2. ARM 구조

– 전체 요약

The ARM processor is a *Reduced Instruction Set Computer* (RISC). The RISC concept, as we saw in the previous chapter, originated in processor research programmes at Stanford and Berkeley universities around 1980.

In this chapter we see how the RISC ideas helped shape the ARM processors. The ARM was originally developed at Acorn Computers Limited of Cambridge, England, between 1983 and 1985. It was the first RISC microprocessor developed for commercial use and has some significant differences from subsequent RISC architectures. The principal features of the ARM architecture are presented here in overview form; the details are postponed to subsequent chapters.

In 1990 ARM Limited was established as a separate company specifically to widen the exploitation of ARM technology, since when the ARM has been licensed to many semiconductor manufacturers around the world. It has become established as a market-leader for low-power and cost-sensitive embedded applications.

No processor is particularly useful without the support of hardware and software development tools. The ARM is supported by a toolkit which includes an instruction set emulator for hardware modelling and software testing and benchmarking, an assembler, C and C++ compilers, a linker and a symbolic debugger.

2.1 Acorn RISC Machine

- 첫 ARM 프로세서는 1983 10월에서 1985년 4월 사이에 Acorn 컴퓨터 회사에 의해 만들어짐, 이 당시의 ARM은 Acorn RISC Machine의 약자를 나타냄
- Acorn사는 1982년 BBC 마이크로 컴퓨터의 성공으로 개인용 컴퓨터 시장에서 핵심적인 위치를 차지하였고 다음 버전의 마이크로프로세서 설계를 고려하기 시작
- 기존의 16 비트 CISC 마이크로 프로세서는 메인 메모리보다 느렸고 명령어를 처리하는 시간이 많이 길고 인터럽트 처리 시간도 길어 비효율적
- Acorn사는 독자적인 마이크로프로세서 설계를 고려하기 시작하였으나 제한된 인원과 큰 규모의 설계 경험이 미흡
- Berkeley RISC I은 ARM 프로세서 설계의 기초가 되었고 설계 요소들의 우연한 조합에 의해 ARM 프로세서가 만들어짐
- 1990년에 Acorn사는 ARM로 이름을 바꾸고 ARM은 Advanced RISC Machine의 약자로 변경됨

2.2 구조적인 계승

- Berkeley RISC와 같은 ARM 특징

  ▷ load-store 구조, 32 비트 고정 명령어 길이

  ▷ 3주소 명령어 형식

- Berkeley RISC와 다른 ARM 특징

  ▷ 레지스터 윈도우

  Berkeley RISC 프로세서는 많은 레지스터를 사용, 특히 프로시저 입출력 명령어일 때 새로운 레지스터 접근을 용이하게 하기 위해 레지스터 윈도우를 이용하여 접근할 수 있는 32개의 레지스터를 조정, ARM은 레지스터 수를 줄이고 예외 처리를 위해 shadow 레지스터를 사용

  ▷ 지연 분기 명령어

  대부분의 RISC 프로세서가 파이프라인 제어 해저드를 해결하기 위해 도입하였지만 예외처리를 복잡하게 하고 branch 예측 메카니즘에서 나쁘게 상호 작용하고 슈퍼스칼라 구현을 어렵게 하므로 ARM은 사용 안함

▷ 모든 명령어의 single 사이클 실행

데이터 이동 명령어는 명령어와 데이터를 위해 두번의 메모리 접근이 필요한데 single 사이클 동작을 위해 명령어와 데이터 메모리가 별도로 필요

ARM은 이러한 제약을 없애기 위해 멀티 사이클 실행 명령어를 도입하고 메모리 접근에 효율적인 어드레싱 모드를 추가

- ARM Simplicity

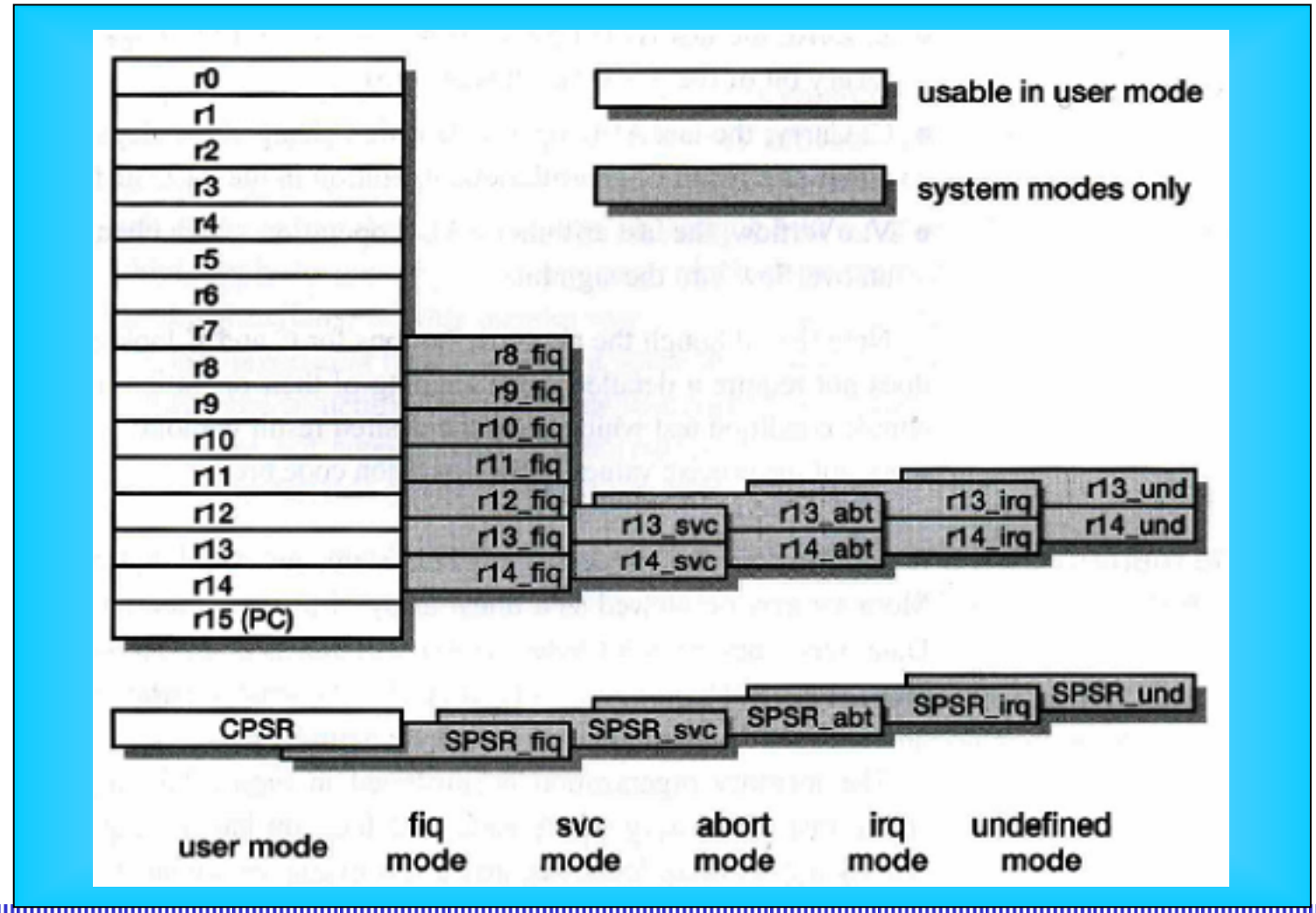
▷ 명령어 세트보다 하드웨어 조직과 구현 때문에 간단

▷ 중요한 CISC 특징을 유지하면서 RISC 아이디어의 기초에서 만들어진 명령어 세트와 간단한 하드웨어의 결합은 순수한 RISC보다 코드 크기를 줄이고 파워 효율성을 가지게 됨

2.3 ARM 프로그램 모델

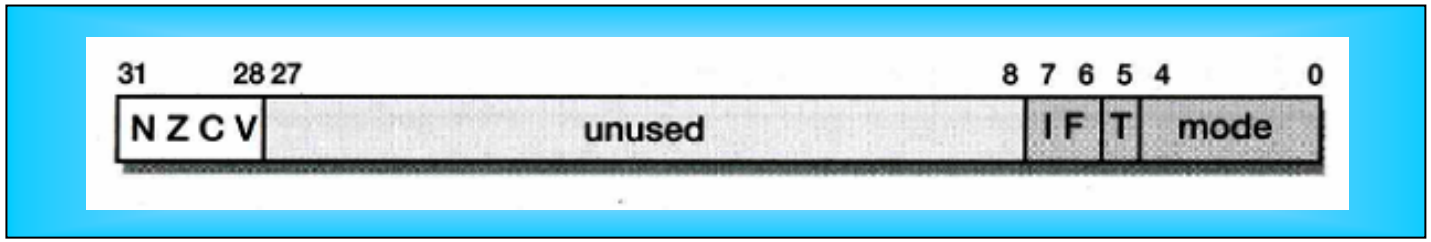
- 명령어는 정해진 규칙에 의해 명령어 처리 전 상태에서 명령어 처리 후의 상태로 프로세서의 상태를 바꾸기 위해 실행됨, 상태는 레지스터나 메모리에 저장

- 사용자 모드는 15개의 범용 32 비트 레지스터 (r0-r14), 프로그램 카운터 (r15), 프로그램 상태 레지스터 (CPSR) 를 사용할 수 있음



- 프로그램 상태 레지스터 (CPSR)

▷ 프로세서의 연산 결과나 비교 동작의 결과를 저장하기 위한 사용자 모드 레지스터로 조건 branch 여부를 결정하기 위해 사용



▷ 프로세서 모드, 명령어 세트, 인터럽트 인에이블은 제어 플래그

- N: Negative; the last ALU operation which changed the flags produced a negative result (the top bit of the 32-bit result was a one).
- Z: Zero; the last ALU operation which changed the flags produced a zero result (every bit of the 32-bit result was zero).
- C: Carry; the last ALU operation which changed the flags generated a carry-out, either as a result of an arithmetic operation in the ALU or from the shifter.
- V: oVerflow; the last arithmetic ALU operation which changed the flags generated an overflow into the sign bit.

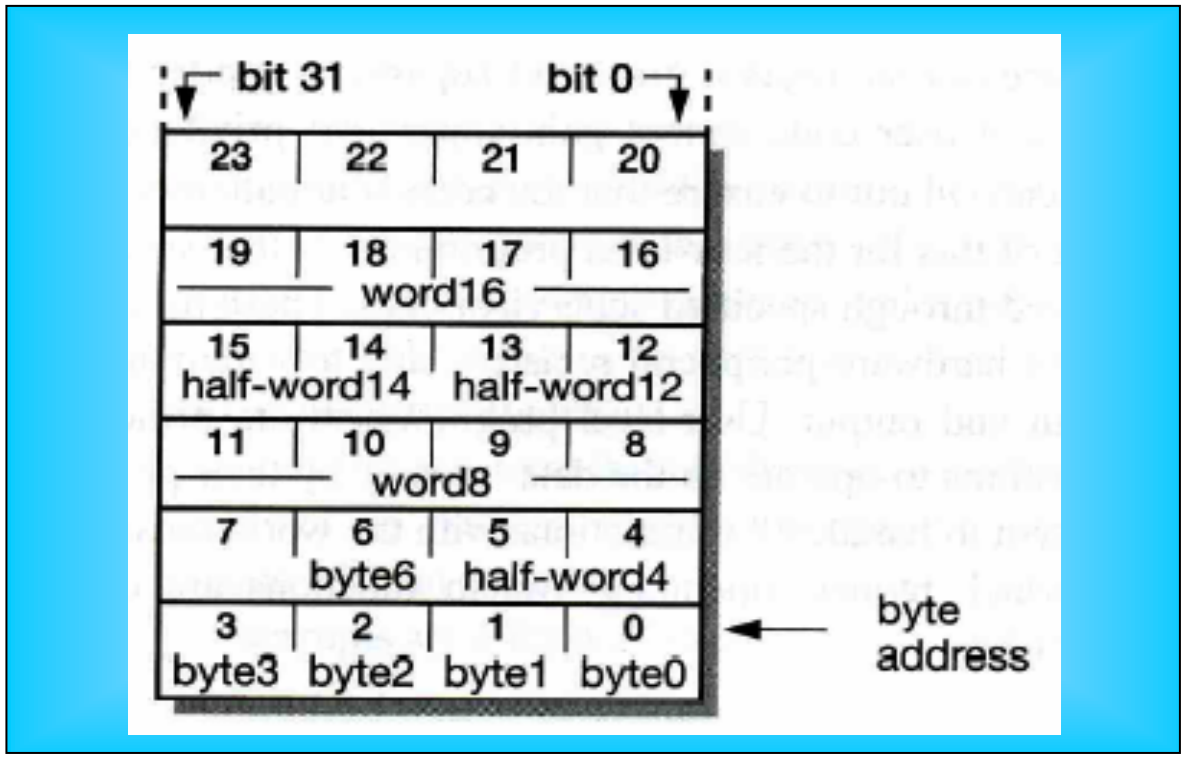
- 메모리 시스템

▷ 프로세서의 상태는 메모리에 저장 (메모리 상태)

▷ 메모리는 바이트 단위의 선형적인 배열 (0 – $2^{32}-1$의 주소를 가짐)

▷ 데이터 형태는 8 비트(바이트), 16 비트 (half word), 32 비트 (word)

▷ ARM 메모리 조직



- Load-store 구조

  ▷ 레지스터에 있는 값만 데이터 처리를 할 수 있고 결과를 레지스터에 저장

  ▷ 메모리와 관련된 동작은 메모리의 데이터를 레지스터로 이동 (load 명령)과 레지스터 내용을 메모리로 이동 (store 명령)

  ▷ CISC 프로세서는 메모리의 데이터를 레지스터 값에 더할 수 있고 결과를 메모리에 저장할 수 있음, 오퍼랜드 중에 하나는 메모리 허용 (8086 계열)

  ▷ ARM의 주요 명령어

  데이터 처리 명령어 – 산술 논리 명령어, 레지스터만 사용

  데이터 이동 명령어 – load, store, exchange 명령어

  제어 흐름 명령어 – branch, branch and link, supervisor call 명령어

- Supervisor mode

  ▷ 사용자 코드에 의한 불법적인 동작으로부터 프로세서를 보호하기 위해 사용자가 접근할 수 없는 supervisor mode 지원

  ▷ 시스템 레벨 함수들은 반드시 supervisor calls을 통해 처리되어야 함

▷ 시스템 레벨 함수에는 하드웨어 주변 레지스터 접근이나 문자 입출력 같이 많이 사용되는 동작들이 포함, 사용자 프로그램은 시스템 레벨 함수를 사용하기 위해 운영체제에 supervisor call을 함

- ARM 명령어 세트

- The load-store architecture;
- 3-address data processing instructions (that is, the two source operand registers and the result register are all independently specified);
- conditional execution of every instruction;
- the inclusion of very powerful load and store multiple register instructions;
- thc ability to perform a general shift operation and a general ALU operation in a single instruction that executes in a single clock cycle;
- open instruction set extension through the coprocessor instruction set, including adding new registers and data types to the programmer's model;
- a very dense 16-bit compressed representation of the instruction set in the Thumb architecture.

- I/O 시스템

  ▷ I/O 장치를 인터럽트 지원하의 memory-mapped 장치로 간주

  ▷ I/O 장치의 내부 레지스터는 메모리 영역 내의 주소가 할당되고 load-store 명령을 사용하여 읽혀지고 쓰여짐, IRQ나 FIQ 입력을 통해 인터럽트 요청

  ▷ IRQ는 대부분의 인터럽트 소스가 연결되고 FIQ는 처리시간이 중요한 한 두 개의 소스가 연결, 레벨에 의해 인터럽트가 감지되고 maskable 가능

- ARM 예외

  ▷ 예외 처리를 통해 모든 인터럽트, 트랩, supervisor call 등을 지원

1. The current state is saved by copying the PC into r14_*exc* and the CPSR into SPSR_*exc* (where *exc* stands for the exception type).
2. The processor operating mode is changed to the appropriate exception mode.
3. The PC is forced to a value between 00_{16} and $1C_{16}$, the particular value depending on the type of exception.

  ▷ 예외 처리기는 스택 주소를 나타내기 위해 r13_exc을 사용

2.4 ARM 개발 툴

- 소프트웨어 개발을 위해 좋은 환경을 제공하지 않는 시스템에서도 ARM이 사용되도록 하기 위해 ARM 개발 툴은 교차 개발 환경 (cross-development)을 지원하여야 함 (PC 윈도우 환경에서 동작하는 개발 툴이지만 ARM 코드를 생성)
- ARM은 어셈블리 언어나 C 언어를 사용하여 프로그램 가능
- ARM symbolic 디버거 (ARMsd)
  - ▷ 소프트웨어 에뮬레이터 (ARMulator)나 ARM 개발보드에서 동작하는 디버깅 프로그램을 지원
  - ▷ 실행 프로그램이 ARMulator나 ARM 개발 보드에 load되고 실행되도록 함
  - ▷ 소스 레벨 디버깅과 breakpoint 설정을 통한 디버깅을 지원

- ARM 교차 개발 toolkit 구조