ARM 프로세서 집중 분석

특집 2부에서는 ARM을 마냥 어렵게만 생각하는 독자에게 ARM의 과거와 현재에 대해서 살펴보며 ARM7TDMI를 중심으로 ARM 아키텍처의 전반적인 내용을 소개하고, ARM 명령어를 사용하는데 필요한 최소한의 지식과 ARM의 MMU에 대한 기본 개념들을 다룬다.

김봉주 | ghostpl@chollian.net

ARM 프로세서를 2000년 5월에 처음 접하게 되었다. 개발 도구는 이미 구매가 되어 있고 사람만 가서 개발하면 되는 데, 개발자인 필자는 ARM에 대해서 전혀 몰랐다. 사람이 죽으라는 법은 없는 것 같다. 저자는 수많은 자료를 뒤져가며 하드웨어의 신뢰성을 확보해야 했으며, 그 와중에 결국 6개월 동안 하나의 하드웨어 플랫폼을 가지고 두 개의 프로젝트를 동시에 수행하고 그 해 11월에 ARM을 적용한 프로토타입 제품을 출시할 수 있었다. 그것이 저자와 ARM과 맺은 첫 번째 인연이었다.

필자는 ARM 프로세서를 이용한 프로젝트를 처음 진행할 당시에 농담반 진담반으로 이런 말을 자주하곤 했다. "ARM으로 개발하려다가 정말 암에 걸리겠다." 처음 새로운 프로세서를 접하면 누구나 한번쯤 겪는 일이다. ARM의 경우는 그 정도가 더욱 심했다. 특히 "MMU는 무엇이며, 왜 필요하고 어떻게 동작하는지? FIQ와 IRQ는 같은 인터럽트인데 왜 달리 구분하는지? 왜 ENDIAN(엔디안)이라는 개념이 나오는 것인지? 도대체 ARM 모드와 THUMB 모드는 무엇이고 어떻게 다른 것인지? AMBA는 또 무엇인가?" 등 수많은 궁금증이 있었고, 그것을 풀기 위해서 밤낮으로 관련 자료를 찾기 위해 전전긍긍하며 웹 사이트를 돌아다니고, 그것도 모자라서 결국에는 주변에 책 동냥을 하러 달려가곤 했었다. 어쩌겠는가? 당면한 문제를 정면으로 헤쳐나가지 않으면 아무런 결과도 얻지 못할 것이 불을 보듯 뻔한데….

지금부터 필자와 ARM의 과거와 현재를 살펴본 후 ARM 아키텍처를 살펴보고, 마지막으로 ARM과 MIPS 중 어느 것을 선택할지 몰라 고민하는 독자들을 위해서 MIPS 프로세서에 대해서도 간단하게 살펴보기로 한다. 앞서 말해 두지만 이번 기사는 ARM의 내용이 주제이므로 MIPS에 대한 내용이 다소 부족하더라도 이해하기 바란다.

ARM의 과거와 현재

ARM의 탄생

최초의 ARM 프로토타입이 탄생한 것은 1985년 4월 26일 영국의 캠브리지에 있는 Acron Computers에 의해서였다. 그러나 정작 ARM(Advanced RISC Machine)이 빛을 바라기 시작하는 것은 1995년이 되어서이다. 특히 1990년 11월에 애플과 VLSI의 조인트 벤처 형식으로 설립된 ARM(Advanced RISC Machines Ltd.)이라는 회사가 생기면서 부터이다. 이 회사에 애플과 VLSI는 자본을 투자하고, 세계 최초의 상업용 원칩 RISC(Reduced Instruction Set Computer) 프로세서를 개발한 Acorn은 12명의 칩 개발자를 투입하게 된다. ARM은 최초의 저가형 RISC 아키텍처란 점만으로도 시장에서 차별화되었다. 1991년에 ARM6 프로세서 시리즈가 임베디드 RISC 프로세서로 처음 소개되면서, ARM이라는 이름도 공식화된 명칭으로 자리를 잡게 된다. 그 이듬해 샤프와 GEC Plessey가 라이선스 계약을 맺었으며, 1993년에는 TI와 사이러스 로직이 라이선스 계약하면서 ARM은 급성장을 시작하게 된다.

ARM 아키텍처의 계보

초기에 개발되어 아직까지 사용되고 있는 가장 오래된 ARMv4 아키텍처는 ARM7 코어 시리즈와 인텔 스트롱암(StrongArm) 프로세서에 아직도 그 영향을 미치고 있다. ARMv4 아키텍처는 32비트 주소 영역에서 32비트 ISA(Instruction Set Architecture) 동작이 가능하다. 16비트 Thumb 명령어 셋을 탑재한 ARMv4T 아키텍처는 32비트 코드의 이점을 그대로 살리고, 메모리 공간을 35% 이상 절약할 수 있도록 해주었다. 1999년에 ARM은 ARMv5TE 아키텍처를 소개하였으며, 개선된 thumb 아키텍처와 'Enhanced' DSP 명령어 셋을 ARM ISA에 추가하였다. 이러한 Thumb의 변화에는 소수의 명령어 추가와 함께 ARM/Thumb 인터워킹(interworking)의 개선, 컴파일 성능의 대폭적인 향상, ARM/Thumb 루틴의 혼합 사용, 코드 크기와 성능에 대한 균형도 포함되어 있다. 또한 'Enhanced' DSP 명령어들은 복잡한 수치연산에 있어 70%의 성능개선을 보여주었다.

2000년에 발표된 ARMv5TEJ 아키텍처에는 Jazelle(자바 하드웨어 가속기) 확장명령어가 추가되었으며, 이로써 자바 가속 기술을 탑재한 아키텍처가 탄생하게 된다. ARMv5TJE 아키텍처는 Jazelle의 탄생함에 따라 가속 기술을 사용하지 않은 JVM(Java Virtual Machine)보다 속도 면에서 8배가 향상되었으며, 소비전력의 측면에서도 80%를 줄일 수 있게 된다. 또 ARMv6 아키텍처가 2001년에 발표되면서 여러 방면에서 기능 개선이 이루어졌다. 특히 메모리 시스템, 예외 처리의 개선, 멀티프로세싱 환경을 위한 더 많은 지원 등이 이에 해당한다. 이것 이외도 ARMv6 아키텍처에는 SIMD(Single Instruction Multiple Data) 소프트웨어 실행을 지원하는 미디어 명령이 포함되어 있으며, SIMD 명령들은 오디오 및 비디오 코덱을 포함하는 응용 프로그램들의 사용 확대를 위해 최적화되었다.

2002년에는 ARMv6 아키텍처를 처음 적용한 제품인 ARM1136J(F)-8 코어를 발표하였다. 스트롱암 CPU는 DEC(Digital Equipment Coporation)에 의해서 ARM과 함께 개발되었다. 이것이 최초의 modified-Harvard 아키텍처(명령어 캐시와 데이터 캐시를 분리해서 사용)를 채용한 제품이며, modified-Harvard 아키텍처로 ARM의 쓰기 처리 능력의 고속화가 가능하게 되었다. 스트롱암의 주요 특징 중에는 5단 파이프라인의 채용, 64비트 곱셈 및 일부 곱셈 기능을 제외한 모든 일반 명령어들의 싱글 사이클 처리 등이 포함되어 있다.

우리에게 잘 알려진 인텔의 ARM 칩셋들에는 Xscale과 함께 SA11xx 시리즈의 제품들이 있다. 이중 SA11xx 시리즈는 오히려 스트롱암으로 더 잘 알려져 있기도 하다. 스트롱암 프로세서는 휴대용 전화기와 가전제품 등의 설계에 적합하며, 현재 컴팩의 iPAQ H3600 포켓 PC, HP의 조나다 핸드헬드 PC, 팜(Palm) PDA 및 많은 제품들에 사용되었다. 인텔이 개발한 SA110은 일반적인 표준 임베디드 프로세서이며, 팜 크기의 기기를 위해 개발된 SA1100와 SA1111 프로세서는 인텔이 사용하는 ARM 코어의 ASIC 패밀리의 공식 명칭이 되었다(SA는 StrongARM의 약어이다).

<표 1> ARM 인스트럭션 셋 아키텍처 테이블
(www.arm.com/products/CPUs/architecture.html 참조)

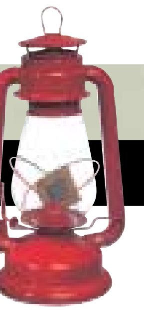
아키텍처	Thumb	DSP	Jazelle	미디어	TrustZone	Thumb-2
v4T	V					
v5TE	V	V				
v5TEJ	V	V	V			
v6	V	V	V	V		
v6Z	V	V	V	V	V	
v6T2	V	V	V	V		V

ARM의 CPU 코어들

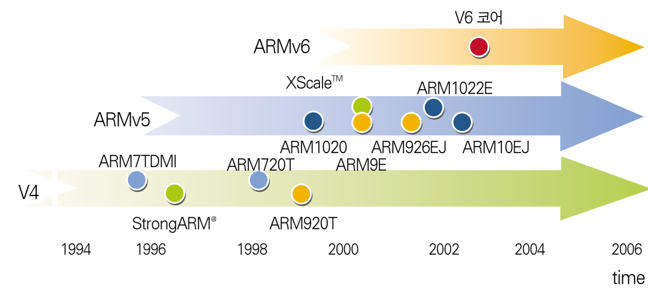
ARM CPU 코어는 애플리케이션 코어, 임베디드 코어, 그리고 시큐어 애플리케이션(Secure Application)으로 구분할 수 있다. 애플리케이션 코어는 무선, 가전제품, 이미지 처리 등의 복잡한 운영체제 처리에 사용할 수 있으며, 임베디드 코어는 대용량 저장장치, 자동화 기기, 산업용 기기 등의 실시간 처리 시스템을 위해 사용되며, 시큐어 애플리케이션은 보안 관련 제품의 제작에 적용이 가능하다.

ARM의 CPU 코어를 패밀리로 분류해 보면 ARM7, ARM9, ARM10, ARM11, 스트롱암과 Xscale이 있다. 각각의 패밀리에는 뒤에 붙는 코드에 따라서 코어의 사양이 조금씩 달라진다. 우리가 ARM 아키텍처에서 다루게 될 ARM7TDMI의 경우는 Thumb(16비트 압축 모드), Debug(디버그 기능), Multiplication(32비트 하드웨어 곱셈기), ICE(In Circuit Emulation) 등이 내장되어 있다. ARM7TDMI의 경우는 MMU(Memory Management Unit)와 AHB(Advanced High performance Bus)는 들어 있지 않다. ARM720T에는 8KB의 캐시와 함께 MMU와 AHB가 포함되어 있다. ARM720T를 채택한 임베디드 프로세서가 국내 회사에서도 이미 출시되었으며, 그 내용은 해당사의 웹 사이트에서 쉽게 찾아볼 수 있다.

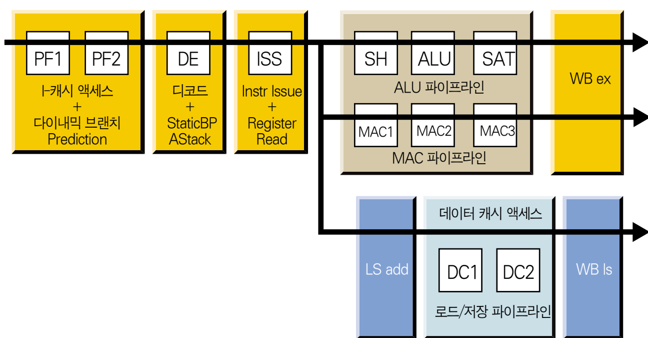
VFP(Virtual Floating Point) 코어 프로세서는 ARM 아키텍처의 선택 사양이다. VFP 아키텍처는 단정도/배정도 실수 연산을 지원하며, 안정적인 소프트웨어 라이브러리의 지원을 위해 IEEE754 규약을 충실히 따르고 있다. ARM VFP의 벡터 연산 능력은 축적 변환, 2차



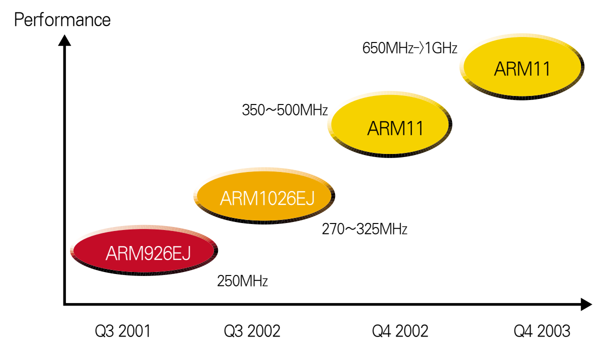
<그림 1> ARM 아키텍처 Revisions(The ARM Architecture Version 6 (ARMv6), David Brash, ARM Ltd. Figure 1 참조)



<그림 2> ARM11 파이프라인 구조(The ARM11 Microarchitecture, David Cormie저, ARM Ltd. Figure 3 참조)



<그림 3> ARM 프로세서 로드맵(The ARM11 Microarchitecture, David Cormie저, ARM Ltd., Figure 2. 참조)



원 및 3차원 변환, 폰트의 생성 및 디지털 필터와 같은 영상 처리의 성능 향상에 사용이 가능하다. ARM은 현재 ARM9, ARM10, ARM11 프로세서 패밀리들에 VFP9-S와 VFP10을 지원하고 있다.

ARM의 파이프라인 채용의 변화

ARM의 중요한 특징 중 하나가 파이프라인을 채용한 RISC 아키텍처라는 점이다. ARM7TDMI 같은 ARM7에서는 페치(fetch), 디코드(decode) 그리고 실행(execute)의 3단계로 구성된 파이프라인을 적용하고 있다. ARM9의 경우에는 페치, 디코드, 실행, 버퍼/데이터, 다시쓰기(write-back)의 5단계 구조의 파이프라인을 적용하고 있으며, ARM10의 경우는 6단계 구조의 파이프라인이 적용되어 있다. 이것도 모자라서 결국 ARM11의 경우는 무려 8단계 구조의 파이프라인을 채택하고 있다.

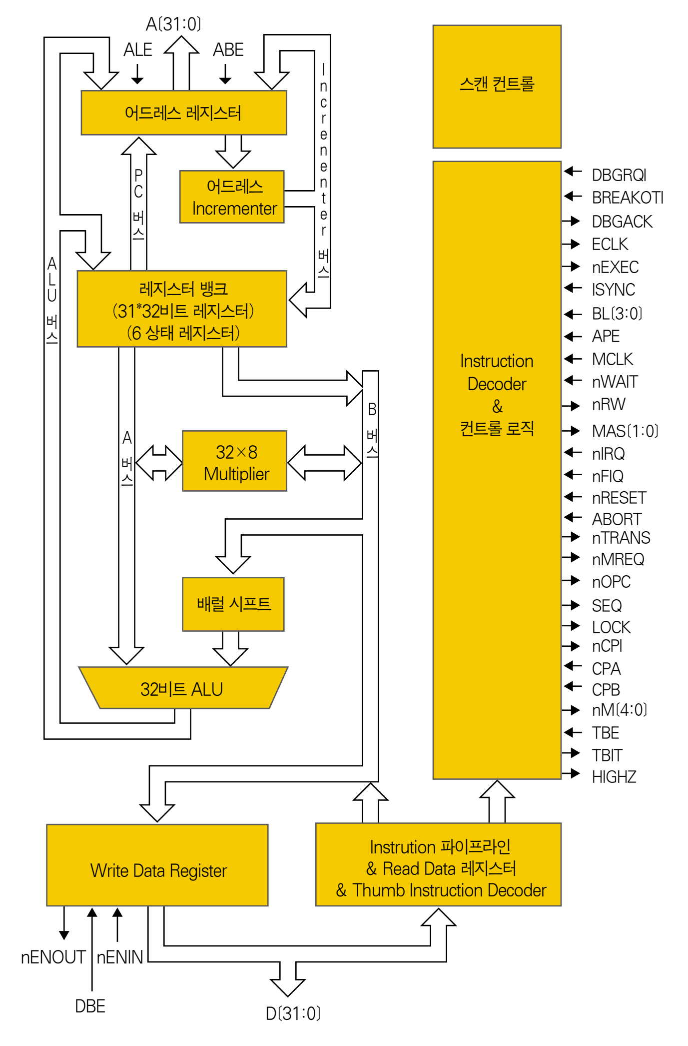
인텔 계열의 스트롱암은 5단계 구조의 파이프라인을, Xscale은 7단계 구조의 파이프라인을 채택하고 있다. 이렇게 많은 파이프라인 구조를 채택하게 된 데에는 가능한 한 프로그램의 코드 공간을 줄여서 데이터 처리 능력을 향상시켜 보고자 하는 ARM 프로세서 개발자들의 숨은 노력이 들어 있다. ARM11 마이크로 아키텍처는 고성능과 고효율을 실현하기 위해서 설계되었다. 그 해결의 열쇠가 바로 파이프라인의 설계이다. ARM11 마이크로 아키텍처의 파이프라인은 이전의 ARM 코어들과는 다르다.

8단계로 구성된 ARM11 마이크로 아키텍처의 파이프라인은 이전의 코어들보다 40% 더 향상된 성능을 제공한다. 8단계 파이프라인은 8개의 다른 프로세스 상태를 동시에 처리할 수 있도록 해준다. 많은 단계의 파이프라인 구조는 과도한 지연현상과 래이턴시에 의해 시스템의 효율을 저하할 수도 있다. 일반적으로 어떤 명령을 실행하는 긴 파이프라인들은 지연의 가능성이 있다는 것이며, 그 이유는 이전 명령어들의 처리결과에 따라 달라지기 때문이다. ARM11 파이프라인은 이러한 지연 현상들을 피하기 위해 파이프라인 간에 forwarding(일종의 pre-fetch 기능) 기법을 광범위하게 사용한다. 긴 파이프라인을 가진 프로세서는 파이프라인을 통해 순조롭게 처리되던 명령이 인터럽트가 발생한 순간, 성능의 저하가 올 수도 있다. ARM11의 파이프라인은 이 문제를 명령어들의 흐름을 예측하는 분기 예측(branch prediction)을 사용해서 해결하고 있다.

ARM11의 등장

<그림 3>의 로드맵에서 보는 바와 같이 ARM11의 마이크로 아키텍처는 350~500MHz 영역에서 제품이 개발되었으나, 1GHz 영역까지 그 성능의 범위를 확대하고 있다. 개발자는 소비전력과 성능을 제어하기 위해서 클럭 주파수와 인가 전압 모두를 변화시킬 수 있다. 일반적으로 빠른 클럭의 하드웨어는 높은 소비전력을 요구하게 된다. 이런 측면에서 본다면, 프로세서의 속도를 소프트웨어에 의해서 조절할 수 있도록 하드웨어가 지원한다면, 프로세서에서 사용되는 소비전력의 일부를 조절할 수 있다는 의미가 된다.

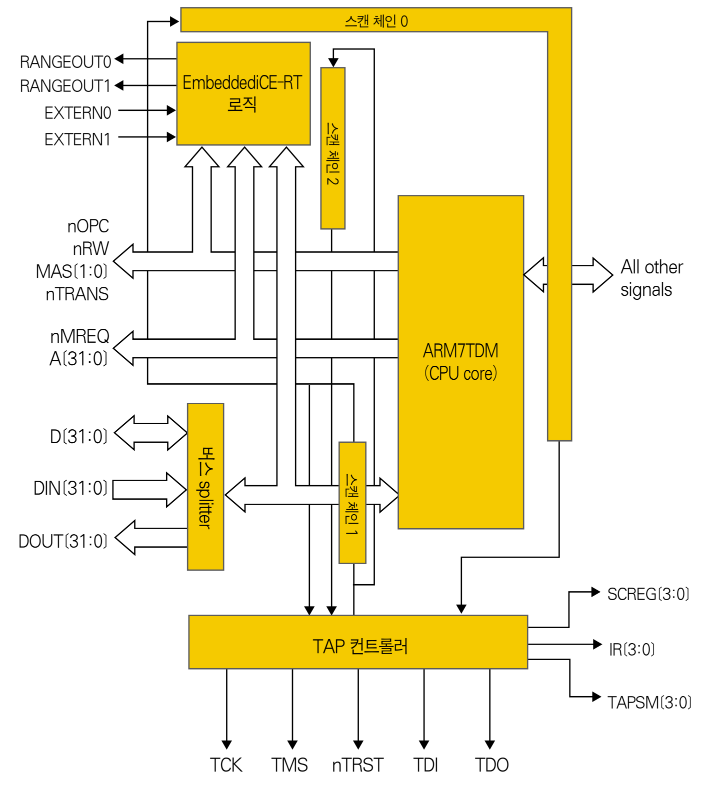
<그림 4> ARM7TDMI 코어 다이어그램(ARM7TDMI rev 4 Technical Reference Manual(ARM DDI 0210B), Figure 1-3 참조)



ARM 영역의 확장

초기의 ARM이라는 회사는 비메모리의 설계 및 생산을 담당하는 반도체 회사에 반도체 공정의 핵심 역할을 담당하는 IP(Intellectual Property)를 개발해서 제공하던 회사였다. ARM의 가장 큰 무기인 저전력 설계가 가능하다는 점은 ARM이 PDA를 비롯한 가전과 모바일 및 무선 시장까지도 장악하게 만들었다. 많은 임베디드 프로세서를 채택한 제품들이 저전력 설계에서 고전을 면치 못하던 시기에 ARM의 등장은 시장의 요구와 맞아 떨어진 것이다. 새로운 2차 전지들의 지속적인 성능 향상에도 불구하고 PDA 등과 같은 휴대용 단말기에서 저전력 기술은 필수이며, 이러한 방향은 앞으로도 지속될 것으로 보인다. 현재 ARM은 하드웨어 및 소프트웨어의 개발에 필요한

<그림 5> ARM7TDMI 프로세서 블럭 다이어그램(ARM7TDMI rev 4 Technical Reference Manual(ARM DDI 0210B), Figure 1-2 참조)



개발 툴들도 함께 제작하여 판매하고 있으며, 이러한 개발 툴들도 지속적으로 발전하고 있다. 단지 흠이라면 가격이 좀 비싸다는 것이다.

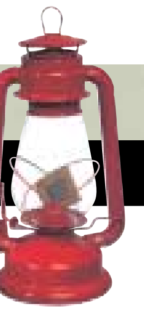
ARM 아키텍처 구조

앞서 살펴보았듯이 ARM 아키텍처는 계속해서 발전하고 있다. 그러나 ARM의 아키텍처에는 변하지 않는 몇 가지 규칙이 있으며, 그 규칙들을 이해하는 것이 ARM을 이해하는데 무엇보다도 도움이 될 것이다. ARM에 대한 자료는 비록 영문이기는 하지만 www.arm.com에서 쉽게 찾을 수 있으며, 설명 또한 매우 쉽게 잘 정리되어 있다.

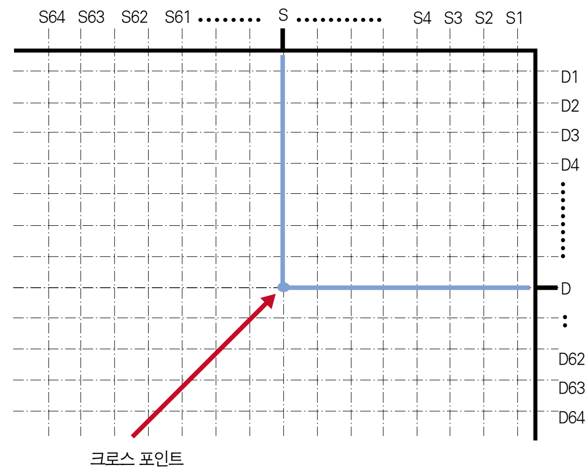
우리는 지금부터 ARM7TDMI의 아키텍처를 살펴볼 것이다. 물론 지금은 스트롱암과 같은 더 뛰어나고 더 발전된 아키텍처가 있지만, 처음 접하는 독자들을 위해서 ARM7TDMI 아키텍처를 위주로 살펴보고, 나머지는 그 위에 살을 붙이는 형식으로 살펴보기로 한다.

ARM7TDMI의 내부 구조

ARM7TDMI의 코어의 구조는 <그림 4>와 같다. 주요 구성을 요약해 보면 ARM7TDMI 프로세서는 32비트 크기의 명령어 셋을 포함하는



<그림 6> 32×32 크로스-포인트 매트릭스의 개념도



ARM state와 16비트 크기의 명령어 셋을 포함하는 Thume state를 지원한다. ARM state의 32비트 명령어들이 실행속도가 대체로 Thume state에 비해 빠르다. Thume state의 16비트 명령어들이 프로세서 내부에서 32비트로 디코딩되어 수행되어 속도가 늦는 반면, 실행 코드의 메모리 공간을 절약할 수 있다는 장점이 있다.

최근에는 Thume state의 이런 단점을 보완하기 위해서 Thume-2 state가 개발되었고, 자세한 내용은 본 기사의 ARM vs. Thume state에서 다루기로 한다. Power-on reset 후의 프로세서는 ARM state로 동작하므로 Thume state로의 진입에는 별도의 설정이 필요하다.

<그림 5>는 ARM7TDMI의 블럭도이다. 우리가 종종 사용하는 소프트웨어 디버깅 도구인 ICE(In Circuit Emulator)와의 인터페이스를 위해서 ICE Breaker와 TAP 컨트롤러가 코어 옆에서 시스템의 Arbitration을 담당한다. 특히 <그림 5>를 보면 스캔 체인 0과 스캔 체인 2가 있는 데, 이것은 반도체 공정에서 칩의 정상동작 여부를 테스트하기 위해서 필요한 데이터를 칩으로 보내게 되는 데, 이때 그 데이터가 지나가는 경로가 스캔 체인이며, 흔히 주문형 반도체(ASIC)에서 사용하는 테스트 벡터(Test Vector)보다 좀더 상위의 개념이라고만 알아두자.

파이프라인

ARM7은 저가 모델로 디자인되어서 3단계의 파이프라인만 지원하지만, ARM9부터는 RISC CPU에서 전통적으로 많이 쓰는 5 단계의 파이프라인을 지원한다. 3단 파이프라인의 각 단계별 수행되는 역할을 살펴보면 다음과 같다.

- **Fetch** : 메모리로부터 명령어를 읽어 와서 명령 파이프라인에 둔다.
- **Decode** : 명령어를 해독하고 데이터 패스 제어를 통해 다음 명령어를 준비한다.
- **Execute** : 데이터 패스 제어에 의해 레지스터 뱅크를 읽어오고 오퍼랜드를 시프트시키고 ALU의 결과를 목적 레지스터에 저장한다.

5단 파이프라인의 경우에는 3단 파이프라인에 비해 다른 점이 있다.

- **Fetch** : 메모리로부터 명령어를 읽어 와서 명령 파이프라인에 둔다.
- **Decode** : 명령어를 해독하고 데이터 패스 제어를 통해 다음 명령어를 준비한다.
- **Execute** : 명령어를 레지스터 뱅크에서 읽어 오고 ALU 결과를 생성한다.
- **Buffer/Data** : 필요에 따라 데이터 메모리에 접근을 시도하며 ALU 결과를 버퍼에 저장한다.
- **Write-Back** : 메모리에서 읽은 데이터와 ALU 결과를 레지스터에 써 넣는다.

바렐 시프트

ARM7TDMI 코어 다이어그램을 보면 ARM7TDMI 아키텍처는 ALU와 함께 한번에 여러 단의 시프트 연산을 수행할 수 있도록 되어 있다. 다른 프로세서의 아키텍처가 shifter와 ALU가 병렬로 연결되어 있어 dataphth의 사이클 타임에 영향을 주지 않도록 되어있는 반면, ARM7TDMI의 아키텍처는 여러분이 ARM7TDMI 코어 다이어그램에서 보았듯이 ALU와 직렬로 연결되어 있다. 이러한 ARM7TDMI의 아키텍처는 시프트 연산의 시간이 datapath의 사이클 타임에 직접적인 영향을 주게 된다. 시프트 연산에서의 이런 지연 시간을 최소화하기 위해서 'cross-bar switch matrix'를 사용하는데, 이는 임의의 입력 값에 대해서 원하는 결과 값을 가진 곳으로 출력 값의 방향을 변경하는 일종의 MUX(Multiplex)와 같은 역할을 한다. 즉, 매번 시프트에 대한 연산이 수행되는 것이 아니라, 시프트 연산의 결과값에 대한 경로가 이미 하드웨어를 이용한 매트릭스로 구성되어 있어 연산처리에 대한 시간을 단축시키고 있는 것이다. ARM7TDMI의 경우 32×32의 크로스-포인트 매트릭스를 사용한다. 결국 프로세서의 속도를 높이기 위한 각고의 노력을 한 흔적을 이곳에서도 엿볼 수가 있는 것이다.

레지스터의 구조

ARM7 프로세서에는 31개의 32비트 범용 레지스터와 6개의 32비트 상태 레지스터가 있다. 이들은 사용자 모드와 시스템 모드에 따라서 사용할 수 있는 레지스터가 달라지며, ARM7의 어셈블러에서 프로그래머가 한번에 사용 가능한 레지스터는 R0~R15가 전부이며, 그 중 일부는 프로그램 카운터(PC)나 스택 포인터(SP) 등의 용도로 사용되고, 일부 레지스터는 프로세서의 Exception 처리를 위해 R0~R15에 리맵핑되어 사용되기도 한다. ARM7 프로세서의 레지스터 구조는 <그림 13>을 참조하자.

범용 레지스터 : R0~R15

범용 레지스터 중 R13~R15는 특수 목적으로 사용되는데, 내용을 살펴보면 다음과 같다.

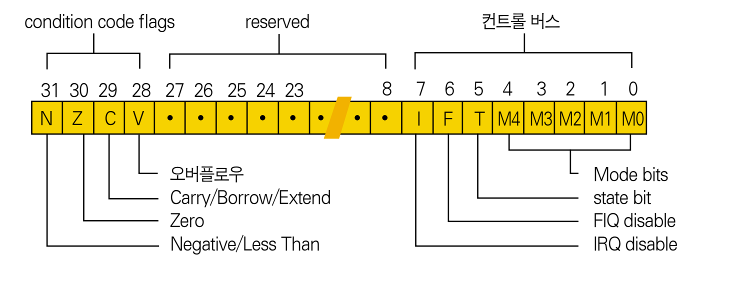
- **프로그램 카운터(R15)** : 우리가 일반적으로 알고 있는 PC(Program Counter)이다. 다만 다른 점은 ARM7에서는 R15 역시 사용자가 레지스터로 사용할 수 있다는 점이다. 즉, 어셈블러에서 일반 연산에 R15를 사용할 수 있으니, 좀더 프로그래머에게 자유도가 많다고 보아도 된다. 다만 그에 대한 부작용은 프로그래머의 몫이 되는 것이다.
- **스택 포인터(R13)** : ARM에서는 SP(Stack Pointer)로 R13을 사용한다. 스택과 관련된 PUSH나 POP 같은 명령어가 따로 제공되지 않는 대신 명령어 자체에 increment나 decrement 명령어 셋이 포함되어 있으므로 실제 동작에서는 별 문제가 되지 않는다.
- **링크 레지스터(R14)** : ARM에서는 다른 프로세서와는 좀 다른 개념인 링크 레지스터(link register)라는 것이 있으며, 이를 R14에 할당하고 있다. 앞서 언급했듯이 ARM에서는 스택과 관련된 명령어가 따로 제공되지 않으니 CALL과 RET와 같은 명령도 없다. 그 대신에 JUMP 명령을 수행하기 전에 돌아올 주소를 R14에 넣고 점프할 번지로 이동하며, 돌아올 주소로 복귀할 때는 R14에 저장된 값을 PC에 넣으면 된다. 이러한 이유로 R14를 링크 레지스터라고 부른다. 물론 이 방식이 전적으로 좋은 것은 아니다. CALL이 한번 만 일어난다면 이 방식은 PUSH나 POP과 같은 스택 연산을 수행하지 않으니 속도가 빠르겠지만, 호출된 프로그램 내에서 다시 호출을 하는 상황에서는 R14를 다른 곳에 보관해 두어야 한다. 또한 어셈블러에서 RET 명령에 익숙해 있던 사용자의 경우는 더 생소하게 느껴질 것이다. 해결책이 전혀 없는 것은 아니며, RET를 하나의 매크로로 만들어서 사용하는 것도 방법이 될 것이다.

ARM7 상태 레지스터

ARM7에는 6개의 상태 레지스터(Status Register)가 있다. 일반적으로 CPSR(Current Processor Status Register)이라고 하며, PSR이라고 나오더라도 같은 의미이니 당황하지 말기 바란다. <그림 7>은 CPSR에 대해서 설명하고 있다.

<그림 7>에서 보는 바와 같이 CPSR은 연산의 결과에 따라 달라지는 Flag 비트들과 프로세서의 흐름 제어를 위한 컨트롤 비트들로 나누어진다. Flag 비트들에는 연산의 결과가 마이너스인 경우에 1이 되는 N(Negative/Less Than) flag, 연산 결과가 0일 때 1이 되는

<그림 7> ARM 프로그램 상태 레지스터(ARM7TDMI rev 4 Technical Reference Manual(ARM DDI 0210B), Figure 2-6 참조)



Z(Zero) flag, 연산 결과에서 Carry나 Barrow가 발생할 때 1이 되는 C(Carry/Barrow/Extend) flag, 그리고 연산 결과에 오버플로우가 발생했을 때 1이 되는 V(Overflow) flag가 있다.

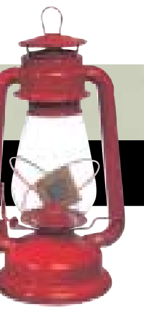
컨트롤 비트들에는 FIQ와 IRQ을 금지할 수 있는 FIQ disable 비트, IRQ disable 비트, 프로세서의 동작 상태를 나타내는 모드 비트(M0~M4)가 있다.

Exceptions과 Operation Modes

Exception이란 하드웨어의 입장에서 보면 실행되는 코드의 진행에 대한 예상하지 못한 상황에 대해서 예외처리의 기준을 마련하는 것을 의미한다. 소프트웨어를 설계하는 사람의 입장에서 설명한다면 NMI(Non-Maskable-Interrupt) 처리와 유사한 형태를 지닌다. 우리가 윈도우 운영체제에 Access Violation Error를 가끔 접하게 되는 데, 이것이 하드웨어의 입장에서 본 exception 처리에 대한 하나의 예가 된다. ARM은 이러한 예외 처리의 단계를 5가지로 구분하고 있으며 그 단계마다 적절한 대응을 위한 예외 처리 벡터를 설정해 놓고 있다.

구체적으로 ARM7에는 FIQ(Fast Interrupt reQuest)와 IRQ(Interrupt ReQuest), Abort, 소프트웨어 인터럽트, Undefined Instruction Trap의 5가지 Exception이 있고, 각각의 Exception이 발생하면 프로세서는 해당 동작 모드로 전환한다. 사용자 모드도 하나의 동작 모드로 보아 총 6개의 동작 모드가 존재하게 된다.

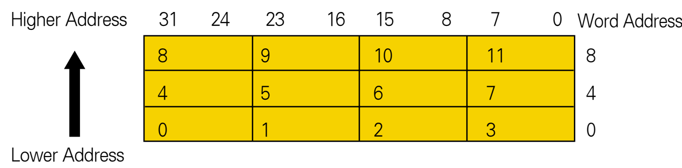
IRQ Exception은 우리가 흔히 인터럽트라고 말하는 개념과 동일하다고 보면 된다. FIQ Exception은 IRQ보다 좀더 빠른 처리를 할 수 있는 인터럽트이다. 물론 자원이 남는다면 IRQ에서 처리할 내용을 FIQ에서 처리해도 무방하다. Abort Exception은 쉽게 설명하면 메모리와 관련된 대부분의 비정상적인 동작이 발생하면, Abort Exception이 발생한다고 생각하면 된다. 예를 들면 프로세서가 메모리에서 접근하면서 발생하는 오류가 이에 해당한다. Abort은 다시 Pre-fetch Abort과 Data Abort으로 나누어진다. 소프트웨어 인터럽



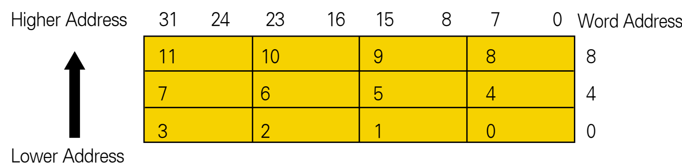
<표 2> Exception 벡터 테이블(ARM7TDMI Data Sheet(ARM DDI 0029E), Table 3-3 참조)

Address	Exception	Mode on entry
0×00000000	리셋	슈퍼바이저
0×00000004	Undefined instruction	Undefined
0×00000008	소프트웨어 인터럽트	슈퍼바이저
0×0000000C	Abort(prefetch)	Abort
0×00000010	Abort(데이터)	Abort
0×00000014	Reserved	Reserved
0×00000018	IRQ	IRQ
0×0000001C	FIQ	FIQ

<그림 8> Big endian addresses of bytes within words(ARM7TDMI Data Sheet(ARM DDI 0029E), Figure 3-1 참조)



<그림 9> Little endian addresses of bytes within words(ARM7TDMI Data Sheet(ARM DDI 0029E), Figure 3-2 참조)



트 Exception은 IRQ가 하드웨어에 의해서 발생하는 인터럽트라고 한다면 이것은 소프트웨어에서 수행되는 인터럽트라고 볼 수 있다. 엄밀히 말하면 조금 개념은 다르지만 우선은 그렇게만 알아두자. UIT(ndefined Instruction Trap)은 프로세서에 의해 자체적으로 해독이 불가능한 명령어를 만나면 발생하는데, 예를 들면 코어 프로세서의 명령은 UIT에 의해서 처리된다.

〈표 2〉 이외에도 리셋과 같은 Exception이 있는 데, 이는 Power On Reset시에 프로그램이 시작하는 부분이니 별도의 설명이 필요 없을 것 같다.

ARM에서는 동시에 여러 개의 Exception이 들어오는 경우를 대비하여 Exception의 우선순위를 지정하고 있으며, 우선순위는 Reset(1st) 〉 Data abort(2nd) 〉 FIQ(3rd) 〉 IRQ(4th) 〉 Prefetch abort(5th) 〉 SWI, undefined instruction(6th)의 순서가 된다. SWI가 포함된 6번째 우선순위는 동시에 두 가지가 발생하지 않도록 하드웨어적으로 구성되어 있다.

ARM7의 명령어 셋

ARM7 명령어의 주요 특징을 살펴보면, 첫째 ARM7에서 모든 명령어는 하나의 32비트 문자에 모두 들어가도록 설계되어 있다. 명령어의 수가 많지 않고 상대 어드레싱 모드를 사용한다. 이는 RISC 프로세서의 특성상 성능 면에서 상당한 이점을 가지게 된다. 물론 어셈블리 언어로 코드를 작성하게 되면 32비트 Immediate 값을 지정할 때는 다소 불편한 점이 없는 것은 아니다. 둘째, 모든 명령어가 같은 크기로 처리되므로 파이프라인의 구현이 용이하고, 이러한 구조는 결국 부가적인 명령어 해독기를 두지 않아도 되므로 명령어의 고속 처리가 가능하다. 셋째, 모든 명령어에 조건 코드의 사용이 가능하다. ARM에서의 가장 큰 강점이기도 하다. 넷째, C와 같은 고급 언어의 지원을 위한 명령어 셋을 내장하고 있다는 점이다. 예를 들면, C 언어에서 ++i, i++, --i, i--?와 같은 연산을 위한 명령어 셋이 별도로 마련되어 있다.

명령어와 관련된 기본 개념

ARM7의 명령어 셋은 기타 어셈블리 언어를 다루는 것과 크게 다르지 않다. 다만 다른 프로세서들에서 볼 수 없는 몇 가지 특징들이 있는데, 아키텍처와 관련된 기본 개념을 충분히 숙지하고 몇 가지 필요한 규칙들을 숙지한다면 독자 여러분 역시 어셈블리 언어를 그리 어렵지 않게 사용할 수가 있으리라고 본다. 명령어와 관련된 부분을 살펴보자.

◆ **메모리 구조(엔디안의 개념 설명)** : 리틀 엔디안(Little-Endian)과 빅 엔디안(Big-Endian)이라는 개념은 비단 ARM을 위해서 개발된 개념은 아니다. 다만 ARM에서는 좀더 엔디안에 대한 자유도가 있을 뿐이다. 리틀 엔디안이 최하위 바이트가 최하위 주소에 놓이고, 문자는 최하위 바이트의 바이트 주소에 놓이게 되는 반면, 빅 엔디안은 최상위 바이트가 최하위 주소에 놓이고, 문자는 최상위 바이트의 바이트 주소에 놓이게 된다. 다음의 〈그림 9〉와 〈그림 10〉을 보면 좀더 이해가 쉬울 것이다. ARM에서는 이를 컴파일시에 선택할 수 있도록 하고 있다.

더 자세히 알고 싶은 독자는 ARM의 공식 사이트에서 제공하는 Application Note 61 Big and Little Endian Byte Addressing (Document number: ARM DAI 0061A)을 참조하면 된다.

◆ **데이터 타입** : ARM7 명령어들은 모두 32비트 문자로 구성되며, 반드시 문자 단위로 정렬(word-aligned)되어야 한다. Thume 명령의 경우는 하프 문자로 구성되며, 2바이트 영역으로 정렬되어야 한다. 내부적으로는 모든 ARM 명령은 32비트 오퍼랜드가 되며, 32비트보다 작은 데이터들은 단지 데이터 전송 명령에서만 지원한다. 초기에 ARM에서 지원하는 데이터 타입에는 8-bit unsigned byte와 4바이트 바운더리로 정렬되는 32비트 문자로 제한되었지만, 이후에는 8-bit signed byte와 2바이트 바운더리로 정렬되는 16-bit signed와 unsigned half-word도 같이 지원하게 된다. 현재는 이들 모두를 사용할 수 있으며, 코프로세서의 경우는 부동 소수점 연산과 같은 처리를 위해서 이와 같은 데이터 타입을 가지기도 한다.

◆ **privileged mode** : 대부분의 프로그램은 사용자 모드에서 동작하지만, Exception 처리나 슈퍼바이저 콜과 같은 특수한 루틴의 처리를 위해서 privileged mode를 가지고 있다. Privileged mode는 특정 제어권을 가진 경우에만 들어갈 수 있도록 되어 있다. 시스템 모드를 제외한 각각의 privileged mode는 SPSR(Saved Program Status Register)과 연계되어 수행된다. SPSR은 사용자 모드에서 privileged mode로의 진입시 CPSR의 상태를 저장하고, 사용자 모드로의 복귀시에는 SPSR의 값들을 CSPR로 저장시키는 데 이용된다. 임베디드 시스템을 사용하는 ARM 프로세서들에서는 이러한 보호 기능을 부적절하게 사용하기도 하지만, RTOS의 구성요소에 있어서 privileged mode는 매우 중요한 의미를 지닌다.

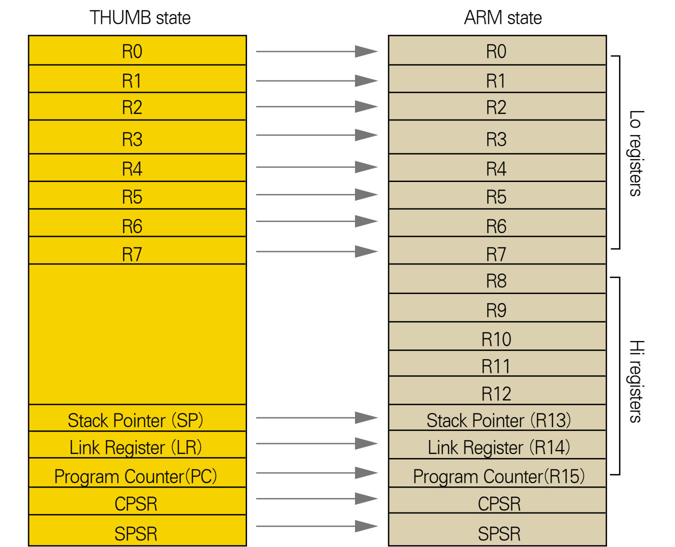
◆ **ARM & Thume state** : ARM 모드와 Thume 모드의 가장 큰 차이점은 레지스터의 사용에서 발견할 수 있다. Thume 모드의 경우 〈그림 10〉에서 보는 바와 같이 실질적으로 사용 가능한 레지스터가 r0~r7까지의 8개 레지스터라는 것이다. Thume 모드에서의 레지스터는 최대 16비트 크기를 가질 수 있다. 명령어의 크기도 Thume 모드의 경우 모두 16비트의 구조를 지니고 있다. 대부분의 Thume 모드의 명령어들은 프로세서 내부에서 32비트 명령으로 확장되어 실행하도록 하고 있어 Thume 모드의 사용에 의한 속도 저하를 만회하도록 하고 있다. Thume 모드로 프로그래밍을 하면 코드의 크기가 작아지는 것은 당연한 일일 것이다. 〈그림 11〉에서 노란색 바탕으로 표시된 레지스터들은 ARM state에서만 사용할 수 있는 레지스터들이다.

컨디션 코드

ARM 명령어 셋은 다음의 코드에서 보는 바와 같이 OPcode의 MSB 4비트에 조건부 코드가 있으며, 이 조건부 코드는 모든 명령어 셋에서 동일하게 적용된다. 조건부 점프 명령은 모든 명령어 셋의 표준으로 되어 있으나, ARM의 경우는 슈퍼바이저 콜과 코어 프로세서 명령들을 포함한 모든 명령어로 이를 확장해서 적용한다. 모든 ARM 명령어의 니모닉의 뒤에 2글자의 조건부 니모닉을 붙여서 사용하며, 아무런 조건을 지정하지 않으면 'always' 조건인 경우로 간주하고 AL은 생략이 가능하다.

```
예제      BEQ      LABEL        ; Branch to LABEL    if Z flag set
          MOVCS    r3, r5       ; r3 ? r5            if C flag set
MOVNE              r2, r4       ; r2 ? r4            if NZ flag set
ADDVS              r0, r1, r2   ; r0 ? r1 + r2       if V flag set
```

<그림 10> ARM vs. THUMB의 레지스터 비교(ARM7TDMI Data Sheet(ARM DDI 0029E), Figure 3-5 참조)



<그림 11> ARM state의 레지스터 구조(ARM7TDMI Data Sheet(ARM DDI 0029E), Figure 3-3 참조)

ARM State General Registers and Program Counter

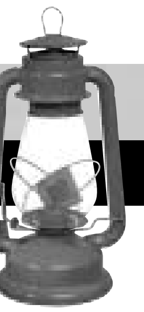
System & User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)

ARM State Program Status Registers

System & User	FIQ	Supervisor	Abort	IRQ	Undefined
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

◣ = banked register

예제의 내용을 살펴보면 첫번째 행은 Zero flag가 설정되면 LABEL로 분기한다. 두번째 행에서는 Carry flag가 설정되었으면, 레지스터 r3로 레지스터 r5의 내용을 가져온다. 세번째 행은 Zero flag가 설정되지 않았으면, 레지스터 r2로 레지스터 r4의 내용을 가져온다. 네번째 행은 오버플로우(V flag가 설정)가 일어나면 레지스터



r1과 레지스터 r2의 내용을 더해서 레지스터 r0에 저장한다.

분기 명령어들

ARM의 분기 명령은 Branch(B) 명령어와 Branch with Link(BL) 명령어로 구분된다. BL의 경우에는 분기가 일어나면 링크 레지스터인 R14에 PC 값을 연결해 놓지만, B의 경우는 단순히 분기만 한다. 따라서 R14에 영향을 주지 않는다.

<표 3> 컨디션 코드 요약(ARM7TDMI Data Sheet(ARM DDI 0029E), Table 4-2 참조)

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N se	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

<표 4> 데이터 처리 명령어의 종류(ARM7TDMI Data Sheet(ARM DDI 0029E), Table 4-3 참조)

Assembler Mnemonic	OpCode	Action
AND	0000	operand1 AND operand2
EOR	0001	operand1 EOR operand2
SUB	0010	operand1 - operand2
RSB	0011	operand2 - operand1
ADD	0100	operand1 + operand2
ADC	0101	operand1 + operand2 + carry
SBC	0110	operand1 - operand2 + carry - 1
RSC	0111	operand2 - operand1 + carry - 1
TST	1000	as AND, but result is not written
TEQ	1001	as EOR, but result is not written
CMP	1010	as SUB, but result is not written
CMN	1011	as ADD, but result is not written
ORR	1100	operand1 OR operand2
MOV	1101	operand2 (operand1 is ignored)
BIC	1110	operand1 AND NOT operand2 (Bit clear)
MVN	1111	NOT operand2 (operand1 is ignored)

```
어셈블러 형태          B{L}{cond} <target address>
예제                   B            LABEL       ; unconditional jump
                       :
      LABEL            MOV          r0, #10     ; r0 ? #10
                       :
      LOOP      SUBS   r0, #1       ; r0 ? r0 ?1 with flag status
                       BNE          LOOP        ; if counter(r0) <> 0 repeat LOOP
```

예제는 LABEL로 무조건 점프한 후에 LABEL에서 루프를 반복할 횟수인 10을 r0에 설정하고, 매 루프를 반복할 때마다 1을 감산하여 결과가 0이 되면 루프를 빠져나온다.

소프트웨어 인터럽트

SWI(SoftWare Interrupt) 명령은 운영체제를 호출하는데 사용되며, 흔히 '슈퍼바이저 콜'이라고도 한다. SWI 명령은 프로세서를 슈퍼바이저 모드(Supervisor Mode)로 전환하고, 0x08번지부터 코드가 실행되는 소프트웨어 인터럽트 명령이다. 자세한 프로세서의 동작은 다음과 같다.

1. 현재 수행 중이던 프로그램의 주소인 PC(R15)를 r14_svc에 저장한다.
2. 현재 사용 중이던 상태 레지스터인 CPSR을 SPSR_svc에 저장한다.
3. 슈퍼바이저 모드 진입한 후, CPSR[4:0]에 "10011"을, CPSR[7]에 "1"을 저장해서 IRQ를 disable한다(FIQ는 disable되지 않는다).
4. PC를 0x08로 셋팅하고, 0x08번지부터 명령어를 실행한다.

반드시 0x08번지에는 SWI 처리를 위한 루틴이 들어 있어야 한다. 그렇지 않으면 프로세서가 어떤 동작을 할지 아무도 모른다. SWI 루틴을 끝내고 다시 돌아오고 싶으면 R14 값을 PC에 옮겨 주는 것 이외에도 SPSR_svc에 저장된 이전의 CPSR도 현재의 CPSR로 가지고 와야 한다.

```
어셈블러 형태   SWI{cond} <24-bit immediate>
예제            ; Enter SWI mode
                MOV          r0, #'A'       ; r0 ? 'A'
                SWI          SWI_writeC                  ; call SWI
                             :
                ; Exit SWI mode
                MOV          PC, r14                     ; PC <-- r14 (Return)
```

앞의 코드는 간단히 r0에 'A'라는 문자를 넣고 SWI_writeC를 호출했지만, SWI 실행 루틴에는 이를 처리하기 위한 코드들이 있어야 한다. 예제가 복잡하지 않으면 R14 값은 PC에 넣는 것만으로도 복귀가 가능하다.

데이터 처리 명령

ARM의 데이터 처리 명령은 레지스터 값을 변경하는데 사용된다. 또한, 3-address 형식을 가지고, 2개의 소스 오퍼랜드와 1개의 목적 레지스터를 이용하고 있다. 하나의 소스 오퍼랜드는 항상 레지스터이어야 하며, 두 번째 오퍼랜드는 레지스터, 시프트 레지스터, 직접적인 값(immediate value)이 될 수 있다. 명령어의 코드(Opcode)에 따라서 다음과 같은 명령이 결정된다.

```
어셈블러 형태        <op>{<cond>}{S} Rd, Rn, #<32-bit immediate>
예제                 <op>{<cond>}{S} Rd, Rn, Rm, {<Shift>}
          LABEL
                     ADD      r5,  r1, r3            ; r5 <-- r1 + R3
                     SUBS     r2, r2, #1             ; r2 <-- r2 <-- 1
                     BEQ      LABEL                  ; r2가 0이면 LABEL로 분기
                     ADD      r0, r0, r0, LSL #2     ; r0 <-- r0 + (r0 << 2)
```

예제의 내용에서 다소 어려운 부분은 가장 마지막의 ADD 명령 부분일 것이다. 잠시 살펴보면, 가장 오른쪽의 r0 레지스터 값을 왼쪽으로 두 번 시프트하면, r0×4가 된다. 여기에 본래의 r0 값을 더하면 결국 r0×4+r0가 되어 r0 ×5의 결과가 나온다.

Single Data 전송 명령

Single Data 전송 명령에는 LDR과 STR 있다. 이 명령어들은 ARM의 레지스터와 메모리 사이의 싱글 바이트 데이터나 문자 바이트 데이터의 전송을 하는 가장 편리한 방법이다. 이 명령어는 명령어의 오피코드(Opcode) 24번 비트인 P의 값에 따라 1인 경우에는 Pre-Indexed Address Mode로, P의 값이 0인 경우에는 Post-Indexed Address Mode로 나누어진다.

다음의 어셈블러 형태를 보면 {B}는 부호 없는 바이트 전송(unsigned byte transer)을 의미하며, {T}는 메모리 변환(memory translation)을 의미하는데, {T} 옵션은 시스템 모드에서만 선택이 가능하다. 또 {!}는 pre-indexed mode에서 다시 써넣기(write-back)의 의미를 가지며, 자동 인덱싱(auto-indexing) 처리를 한다. 다시 써넣기란 베이스 레지스터의 값에 오프셋 값을 더해서 다시 베이스 레지스터로 써넣는 기능이다.

```
어셈블러 형태
   Pre-Indexed form  : LDR|STR{<cond>}{B} Rd, [Rn, <offset>]{!}
Post-Indexed form : LDR|STR{<cond>}{B}{T} Rd, [Rn], <offset>
PC-relative form   : LDR|STR{<cond>}{B} Rd, LABEL
예제                        LDR          r1, UARTADDR      ; r1 <-- UART address
                            STRB         r0, {r1}          ; [r1] <-- r0
                            :
        UARTADDR  &                       &10000000         ; Address literal
```

앞의 예제에서 어셈블러는 UARTADDR의 주소를 r1에 로드하기 위해 pre-indexed, PC-relative 어드레싱 모드를 사용할 것이다. UARTADDR의 리터럴은 반드시 4K 영역 내에 있어야 한다.

SWP

SWP(Swap memory and register instructions) 명령은 다른 로드나 스토어(store) 명령어와 결합되어 사용되며, 두 개의 레지스터의 값을 서로 바꿔주는 명령어이다. SWP 명령의 수행 도중에는 인터럽트에 한 수행정지가 일어나지 않으므로 세마포어의 기본 메커니즘을 구현하는데 유용하게 사용될 수가 있다. SWP 명령을 사용하면서 한 가지 주의할 점은 PC(R15)는 이 명령어에서 레지스터로 사용하지 않도록 한다.

```
어셈블러 형태          SWP{<cond>}{B} Rd, Rm, [Rn]
예제                          ADR  r0, SEMAPHOER
                SWPB  r1, r1, [r0]  ; exchange byte
```

예제는 세마포어의 주소 값을 r0에 저장한 후에 r1의 주소의 내용과 r0의 주소의 내용을 교환하는 기능을 한다.

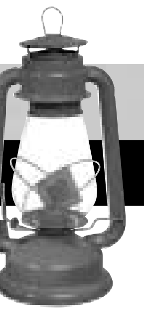
기타 명령어들

이상으로 ARM에 대한 명령어의 소개를 마치기로 한다. 본 기사에 소개한 명령어 이외에도 더 유용한 명령어들이 많이 있으나, ARM의 명령어에 대해서 좀더 자세한 내용을 접하고 싶은 독자는 ARM의 공식 웹사이트에서 제공하는 ARM7TDMI Data Sheet(Document:ARM DDI 0029E)를 참고하기 바란다.

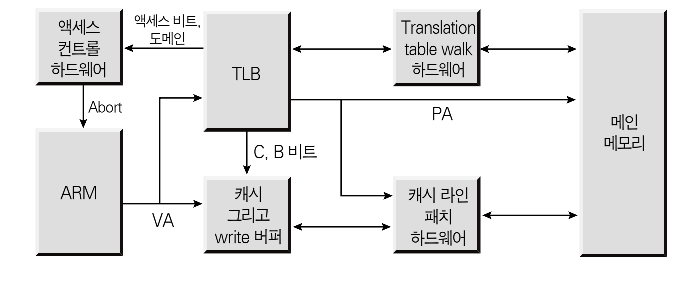
ARM 프로세서의 몇 가지 장점

필자가 사용한 ARM 프로세서는 하이닉스 반도체의 GMS30C7201과 삼성의 S3C44B0X이다. 그 중에서 GMS30C7201 프로세서가 여러분이 잘 알고 있는 스트롱암에 좀더 가깝다. 이들을 잠시 비교해 보자.

첫째로 가장 중요한 점으로 생각되는 것은 CPU의 안정적인 동작이다. 보드를 며칠씩 켜 놓아도 안정적이다. 반도체 설계를 담당하는 주변 사람들의 말에 의하면 칩에서 열이 나지 않는다는 것은 그 만큼 칩에 대한 설계가 잘 되어 있다는 것을 의미한다고 한다. 실제 필자가



<그림 12> ARM MMU 구조



개발 당시에 사용했던 칩들은 모두 상당히 안정적으로 동작했다.

두 번째로는 ARM의 강력한 어셈블러 기능이다. 바이너리 데이터와 코드 데이터를 자유자재로 조합해서 사용할 수 있는 것은 물론 이거니와 블럭 전송 명령에 사용 가능한 모든 레지스터를 동원해서 처리하는 것 등은 필자에게 아주 깊은 인상을 주었다. 실제 필자는 프로젝트를 진행하면서 수백 KB의 바이너리 데이터를 가공하지 않고 프로그램에서 사용해야 하는 경우가 많았으며, 이 부분을 만약 컴파일을 통해서 처리하려고 한다면 아마 상당한 고생을 했을 것이다.

세 번째 인상깊었던 것은 MMU이다. ARM의 강력한 기능 중의 하나이지만 처음에 그 개념을 알기까지는 상당한 어려움이 있었던 것도 사실이다. MMU가 없는 아키텍처를 사용하면 나중에 RTOS에서 포팅할 때 돌아올 수 없는 강(?)을 건너야 한다. 즉, 가상 메모리를 실제 메모리에 전환시키는 작업을 하드웨어가 하지 않으면 그 역할은 고스란히 소프트웨어의 몫으로 남는다. 그러고도 문제가 해결되지 않는다면 독자 여러분들도 MMU가 있는 프로세서를 사용할 것을 권한다.

네 번째로는 LCD 컨트롤러와 VGA 컨트롤러가 같이 내장되어 있어 컬러 모니터와의 인터페이스에 그리 많은 노력을 들이지 않았었다. 다만 컬러의 수가 다소 떨어지는 면이 없지 않았지만, 지금 출시되는 칩들은 이런 걱정들은 하지 않아도 된다.

ARM의 MMU

임베디드 프로세서도 요즈음은 동시에 많은 양의 프로그램을 수행하게 된다. 물론 단일 프로세서가 한번에 하나의 프로그램을 수행하지만 RTOS와 같은 환경에서는 응용 프로그램 개발자가 시스템에 있는 메모리 전부를 다룰 수 있도록 되어 있다. 이러한 일련의 작업들을 지원하기 위해서는 별도의 하드웨어가 필요한데, 이를 위해서 MMU라는 개념을 도입하였다.

ARM7TDMI에는 없지만 그 이상의 버전에서 ARM을 사용하면서 MMU를 모르고 사용한다면 절름발이 ARM을 쓰는 것이다. 물론 자료도 많지 않아서 관련된 자료를 읽어보아도 다소 설명이 어려운 것은 사실이다. 그러나 다시 한번 산을 넘어보자. 이번에는 MMU와 관련된 용어와 그 개념들에 대해서만 알아보자.

ARM의 MMU를 구성하는 데는 적어도 다음의 세 가지 구성요소가 필요한데, TLB(Translation Look-aside Buffer), 접근 제어 로직(Access Control Logic) 그리고, translation table-walking 로직이며, 그 중에서 TLB는 프로그래밍에 도움이 되므로 좀더 자세히 다루기로 한다.

ARM에서 MMU의 중요한 역할 중 하나가 가상 메모리와 물리적 메모리로 변환해 주는 역할이며, 다른 하나는 프로세서의 동작 모드에 따라서 메모리의 접근 권한을 관리하는 역할이다. ARM MMU는 2가지 레벨의 페이지 테이블을 사용하는데 하나는 테이블 작업을 수행해 주는 하드웨어이며, 다른 하나는 최근에 사용된 변환 페이지가 저장되는 TLB이다.

[꼭 컴파일을 많이 해야 프로그램 제작 속도가 빨리 나오는 것일까?]

필자의 경우는 컴파일을 실행하는 수가 그렇게 많지 않다. 컴파일을 해서 버그를 잡는 시간보다는 프로그램의 전체 윤곽과 설계에 전체 개발 시간의 절반 이상을 보낸다. 그래서 가끔은 주변에서 저 사람은 개발기간의 절반은 놀고 있다(?)라는 오해를 받기도 한다. 누구건 경험하는 초보 시절은 있기 마련이다.

초보 시절의 대표적인 악순환 중에 하나는 프로그램의 결정적인 버그를 잡는데, 많은 시간을 보낸다는 것이다. 필자 역시 그런 시절을 겪었다. 프로젝트는 시간이 지나면 지날수록 급박해지고, 주변의 개발자들은 내가 개발한 결과물을 가져가야 다음 일을 진행하고, 그런 상황이 닥치는 순간마다 결과를 빨리 보고 싶어지는 것은 일반적인 사람의 심리인 것이다.

필자와 같이 일을 해 본 프로그래머들의 대부분은 매우 훌륭한 습관들을 가지고 있었다. 그 프로그래머들의 감각과 개발 속도는 가히 따라 배울 만 하다고 이야기하고 싶다. 그들이 성장하는 과정을 지켜보면서 역시 그들도 버그를 잡는데 많은 시간을 투자하는 과정을 피하지는 못하고 있었다. 그들 역시도 컴파일하는 데 많은 시간을 보낸다.

그러나 단지 다르다면 그들은 의도적인 버그를 유도해 내는데 컴파일을 사용한다는 것이다. 즉, 기본적인 개념이 정립되고 나서야 초벌 코딩을 하고, 초벌 코딩을 한 후에 컴파일을 하고, 컴파일에서 나온 사소한 버그들을 잡는 것으로 기본적인 코딩이 끝난다는 것이다.

훌륭한 프로그래머가 되고 싶다면 우선은 가지고 있는 잘못된 습관을 고치는 것부터 시작해야 한다. 그렇지 않으면 늘 버그를 잡는데 대부분의 시간을 허비하는 초급 프로그래머로서 머물러 있는 시간이 많아지게 된다. 하드웨어를 설계하든 소프트웨어를 설계하든 개발을 하는 기본적인 순서와 골격은 크게 벗어나지 않는다.

그리고 좀더 객관적으로 개발이 진행되는 형태를 지켜보면, 하드웨어 개발자이든 소프트웨어 개발자이든 매번 개발할 때마다 같은 문제로 고민을 하고 있다는 것이다. 그 문제의 실체를 알기 전까지는 개발을 하는 당사자도 그 문제가 과거 개발시에 나타났던 문제라는 것을 알고 나서야 그것을 시인하는 것이다. 과거의 개발 과정에 대한 자료를 잘 정리해 두면 위기 상황에서 그러한 문서들이 매우 유용한 경우가 많다.

메모리의 형태

메모리의 맵핑은 동일한 기본 메커니즘에 의해 다양한 형태로 수행되며, 크게는 섹션과 페이지로 구분할 수 있다. 섹션은 1MB 블럭 메모리이며, 페이지는 다시 Large(64KB 블럭), Small(4KB 블럭), 그리고, Tiny(1KB 블럭)로 구분된다. 섹션과 Large Page 방식은 한 개의 TLB로 큰 메모리 영역을 맵핑할 수 있다. 또한 Small Page 방식은 1KB 블럭의 sub-Page로 확장되고, Large-Page 방식은 16KB 블럭으로 확장된다. 그러나 Small page 영역에 많은 양의 데이터를 할당하면, 경우에 따라서는 TLB 성능이 떨어질 수도 있다고 하니 적절한 메모리 모델을 선택하는 것도 중요하다.

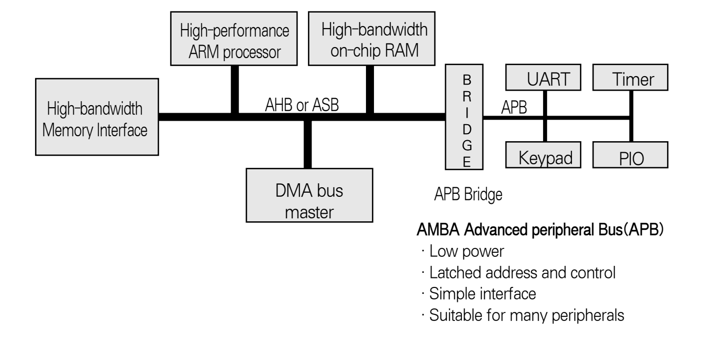
도메인이란?

도메인은 ARM MMU 아키텍처의 특징 중 하나인데, 도메인은 섹션이나 개별 접근 권한을 가지는 페이지들의 그룹이다. 도메인은 동일한 변환 테이블을 가지고 수많은 다른 프로세스들을 실행할 수 있도록 해준다. 도메인에 의해 각각의 프로세스들이 각각의 변환 테이블을 가지고 있을 때보다 좀더 가벼운 프로세스의 전환 메커니즘을 가지게 된다.

TLB란?

TLB란 Translation Look-aside Buffer의 약자인데, 페이징(Paging)의 개념이 프로그래머에게 메모리를 사용하는 데 있어 완전한 자유도와 투명성을 제공한다. 각각의 메모리에 대한 접근을 위해서 필요한 두 가지의 오버헤드(페이지 디렉토리와 페이지 테이블의 구성)가 필요하며, 이러한 오버헤드를 피하기 위해서 TLB라는 개념을 도입했다. TLB는 페이지 변환에 사용되는 일종의 캐시이며, 그 역할은 가상 어드레스를 물리 어드레스로 변환과 접근 권한을 캐싱(caching)하는 것이다. 즉, TLB에 가상 어드레스를 위한 변환 엔트리가 있으면, 메모리 접근제어 로직이 메모리의 접근가능 여부를 판단하고, 접근이 허용되는 경우에만 MMU는 virtual address에 대한 physical address를 보내주고, 그렇지 않으면 MMU에 의해 프로세서는 abort 신호를 출력한다. 특히, ICE와 같은 디버깅 도구를 사용하다 보면 프로그램의 다운로드가 정상적으로 되지 않으면서 abort 신호가 나오는 경우를 볼 수가 있는데, MMU의 이런 기능 때문에 발생하는 것이다. TLB가 없는 경우도 있을 수 있는데, 이런 경우는 physical memory에 있는 변환 테이블에서 정보를 읽어다 TLB에 저장한다.

<그림 13> AMBA 시스템에서의 APB(AMBAtm Specification rev2.0(ARM IHI 0011A), Figure 1-1 참조)



변환 테이블

ARM의 MMU는 두 가지 레벨의 변환 테이블을 가지고 있는데, 섹션의 변환과 섹션 레벨 테이블의 포인터를 가지고 있는 1st 레벨 테이블과 Large/Small 페이지의 변환 값을 가지는 2nd 레벨 테이블이 그것이다. 변환에 대한 자세한 방법은 지면 관계상 생략하기로 한다.

ARM의 AMBA

프로세서의 버스 아키텍처를 이해하는 것은 프로세서의 성능을 예측하는데 있어서 상당히 중요하다. ARM7TDMI에서는 제공하지 않지만, AMBA는 엄연히 ARM의 버스 인터페이스에 있어서 핵심 중의 하나이다. 고속 캐시와 인터페이스를 위한 최적화를 위해서 AHB, ASB, APB로 구성된다.

AHB는 고속의 버스 처리를 위해 설계되었으며, 고성능 시스템 모듈간의 버스 연결에 사용된다. 특히 AHB는 Burst Mode의 데이터 전송과 분산 처리에 이용된다. ASB(Advanced System Bus)는 고성능 시스템 모듈에 연결되어 사용된다. 이것은 Burst Mode의 데이터 전송에 이용된다. APB(Advanced Peripheral Bus)는 전송속도가 느린 장치들의 인터페이스에 사용되며, 시리얼 통신 모듈과 같은 부분은 APB에 접속된다.

ARM과 MIPS 중 어느 것을 선택할 것인가?

ARM이 시장 점유율 면에서 MIPS보다 우위를 점령하고 있는 것은 사실이고, ARM 아키텍처가 우수하다는 점은 이미 여러 가지 면에서 부인할 수 없는 사실이다. 그러나 MIPS도 그 구조를 살펴보면 ARM에 못지 않은 훌륭한 아키텍처를 가지고 있다. MIPS가 시장에서 고

전하는 이유는 개발자의 저변 확대에 있어서 아직은 ARM을 따라가지 못하고 있기 때문이다. 아직 시장에서 두 프로세서들 간의 경쟁은 끝나지 않았다. MIPS는 ARM을 추격하기 위해서 엄청난 노력을 하고 있으며, 일부에서 그 결과가 나타나고 있다. 다만 지금은 ARM이 시장을 지배하고 있다는 사실과 ARM 사용자가 MIPS 사용자보다 많고 아직은 기술지원의 측면에서 MIPS가 ARM보다는 다소 떨어지는 면이 없지 않다고 본다.

하드웨어 개발자든 소프트웨어 개발자든 자신이 익숙한 것을 계속 사용하고 싶은 습성이 있다. 그러한 측면에서 본다면 이미 ARM을 사용하던 사용자가 MIPS를 사용하기 위해서 돌아설 가능성은 다소 희박하다고 볼 수 있다. 그러나 처음에 프로세서를 선택하는 경우라면 좀더 자신이 쉬워 보이는 프로세서를 선택할 것이다. 언제 무엇을 선택하든 그 기준은 누가 많이 사용하는가도 중요하지만 그보다도 중요한 것은 "개발할 제품의 성격에 맞는가"라는 질문을 독자 여러분 스스로 해 보아야 한다는 것이다. 어느 것을 선택해도 아키텍처를 검토하고 개발자의 것으로 만들기까지의 과정은 항상 험난하기 마련이다. 어디까지나 선택은 독자 여러분의 몫이라는 점을 잊지 않기를 바란다.

[어셈블리 언어가 너무 어렵다?]

어셈블리 언어가 너무 어렵다는 것은 이제 옛말이 된 것 같다. 요즈음 나오는 프로세서들을 보면 언어와 관련된 자료도 꽤 풍부한 편이고, 주변에 저변도 많이 넓어진 편이다. 일반적으로 소프트웨어만을 배워온 독자라면 어셈블리 언어가 기타 언어보다 어렵게 느껴지는 것은 어쩌면 당연한 일인지도 모른다. 어셈블리 언어가 어렵다는 것은 결국 해당 프로세서의 하드웨어 아키텍처를 제대로 이해하지 못했다는 이야기가 된다. 어셈블리 언어는 명령어 그 자체가 어려운 것이 아니라, 하드웨어와의 상관 관계를 고려하지 않고는 생각할 수 없는 언어이기 때문이다. 좀더 다른 관점에서 생각해 본다면 어셈블리 언어의 명령어 자체는 매우 단순하고도 간단한 개념만을 모아놓은 것에 불과하다. 즉, 프로세서의 하드웨어 아키텍처 설계상의 단점을 보완하기 위해서 고안된 제2의 하드웨어로 볼 수 있다는 것이다.

어셈블리 언어는 이러한 관점에서 본다면 제한된 조건에서 시스템의 타이밍과 관련된 문제를 해결하는데 종종 아주 좋은 대안이 되는 이유가 된다. 이때의 제한된 조건이라는 것은 짧은 시간에 많은 기능을 구현하는 코드를 개발하는 소프트웨어 개발자의 생산성을 고려한다면 어셈블리 언어에 익숙하지 않은 소프트웨어 개발자에게 어셈블리 언어가 좋은 대안이 되지는 못하기 때문이다. 요즈음 C와 같은 고급언어와 어셈블리 언어를 혼합해서 사용하는 이유가 바로 이런 단점을 보완하는 것이다. 혹자는 어셈블리 언어가 아주 필요없는 언어라고 이야기할 수도 있지만, 시스템의 전반적인 구성요소로 본다면 어셈블리 언어는 하드웨어와 응용 프로그램의 중간에서 미들웨어의 역할이 필요한 부분에 있어서 없어서는 안될 중요한 언어이다. 모든 프로그래머가 어셈블리 언어를 배울 필요는 없지만, 적어도 임베디드 시스템을 다루어야 하는 프로그래머라면 반드시 한번은 넘어야 할 산임에는 분명하다.

ARM 프로세서의 개념을 잡기 바라며

우선 그 무엇보다도 처음 ARM을 접하는 독자들에게는 CPU 아키텍처에 대한 충분한 사전 지식을 쌓아달라고 권하고 싶다. 대부분의 프로그래머들이 CPU 아키텍처에 대한 부분만 나오면 외면하는 경향이 있는데, ARM은 절대적으로 아키텍처에 대한 사전지식이 필요한 프로세서이다. 요즘은 ARM에 대한 개발 툴들도 상당히 보편화되어 있는 편이다. 문제는 아무리 개발 툴을 잘 가지고 있더라도 본인이 프로세서에 대한 배경지식을 쌓지 않으면, 어려운 문제에 당면했을 때 결코 해결할 수 없다는 것이다.

필자의 경우는 개발 당시에 ARM에 관련된 자료는 주저하지 않고 밤을 세워가면서라도 모두 보았다. 그렇게 읽은 많은 문서들은 필자에게는 지금도 매우 유용한 지식으로 남아 있으며, ARM에 대한 문제가 나오면 가장 먼저 살펴보는 것이 칩을 개발한 회사에서 제공하는 기술문서들이다. ARM에 대한 모든 자료는 거의 ARM 공식 사이트인 www.arm.com에서 분야별로 모두 쉽게 구할 수가 있다. 혹시 부족하다고 생각되는 독자는 참고자료에서 언급한 Steve Furber의 『ARM System Architecture』를 읽어볼 것을 권한다. 필자는 무엇보다도 모든 문제를 정석대로 푸는 것을 좋아하고 실제 프로젝트에서도 그렇게 문제를 풀어간다. 무엇을 하든 기본은 개발사가 제공하는 문서들을 하나도 빠짐없이 꼼꼼히 읽어보는 것이 중요하다.

어떤 경우에도 문제를 해결하는 것은 결국 개발자인 당사자가 하는 것이다. ARM 프로세서의 개념이 처음 접하는 독자들에게 결코 쉽지 않다는 것을 감안해서 최대한 쉽게 설명하려고 노력했지만 그래도 지면 관계상 다루지 못하는 내용이 너무 많았다는 점을 필자도 시인한다. 다만, 이 기사가 ARM을 처음 접하는 독자들에게 조금이나마 도움이 되었으면 하는 것이 필자의 바람이며, 지금도 밤낮으로 개발에 몰두하는 많은 개발자들에게 아낌없는 찬사를 보낸다.

정리 | 조규형 | jokyu@korea.cnet.com

참 고 자 료

❶ "ARM System Architecture", Steve Furber, ADDISON-WESLEY
❷ "ARM Architecture Reference Manual", David Seal, Addison-Wesley
❸ "ARM7TDMI Data Sheet", ARM DDI 0029E, www.arm.com
❹ "ARM system-on-chip Architecture, 2nd Edition", Addison-Wesley
❺ "The ARM11 Microarchitecture", David Cormie, www.arm.com/support/White_Papers
❻ "ARM7TDMI rev 4 Technical Reference Manual", ARM DDI 0210B, www.arm.com
❼ "AMBAtm Specification rev2.0", ARM IHI 0011A, www.arm.com