coprocessor[cp_num]))
address = address + 4
Memory[address,4] = value from Coprocessor[cp_num]
assert address == end address

```


< Description >

"LDC" 명령어와 데이터의 이동방향이 반대인 코프로세서 명령어이다. "STR"명령과 유사하나 코프로세서용이라고 이해하면 편리하다. 코프로세서 내부 레지스터의 내용을 외부 메모리에 저장하는데 사용한다.

- [MCR\(Move to Coprocessor from ARM Register\)](#)

: ARM 레지스터에 저장된 데이터를 코프로세서의 레지스터로 복사할 때 사용합니다.

위의 LDC/STC는 코프로세서를 위한 LDR/STR 명령어이고, 이 명령어는 코프로세서를 위한 특별한 MOV 명령어로 이해하시면 됩니다.

 **Tip**

MCR

<Syntax >

```

MCR{<cond>} <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>
{, <opcode_2>}

```

```

MCR2 <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>
{, <opcode_2>}

```

<Operation >

```

if ConditionPassed(cond) then
    send Rd value to Coprocessor[cp_num]

```


<Description >

ARM 코어 레지스터의 내용을 코프로세서로 이동하는데 사용되는 명령어이다. 외부 메모리로부터 코프로세서 레지스터와의 전송을 위해서는 "LDC", "STC" 명령어를 사용합니다. 해당 코프로세서가 없을 경우에는 "undefined instruction exception"이 발생합니다.

- [MRC\(Move to ARM Register from Coprocessor\)](#)

: 코프로세서의 레지스터에 저장된 데이터를 ARM 레지스터에 복사할 때 사용합니다.

MCR과 데이터의 이동방향이 반대입니다.

 **Tip**

MRC

<Syntax >

```

MRC{<cond>} <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>
{, <opcode_2>}

```

```

MCR2 <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>
{, <opcode_2>}

```

<Operation>

```
if ConditionPassed(cond) then
    data = value from Coprocessor[cp_num]
    if Rd is R15 then
        N flag = data[31]
        Z flag = data[30]
        C flag = data[29]
        V flag = data[28]
    else /* Rd is not R15 */
        Rd = data
```

<Description>

코프로세서 레지스터로부터 ARM 코어 레지스터로 데이터를 이동하는 명령어입니다. "MCR" 명령어와 데이터 이동의 방향이 반대입니다.

- **CDP(Coprocessor Data Operation)**

: 코프로세서에게 데이터 처리를 수행하도록 하는 명령어입니다.

아래의 예는 코프로세서 5번(p5)에게 데이터 처리를 수행하도록 하는 명령어인데, 데이터 처리시 입력 데이터를 가지고 있는 레지스터는 c10과 c3이고, 그 결과를 저장하는 레지스터는 c12입니다.

코프로세서 데이터 처리를 위한 OPCODE는 각각 OPCODE1=2, OPCODE2=4입니다. 이는 코프로세서에 따라 결정됩니다.

example) CDP p5, 2, c12, c10, c3, 4



Tip

CDP

<Syntax >

CDP{<cond>} <coproc>, <opcode_1>, <CRd>, <CRn>, <CRm>, <opcode_2>

<Operation>

```
if ConditionPassed(cond) then
    Coprocessor[cp_num]-dependent operation
```

<Description>

ARM 레지스터 혹은 메모리와 독립적으로 코프로세서에게 특정 작업을 지시하는 명령어이다. 코프로세서 번호와 작업을 설정해서 실행시킨다.

5장 Psedo Instruction



개 요

5장에서는 ARM 의사명령어(pseudo instruction)와 매크로, 조건 컴파일 및 변수에 대하여 살펴봅니다.

이를 통하여 프로그램 작성 시 다양한 조건을 통해 개발의 편리성을 도모합니다.

학습목표

의사 명령어, 매크로, 조건 컴파일의 이해능력을 배양합니다.

학습내용

1. Pseudo Instruction
2. Directive
3. Data Definition
4. Macro
5. Symbol Definition

5.1 Pseudo Instruction

- 의사명령어는 말 그대로 실제 명령어가 아닙니다. 의사명령어를 사용하여 프로그램을 작성하면, 해당하는 적절한 실제 명령어로 대체되어 컴파일됩니다.
- 의사명령어는 아래의 표에서 보는 바와 같이 ARM/THUMB 명령어에 따라 약간의 차이를 가집니다. 하나씩 살펴보도록 합니다.

ARM Instruction	Thumb Instruction
ADR/ADRL	ADR
LDFD/LDFS	LDR
LDR	NOP
NOP	MOV

● ADR

ADR{condition} register, expression

: ADR 의사명령어를 사용하면, 분기하는 주소가 그 이전인지 이후인지에 따라 컴파일시에 ADD 혹은 SUB 명령어로 대체됩니다. 현재 PC 값을 기준으로 변화할 수 있는 주소의 범위는 아래와 같습니다. 이보다 더 멀리 위치하는 곳으로 분기할 때에는 ADRL 명령어를 사용해야 합니다.

: 오프셋 어드레스의 값은 파이프라인을 고려하여 결정해야 합니다.

: non word-aligned address (= < 255 bytes)

: word-aligned address (= < 1020 bytes)

: example

```
start  MOV      r0,    #10
        ADR      r4,    start          ; SUB r4, pc, #0x0c로 대체된다.
```

: example

```
start  MOV      r0,    #10
        ADR      r4,    test          ; ADD r4, pc, #0x04로 대체된다.
        MOV      r1,    #0
        MOV      r2,    #0
test
```

● ADRL

ADRL{condition} register, expression

: ADR 명령어와 동일하나 오프셋 어드레스의 범위가 다릅니다.

ADD나 SUB명령어가 2개씩 사용하여 오프셋 어드레스 범위를 늘렸습니다.

: non word-aligned address(=<64KB)

: word-aligned address(=<256KB)

● LDR

LDR{condition} register, expression / label-expression

: 아래의 2가지 경우에 있어, LDR 명령어는 의사명령어에 해당합니다.

1) 32비트 상수를 로드하는 경우

2) 어드레스를 로드하는 경우

: 레퍼런스 매뉴얼에는 MOV,MVN or LDR로 대체된다고 나오지만, 실제로는 항상 LDR 명령어로 대체되며 2 word(64bit)가 됩니다.

: example

```
        LDR      r0,      =0x12345678
        LDR      r0,      =test
```

test

● NOP

NOP ; MOV r0, r0

: NOP는 아무런 기능도 하지 않는 의사명령어입니다. 실제 이런 명령어가 존재하지 않으며, 위와 같이 MOV 명령어로 대체됩니다.

● LDFD

LDFD{condition} fp-register, =expression

: 플로팅 상수값을 레지스터에 읽어 올 때 사용하는 의사명령어입니다.

: 상수값의 범위는 아래와 같으며, 상수값과 PC와의 차이는 4KB 미만이어야 합니다.

: Max 1.79769313486231571e+308

: Min 2.225073858507201382-308

- MOV

: THUMB 명령어를 사용할 경우에만 발생하는 의사명령어입니다.

: 하위 레지스터의 값을 다른 하위 레지스터로 이동하고자 할 때, 이에 해당하는 실제 명령어가 존재하지 않으며 이 경우 ADD 명령어로 대체됩니다.

: example

MOV Rd, Rs ADD Rd, Rs, #0

5.2 Directives

- AREA

어셈블 해야 할 새로운 코드 영역이나 데이터 영역을, 어셈블러에게 알려 줄 때 이용됩니다. AREA 는 링커가 다루는 최소 단위이므로, 비슷한 속성을 가진 area끼리 링커를 사용하여 묶을 수 있습니다. 예를 들어, Example 이름의 read-only code 영역을 다음과 같이 표현할 수 있습니다.

AREA Example, CODE, READONLY

그러면, 이 이후에 쓰여지는 코드들은 ROM 안의 데이터 영역 안에 있음을 의미합니다.

[형식]

AREA name, attr, attr, ...

· name : 주어진 영역의 이름을 의미한다. 이는 프로그래머가 임의로 정할 수 있습니다.

단, 한 글자로 시작하는 이름은 bar(|)와 함께 사용해야합니다.

예를 들면, |1_ DataArea|, |C\$\$code|, ...

- AREA 지시어는 최소한 한번은 사용하여야 합니다.

· {attr} : AREA의 특징을 규정한다. 여기에 올 수 있는 내용들은 다음과 같습니다.

- ALIGN=expression

AOF 영역은 기본적으로 4-byte 영역이 할당되어 있습니다.

expression에는 2에서 31 사이의 정수 값이 올 수 있기 때문에, 이 영역으로 byte 만큼을 할당할 수 있습니다. 예를 들어 expression이 10이라면, 이 영역에는 1KB가 할당됩니다.

- CODE

여기에는 기계 명령어가 포함되어 있는데, 기본은 READONLY입니다.

- COMDEF

영역의 내용을 정의합니다. 이 영역에는 코드나 데이터가 포함될 수 있습니다.

- COMMON

일반적인 데이터 영역을 말합니다.

- DATA

명령어가 아닌 데이터를 포함합니다. 기본은 READWRITE입니다.

● INTERWORK

코드 영역이 ARM 영역인지 Thumb 영역인지를 알려줍니다.

● CODE16

이 지시어는 다음에 오는 명령어를 16-bit Thumb 명령어로 해석하라는 의미를 컴파일러에게 전달하고자 할 때 사용됩니다. 따라서 이 지시어를 사용한다고 해서 모드가 전환되지는 않습니다. 모드 전환은 BX 명령어를 사용하여야 하며, Thumb 상태로 분기할 때 이 지시어를 함께 사용합니다. 즉, 이 지시어는 어셈블리어에서 ARM 명령어와 Thumb 명령어를 혼합하여 사용하기 위해 존재합니다.

다음의 예는 ARM 명령어에서 Thumb 명령어로 분기할 때, CODE16이 어떻게 사용되고 있는지를 보여줍니다.

예)

AREA ThumbEX, CODE, READONLY

; 현재는 ARM 명령어이다.

ADR r0, start+ 1

BX r0 ; 분기 후 명령어 셋을 전환한다.

CODE16 ; 아래의 명령어들은 Thumb 명령어이다.

start MOV r1, #10

● CODE32

이 지시어는 다음에 오는 명령어들을 32-bit ARM 명령어로 해석하라는 의미를 컴파일러에게 전달하고자 할 때 사용합니다. Thumb 명령어를 사용하다가 ARM 명령어를 사용하고 싶을 때 BX 명령어와 함께 사용합니다.

예)

CODE16

: 앞으로는 Thumb 명령어가 사용될 것이다.

AREA ThumbEX, CODE

MOV r1, #10 : 이것은 Thumb 명령어!

ADR r0, goARM

BX r0 : 분기 후 명령어 셋을 전환한다.

CODE32 : 다음에 오는 명령어는 ARM 명령어이다.

goARM

MOV r4, #15 : 이것은 ARM 명령어!

● END

: 이 지시어는 소스 파일의 끝을 의미합니다.

모든 어셈블리어는 반드시 이 지시어로 끝을 맺어야 합니다.

● ENTRY

: 이 지시어는 어떤 프로그램이 이 지점에서부터 최초에 시작된다는 것을 의미합니다.

이 지시어는 프로그램 내에서 단 한번만 사용되어야 한다.

예)

AREA ARMEX, CODE, READONLY

ENTRY

; 프로그램 시작위치!

5.3 Data Definition

● ALIGN

: ALIGN 지시어는 현재 위치를 4바이트 범위안에서 정렬시킵니다. ARM 모드인 경우에는 "word align"을 THUMB 모드인 경우에는 "half-word align"을 수행합니다.

: ARM 940T와 같이 캐시가 4-word 단위로 동작할 경우에는 16바이트 범위에서 정렬을 시킬 필요가 있습니다. 이 경우에는 "ALIGN 16"와 같이 작성할 수 있습니다.

● DCB/DCW/DCD

: 메모리의 위치에 1개 혹은 여러 개의 공간을 할당합니다.

: 8비트인 크기로 할당할 경우는 DCB

: 16비트인 크기로 할당할 경우는 DCW

: 32비트인 크기로 할당할 경우는 DCD

: example

DCD 1, 5, 20 ; 10진수 데이터가 32비트 단위로 할당된 메모리 공간에 저장됩니다.

DCD mem06 ; mem06 레이블의 주소가 저장됩니다.

● %

: 메모리 공간에 0의 값을 갖는 공간을 확보합니다.

: example

AREA MyData, DATA, READWRITE

data1 % 255

; 0의 값을 갖는 255바이트 공간을 확보합니다.

●

: 해당하는 어드레스부터 일정 영역의 공간을 확보합니다. 하지만 실제로 공간이 할당되지는 않으므로 주의하셔야 합니다. '^'와 함께 사용됩니다. %와는 달리 0의 값으로 초기화되지 않습니다.

● ^

: #으로 확보한 메모리 공간을 특정시작주소에 배치합니다. 예를 들어 설명합니다.

: example 1)

^ 0x100

DCD 1, 5, 20

위와 같은 경우 데이터가 저장되는 시작주소가 0x100부터 시작한다고 오해하기 쉬운데, 절대 그렇지 않으므로 주의하여야 합니다. 이 경우 DCD는 0x100번지에 놓이지 않고, linker가 ^ 명령어는 무시하고 DCD의 데이터를 정해진 위치에 놓습니다. 앞에서 설명한 것처럼 ^의 번지에 영향을 받는 것은 #으로 정의하는 메모리 공간뿐 입니다.

: example 2)

`^ 0x100`

`LABEL1 # 0x100`

`LABEL2 # 0x200`

`LABEL3 # 0x100`

위와 같은 경우 LABEL1은 0x100, LABEL2는 0x200, LABEL3는 0x400의 번지값을 가지게 됩니다. 하지만, LABEL1,2,3에 실제로 메모리 공간이 확보되어진 것은 아니라 단지 LABEL1,2,3의 값만 정해진 것이라는 사실에 유의하여야 합니다.

5.4 Macro

● 매크로는 빈번하게 사용되는 내용을 만들어서 반복적으로 사용할 수 있습니다. 서브루틴과 달리 매크로는 매번 수행될 때마다 해당 코드의 내용이 수행되는 위치에 삽입됩니다.

● 매크로는 아래와 같이 2개의 지시어로 구성됩니다. "MACRO"는 매크로의 시작을 의미하고 "MEND"는 매크로의 끝을 나타냅니다.

```
MACRO
macro_prototype
; code
MEND
```

● 매크로는 레이블과 파라미터들을 가질 수 있습니다.

`{ $ label } macroname { $ parameter1 { , $ parameter2 } ... }`

: 아래의 예는 exception 처리를 위한 매크로 예입니다. "\$HandlerLabel"은 "HandlerFIQ" 라는 레이블로 사용되고 있으며, "\$HandleLabel"은 파라미터로 "HandleFIQ" 주소를 가르킵니다.

```
MACRO
$HandlerLabel HANDLER $HandleLabel
$HandlerLabel
...
MEND
```

HandlerFIQ HANDLER HandleFIQ

5.5 Symbol Definition

● GET/INCULDE

: 이 지시어는 어셈블 파일 안에 또 다른 어셈블 파일을 삽입할 때 사용합니다.

● GBL/ A/S

: 전역변수로 각각을 선언합니다.

각 접미사는 Logical/Arithmetic/String 변수들을 의미합니다.

● SETL/A/S

: 지역변수 혹은 전역변수의 값을 설정하는데 사용합니다. 각 접미사는 GBLL/A/S와 동일합니다.

: example(GBLL, SETL)

GBLL Debug
Debug SETL {TRUE}

: example(GBLA, SETA)

GBLA VersionNumber
VersionNumber SETA 21

: example(GBLS, SETS)

GBLS VersionString

: 이렇게 선언한 변수는 IF문을 사용하여 조건컴파일시 사용하거나 다음과 같이 \$를 앞에 붙여 많이 사용되고 있습니다.

pVersion DCB \$VersionString

● IMPORT/EXPORT

: "IMPORT"는 현재 어셈블 파일에 선언되지 않은 이름을 어셈블러에 제공합니다.
제공되는 이름은 다른 어셈블 파일에서 "EXPORT"로 선언되어 있어야 합니다.

● [, | ,]

: 조건에 따라 선택적으로 컴파일이 될 수 있도록 하기 위해 어셈블 파일에서 사용됩니다.

: "[" 혹은 "IF" 지시어를 사용할 수 있습니다.

: "|" 혹은 "ELSE" 지시어를 사용할 수 있습니다.

: "]" 혹은 "ENDIF" 지시어를 사용할 수 있습니다.

: example(GBLL, SETL)

```
[ Version = "1.0" ; IF ...  
; code and/or  
; directives  
| ; ELSE  
; code and/or  
; directives  
] ; ENDIF
```