6. 고급 언어를 위한 구조적인 지원

- 전체 요약

High-level languages allow a program to be expressed in terms of abstractions such as data types, structures, procedures, functions, and so on. Since the RISC approach represents a movement away from instruction sets that attempt to support these high-level concepts directly, we need to be satisfied that the more primitive RISC instruction set still offers building blocks that can be assembled to give the necessary support.

In this chapter we will look at the requirements that a high-level language imposes on an architecture and see how those requirements may be met. We will use C as the example high-level language (though some might debate its qualification for this role!) and the ARM instruction set as the architecture that the language is compiled onto.

In the course of this analysis, it will become apparent that a RISC architecture such as that of the ARM has a vanilla flavour and leaves a number of important decisions open for the compiler writer to take according to taste. Some of these decisions will affect the ease with which a program can be built up from routines generated from different source languages. Since this is an important issue, there is a defined *ARM Procedure Call Standard* that compiler writers should use to ensure the consistency of entry and exit conditions.

Another area that benefits from agreement across compilers is the support for floating-point operations, which use data types that are not defined in the ARM hardware instruction set.

6.2 데이터 형태

- 정수형 데이터

  ▷ ARM은 32 비트 데이터를 기본적으로 처리하고 16 비트, 8 비트 데이터도 처리할 수 있음, 32 비트 이상의 정수는 하나의 레지스터에 표현 불가능

  ▷ 부호 없는 수의 연산에서 오버플로우를 표시하기 위해 C 플래그를 사용하고 부호 있는 수의 연산에서 오버플로우를 표시하기 위해 V 플래그를 사용

  ▷ 64 비트 정수는 두 개의 레지스터를 이용하고 조건 비트를 이용하여 연산함

- 실수형 데이터

  ▷ 실수는 분수와 초월수로 표현되는 물리적인 양을 나타내기 위해 사용

  ▷ ARM core는 실수 데이터를 지원하지 않기 때문에 실수 데이터를 가지는 명령어는 부동 소수점 coprocessor나 소프트웨어로 emulate되어야 함

- 문자 데이터

  ▷ 문자는 7 비트 ASCII 코드나 unicode 등에 의해 표현되고 ARM 구조에서는 unsigned 바이트 load/store 명령어를 통해 지원됨

- ANSI C 기본 데이터와 파생된 데이터 형태

- **Signed and unsigned characters** of at least eight bits.
- **Signed and unsigned short integers** of at least 16 bits.
- **Signed and unsigned integers** of at least 16 bits.
- **Signed and unsigned long integers** of at least 32 bits.
- **Floating-point, double** and **long double** floating-point numbers.
- **Enumerated** types.
- **Bitfields.**

- **Arrays** of several objects of the same type.
- **Functions** which return an object of a given type.
- **Structures** containing a sequence of objects of various types.
- **Pointers** (which are usually machine addresses) to objects of a given type.
- **Unions** which allow objects of different types to occupy the same space at different times.

▷ ARM 컴파일러는 기본 정수형을 제외하고 최소 크기를 각 데이터형에 할당

▷ Enumerated형은 설정된 범위의 값을 가지고 가장 작은 정수형 형태로 구현

▷ Bitfield형 몇 개가 하나의 정수를 공유

- C 데이터 형태에 대한 ARM의 구조적인 지원

  ▷ Integer, long integer, unsigned 문자 데이터를 위해 signed, unsigned 32 비트와 unsigned 8 비트를 처리하는 명령어를 제공

  ▷ 포인터를 위해 기존의 unsigned 32 데이터 처리 명령어 사용

  ▷ Shorter integer, signed 문자 데이터를 위해 signed, unsigned 16 비트와 signed 8 비트를 처리하는 명령어 제공

  ▷ Array와 structure 지원을 위해 base plus scaled index addressing 모드와 base plus immediate offset addressing 모드를 제공

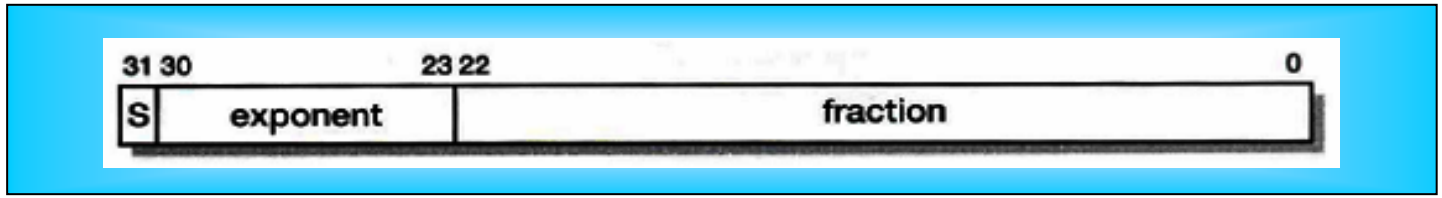
  ▷ 부동소수점 데이터형은 ARM core에서 지원하지 않기 때문에 부동 소수점 지원 하드웨어가 없으면 복잡한 소프트웨어 emulation 루틴에 의해 처리됨

6.3 부동 소수점 데이터 형태

- Single precision 형식

  ▷ 실수의 일반적인 형태 : $R = a \times b^n$ (2진수로 저장하기 위해 b=2)

▷ 부동 소수점 데이터형 표현의 일치성을 위해 IEEE 754에 의해 기본적으로 32 비트 single precision 형식으로 표현 (부호 비트 (S), 지수 (8 비트), 유효자리 (23 비트)로 구성)



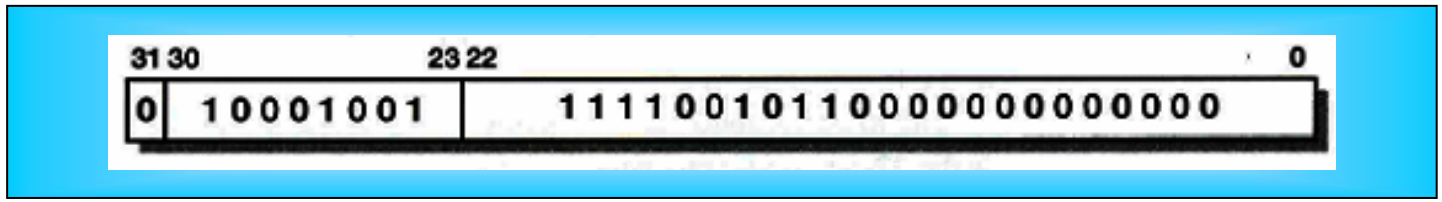
▷ 1보다 작은 실수는 지수가 음수인데 음수 지수 대신 +127 bias한 값을 지수로 사용

▷ 예) 1995를 single precision 형식으로 저장하는 경우

i) 1995를 2진수로 변환 : 1995 = 11111001011

ii) $a \times b^n$ ($1 \le a < 2$, $b=2$) 형태로 정규화 : $1995 = 1.1111001011 \times 2^{10}$

iii) 지수를 +127 bias : 10+127 = 137 (10001001)



▷ 32 비트 정규화된 부동 소수점 형식 : $(-1)^S \times 1.$유효자리 $\times 2^{(지수-127)}$

0 : 지수와 유효자리를 모두 0로 저장

±∞ : 지수는 최대값 (255), 유효자리는 0, 부호에 따라 S는 0(1)

NaN (Not a Number) : 지수는 최대값 (255), 유효자리는 non-zero

▷ 정규화된 형태로 표현할 수 없는 아주 작은 실수는 비정규화된 형태로 표현

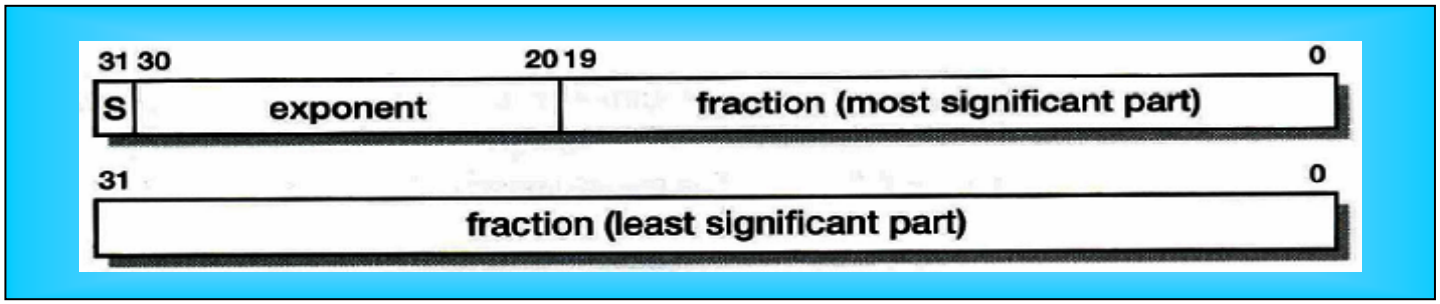
지수는 최소값인 0을 가짐

value = $(-1)^S \times 0.$유효자리 $\times 2^{(-126)}$

- Double precision 형식

▷ 부동 소수점 데이터의 정확도를 높이기 위해 64 비트를 사용

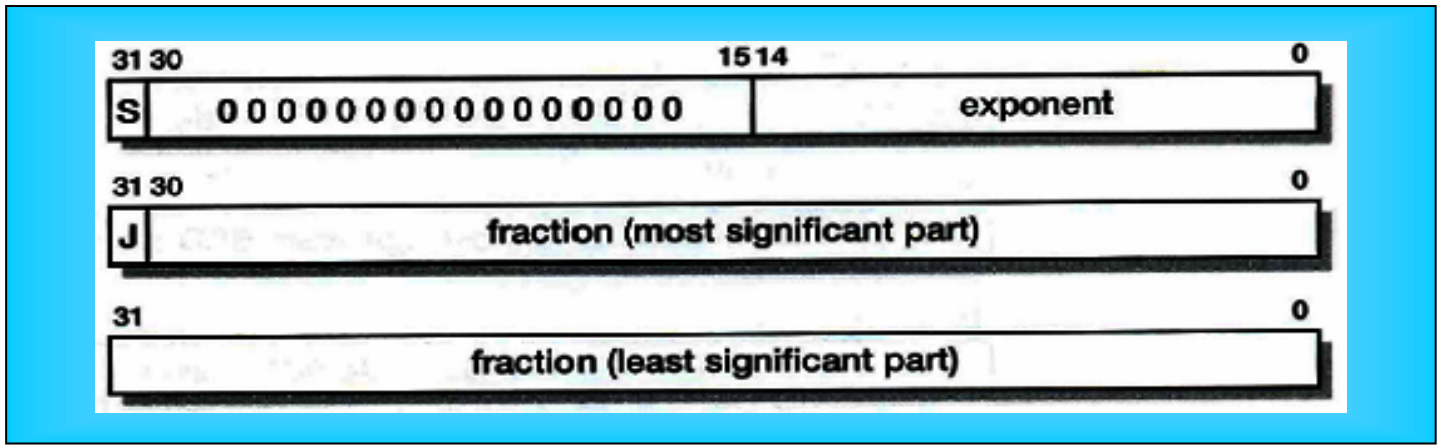
▷ 지수는 11 비트 (bias는 +1023), 유효자리는 52 비트 사용



Single precision		Double precision		설 명
지 수	유효자리	지 수	유효자리	
0	0	0	0	0
0	nonzero	0	nonzero	±비정규화된 수
1-254	anything	1-2046	anything	±부동소수점 수
255	0	2047	0	±무한대
255	nonzero	2047	nonzero	NaN(Not a Number)

- Double extended precision 형식

▷ 정확도를 더 높이기 위해 80 비트를 사용, 지수는 15 비트 (bias는 16383), 유효자리는 63 비트, 정규화된 데이터인 경우 J 비트에 1 저장

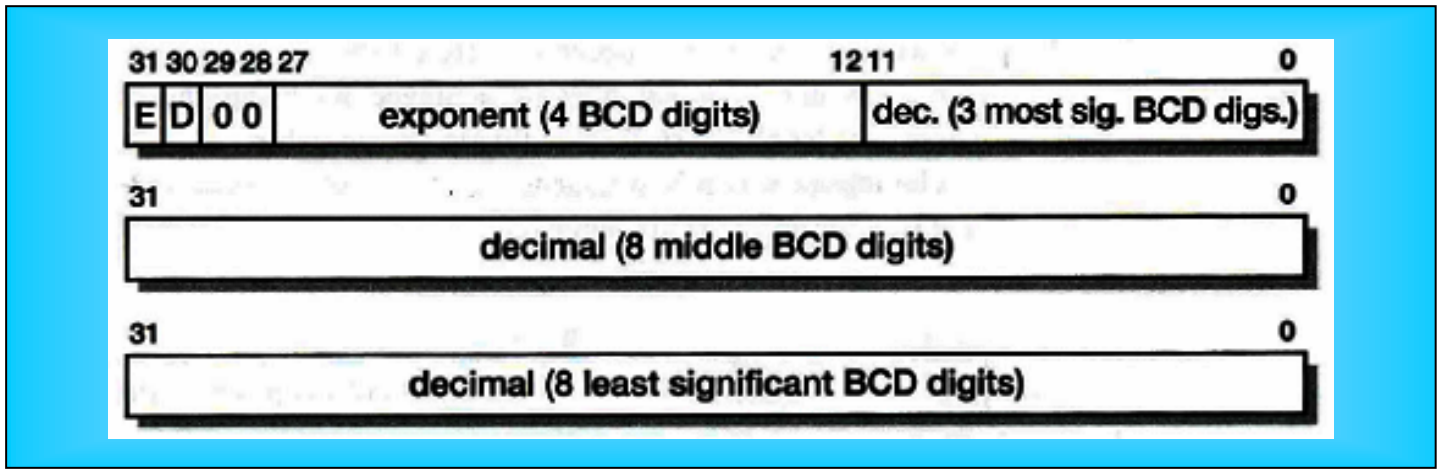


- Packed decimal 형식

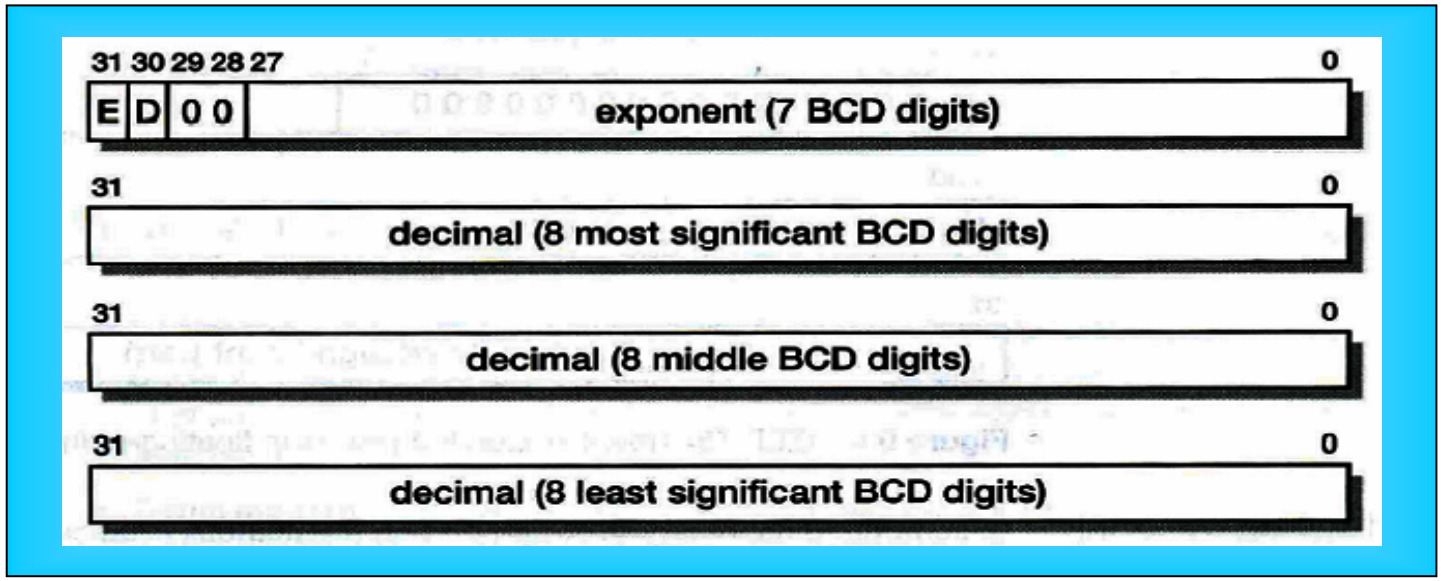
▷ IEEE 754 표준은 이진수 부동 소수점 표현과 함께 10진수 부동 소수점 형식을 정의함, 각 비트는 이진수가 아닌 BCD 코드로 저장됨

$$\text{value (packed)} = a \times b^n \ (1 \le a < 10,\ b = 10)$$

$$= (-1)^D \times \text{decimal} \times 10^{((-1)^{\wedge}E \times \text{exponent})}$$



▷ Extended Packed decimal 형식은 4 바이트를 이용하여 Packed decimal 형식을 확장함



- ARM 부동 소수점 명령어와 부동 소수점 라이브러리

  ▷ ARM 부동 소수점 명령어는 coprocessor 명령어로 정의됨

  ▷ 부동 소수점 명령어는 undefined 명령어 트랩을 통해 소프트웨어에서 처리되지만 일부분은 FPA10 부동 소수점 coprocessor에서 처리됨

  ▷ 부동 소수점 명령어 대신에 ARM은 single, double precision 형식을 지원하는 C 부동 소수점 라이브러리를 제공 (부동 소수점 명령어를 해석하고 emulation할 필요가 없음)

6.4 ARM floating-point 구조

▷ 부동 소수점 지원이 필요하면 ARM 부동 소수점 구조는 전적으로 소프트웨어나 FPA 부동 소수점 accelerator를 토대로 한 소프트웨어/하드웨어 결합 solution을 사용하여 부동 소수점 형태를 지원함

▷ ARM 부동 소수점 구조

- An interpretation of the coprocessor instruction set when the coprocessor number is 1 or 2. (The floating-point system uses two logical coprocessor numbers.)
- Eight 80-bit floating-point registers in coprocessors 1 and 2 (the same physical registers appear in both logical coprocessors).
- A user-visible floating-point status register (FPSR) which controls various operating options and indicates error conditions.
- Optionally, a floating-point control register (FPCR) which is user-invisible and should be used only by the support software specific to the hardware accelerator.

▷ ARM coprocessor 구조는 부동 소수점 emulator 소프트웨어나 FPA10와 FPA 지원 코드의 조합 등의 하드웨어/소프트웨어 조합 등에 모두 사용됨

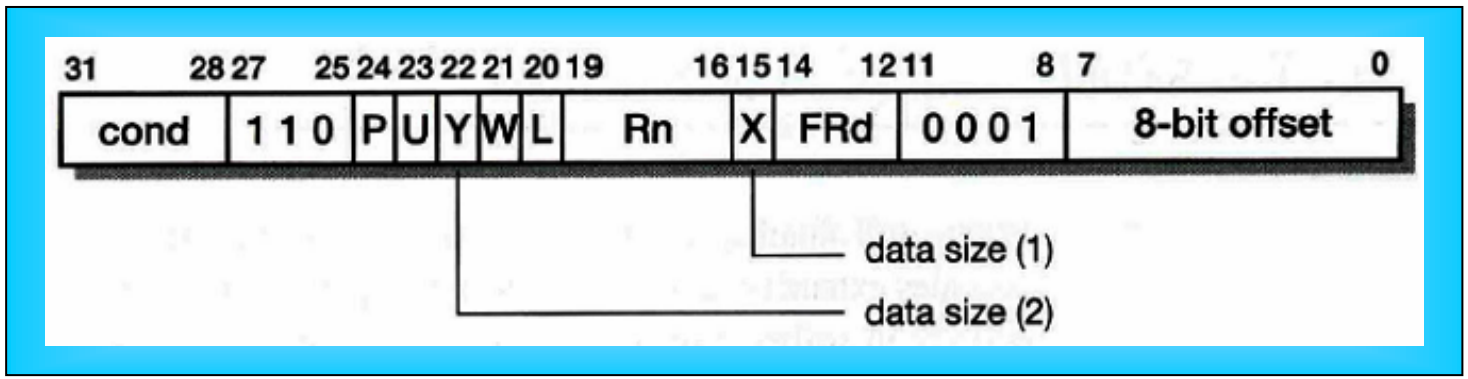
- FPA 10 데이터 형태

▷ ARM FPA 10 부동 소수점 accelerator는 single, double, extended double precision 형식을 지원, Packed decimal 형식은 소프트웨어로만 지원 가능

▷ Coprocessor 레지스터는 모두 extended double precision이고 모든 내부 연산은 extended double precision로 실행됨

▷ 메모리와 레지스터 사이의 데이터 이동은 요청되는 precision로 실행

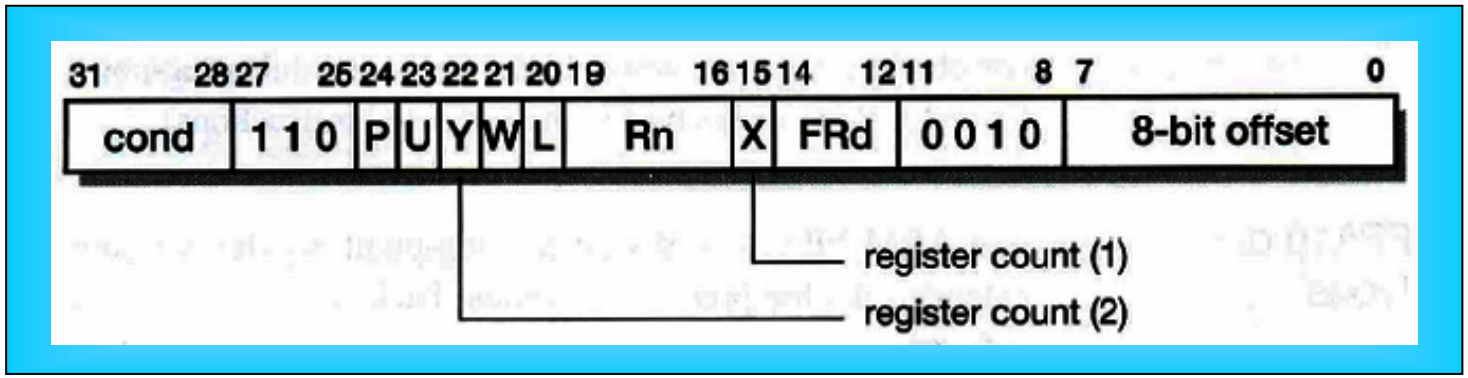
- 적재 저장 부동 소수점 명령어



▷ X, Y 비트를 이용하여 single, double, extended double, packed decimal로 구분, packed decimal과 extended packed decimal은 FPSR에서 구분

- 다중 적재 저장 부동 소수점 명령어

▷ 각 레지스터는 3 개의 메모리 워드에 저장, 데이터 형식은 정의되지 않음



▷ FRd는 전달되는 첫번째 레지스터를 나타내고 X, Y 비트는 전달되는 레지스터의 수를 나타냄 (최대 4개), 이 명령어는 coprocessor number 2를 사용하고 다른 명령어들은 coprocessor number 1을 사용함

- 데이터 처리 부동 소수점 명령어

▷ 간단한 산술 연산 명령 (덧셈, 뺄셈, 곱셈, 나눗셈, 나머지, Power)

▷ 초월 함수 실행 명령 (log, exponential, sin, cos, tan, arcsin, arctan)

▷ 기타 명령 (제곱근, move, 절대값, round)

- 부동 소수점 명령어 빈도

▷ 부동 소수점 emulator 소프트웨어를 사용하여 프로그램을 실행한 결과 load/store 명령어 빈도가 아주 높은 것으로 나타남

Instruction	Frequency
Load/store	67%
Add	13%
Multiply	10.5%
Compare	3%
Fix and float	2%
Divide	1.5%
Others	3%

▷ FPA 10은 load/store 동작이 내부 산술 연산과 동시에 동작하도록 설계됨

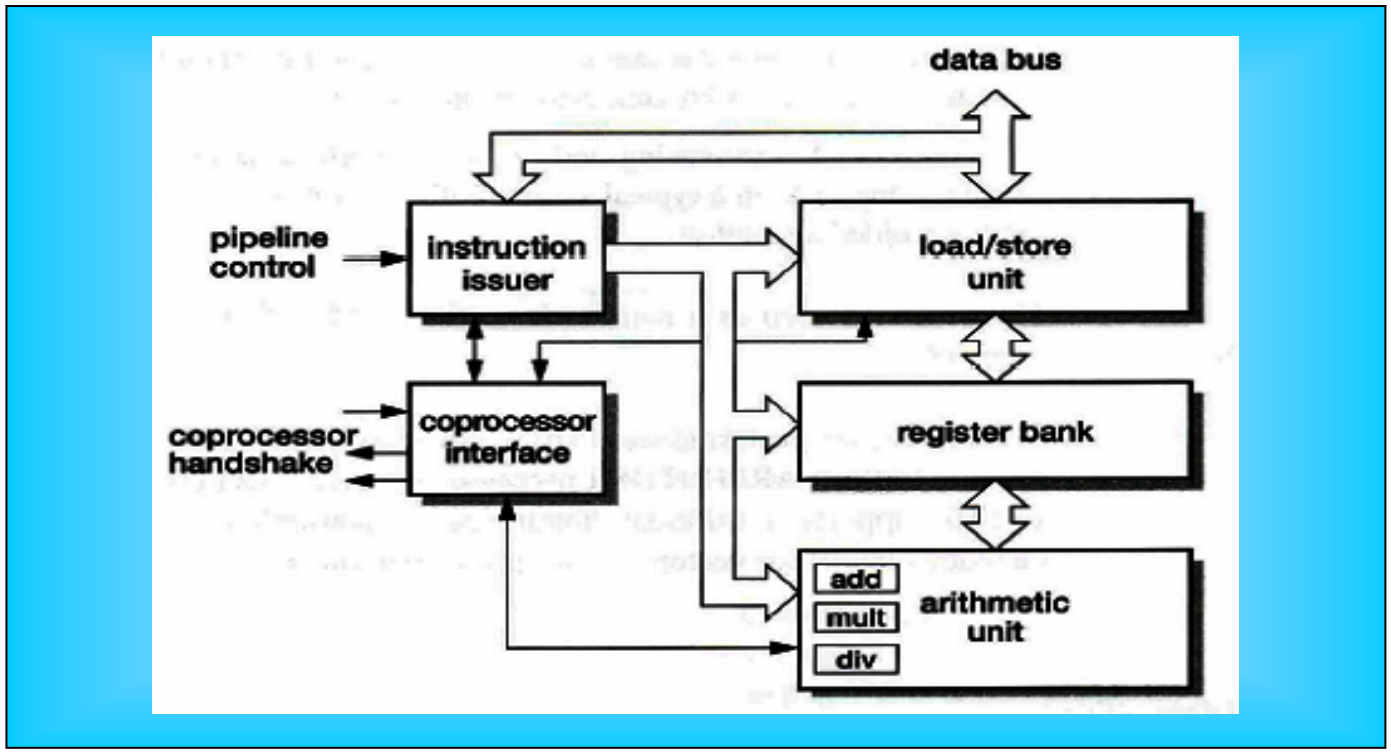
- FPA 10 조직

▷ 외부와의 인터페이스는 ARM 데이터 버스, handshake 신호들임

▷ load/store 장치는 데이터를 적재, 저장할 때 부동 소수점 형식으로 변환

▷ 산술 장치는 덧셈기, 곱셈기, divider, rounding, 정규화 하드웨어로 구성

▷ FPA 10의 내부 조직 구성도



- FPA 10 파이프라인

▷ 산술 연산은 4단계로 구성되며 명령어가 해석될 때부터 부동 소수점 동작은 실행되지만 결과 저장은 handshake 신호를 기다려야 함

▷ 산술 연산의 4 단계 파이프라인 단계

1. **Prepare: align operands.**
2. **Calculate: add, multiply or divide.**
3. **Align: normalize the result.**
4. **Round: apply appropriate rounding to the result.**

- 부동 소수점 context 스위치

▷ FPA 레지스터는 저장하고 복원해야 하는 추가적인 프로세서 상태를 나타냄

▷ 부동 소수점 명령을 사용하는 process는 많지 않기 때문에 FPA 레지스터를 저장하고 복원하는 횟수를 최소화하는 것이 필요함

▷ FPA를 사용하는 하나의 process가 종료되면 FPA 레지스터는 저장되지 않고 FPA를 turn off시킴, 만약 연속적인 process가 부동 소수점 명령을 실행한다면 trap이 발생하고 trap 코드는 FPA 상태를 저장하고 FPA를 동작시킴

▷ FPA를 사용하는 process만이 FPA 상태를 저장하고 복원하게 함

6.5 Expressions

- 레지스터 사용

▷ ARM 정수형 데이터 처리 명령어들은 C 정수형 산술 연산, 논리 연산, 자리 이동의 대부분을 직접 구현하지만 나눗셈은 몇 개의 명령어를 요구함

▷ 복잡한 표현의 평가를 위해 필요한 값들을 레지스터에서 차례로 얻고 자주 사용하는 값들을 레지스터에 유지하여야 함

▷ 레지스터에 저장할 값의 개수와 임시적인 결과를 저장할 레지스터의 개수 사이에는 trade-off가 존재하므로 컴파일러는 이러한 trade-off를 최적화 하는 것이 필요함

▷ ARM 3 주소 명령어 형식은 expression을 평가하는 동안 레지스터 유지와 재사용에 대한 최대한의 유연성을 제공함

▷ Thumb 명령어는 2 주소 명령어이므로 일반적인 레지스터를 더 작게 사용하여야 하고 이로 인해 덜 효율적인 코드가 생성됨

- 프로시져는 다음의 오퍼랜드를 사용하여 처리함

1. As an argument passed through a register.

   The value is already in a register, so no further work is necessary.

2. As an argument passed on the stack.

   Stack pointer (r13) relative addressing with an immediate offset known at compile-time allows the operand to be collected with a single LDR.

3. As a constant in the procedure's literal pool.

   PC-relative addressing, again with an immediate offset known at compile-time, gives access with a single LDR.

4. As a local variable.

   Local variables are allocated space on the stack and are accessed by a stack pointer relative LDR.

5. As a global variable.

   Global (and static) variables are allocated space in the static area and are accessed by static base relative addressing. The static base is usually in r9 (see the 'ARM Procedure Call Standard' on page 176).

- 포인터와 행렬 표현

▷ 포인터의 증가는 포인터가 가리키는 데이터의 크기에 따라 달라져야 함

▷ int *p; p=p+1인 경우 데이터가 정수형이므로 포인터는 +4 증가하여야 함

▷ 변수가 offset으로 사용되면 데이터 크기에 따라 offset은 scale되어야 함

```
int i = 4;
p = p + i;
```

If p is held in r0 and i in r1, the change to p may be compiled as:

```
ADD     r0, r0, r1, LSL #2 ; scale r1 to int
```

▷ 데이터의 크기가 2 지수승이 아닌 경우 덧셈과 자리 이동 명령을 사용

▷ 행렬인 경우 모든 요소들의 크기가 동일하므로 a[i]는 pointer-plus-offset *(a+i) 형태와 동일 (base plus scaled offset 어드레싱 모드가 가장 적합)

6.6 Conditional statements

▷ 조건이 참인 경우 실행되는 C의 문장으로 if..else와 switch 문장 등이 포함

- if...else 문장

▷ ARM 모든 명령어는 조건부 실행이 가능, 조건 분기 명령어를 사용하는 것 보다 더욱 효율적임

```
if (a>b) c=a; else c=b;

CMP     r0, r1
MOVGT   r2, r0
MOVLE   r2, r1
```

```
        CMP     r0, r1
        BLE     ELSE
        MOV     r2, r0
        B       ENDIF
ELSE    MOV     r2, r1
ENDIF   ..
```

- switch 문장

▷ 많은 경우를 가지는 switch 문장을 if..else 문장을 반복하여 구현한다면 코드는 길어지고 느린 코드가 됨

▷ 스위치 표현의 가능한 값에 대한 목표 주소를 포함하는 점프 테이블을 사용하여 구현하는 것이 합리적임

6.7 Loops

– for loops

▷ 컴파일러가 반복 횟수를 알 수 있는 경우에 사용하는 반복 문장

```
for (i=0; i<10; i++) {a[i] = 0}

        MOV     r1, #0              ; value to store in a[i]
        ADR     r2, a[0]            ; r2 points to a[0]
        MOV     r0, #0              ; i=0
LOOP    CMP     r0, #10             ; i<10 ?
        BGE     EXIT                ; if i >= 10 finish
        STR     r1, [r2,r0,LSL #2]; a[i] = 0
        ADD     r0, r0, #1          ; i++
        B       LOOP
EXIT    ..
```

▷ 조건 분기 명령어 BGE를 생략하고 다음 문장들을 반대 조건에 대한 조건부 실행 명령어로 바꾸면 더 효율적인 코드가 됨

▷ 테스트 명령을 루프 마지막으로 이동하면 더욱 효율적인 코드가 됨

- while loops

▷ 반복 횟수가 변수에 의해 정의되어 있어 컴파일러가 반복 횟수를 정확하게 모르는 경우 사용하는 반복 문장

```
LOOP    ..              ; evaluate expression
        BEQ     EXIT
        ..              ; loop body
        B       LOOP
EXIT    ..
```

```
        B       TEST
LOOP    ..              ; loop body
TEST    ..              ; evaluate expression
        BNE     LOOP
EXIT    ..
```

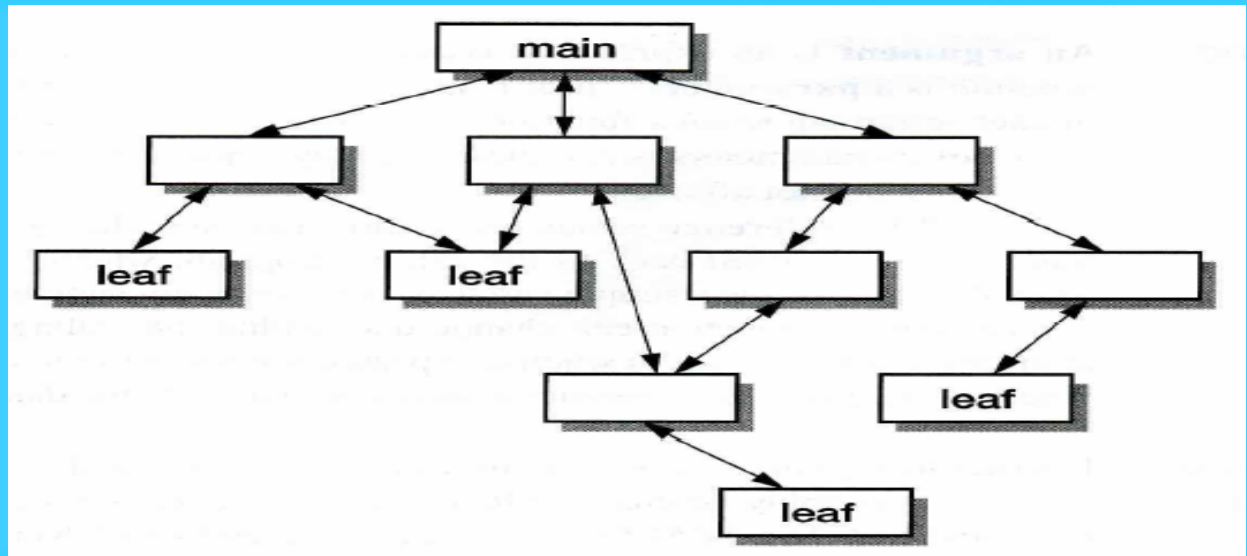
```
        ..              ; evaluate expression
        BEQ     EXIT    ; skip loop if necessary
LOOP    ..              ; loop body
TEST    ..              ; evaluate expression
        BNE     LOOP
EXIT    ..
```

- do...while loops : 최소한 한번은 루프가 실행되는 반복 문장

```
LOOP    ..              ; loop body
        ..              ; evaluate expression
        BNE     LOOP
EXIT    ..
```

6.8 함수와 프로시져

- 프로그램 구조 : 충분히 테스트될 수 있는 작은 부분들로 분리되어 구성되어야 함



- 함수, 서브루틴, 프로시져

- **Subroutine**: a generic term for a routine that is called by a higher-level routine, particularly when viewing a program at the assembly language level.
- **Function**: a subroutine which returns a value through its name. A typical invocation looks like:

```
c = max (a, b);
```

- **Procedure**: a subroutine which is called to carry out some operation on specified data item(s). A typical invocation looks like:

```
printf ("Hello World\n");
```

▷ 다른 언어와 달리 C 언어는 함수와 프로시져가 명확하게 구분되지 않음

▷ 값이 복귀하지 않고 side-effect만을 가지는 함수는 프로시져와 유사한 동작

▷ argument는 함수 요청에 의해 전달되는 표현이고 파라미터는 함수에 의해 수신된 값을 나타냄

▷ C언어는 'call by value'를 사용하고 함수가 요청될 때 argument는 복사됨

- APCS (ARM Procedure Call Standard)

▷ 여러 프로시져들의 효율적인 결합을 위해 프로시져 사용에 대한 규칙 필요

▷ 16개 레지스터는 프로시져 사용에서 특별한 목적으로 사용됨

Register	APCS name	APCS role
0	a1	Argument 1 / integer result / scratch register
1	a2	Argument 2 / scratch register
2	a3	Argument 3 / scratch register
3	a4	Argument 4 / scratch register
4	v1	Register variable 1
5	v2	Register variable 2
6	v3	Register variable 3
7	v4	Register variable 4
8	v5	Register variable 5
9	sb/v6	Static base / register variable 6
10	sl/v7	Stack limit / register variable 7
11	fp	Frame pointer
12	ip	Scratch reg. / new sb in inter-link-unit calls
13	sp	Lower end of current stack frame
14	lr	Link address / scratch register
15	pc	Program counter

▷ 전달해야 하는 인수가 많은 경우는 4개의 인수는 a1-a4에 전달하고 나머지는 역순으로 스택에 저장함

▷ 간단한 결과는 a1 레지스터를 통해 전달되고 복잡한 결과는 a1을 통해 전달되는 메모리 주소에 의해 메모리를 통해 전달됨

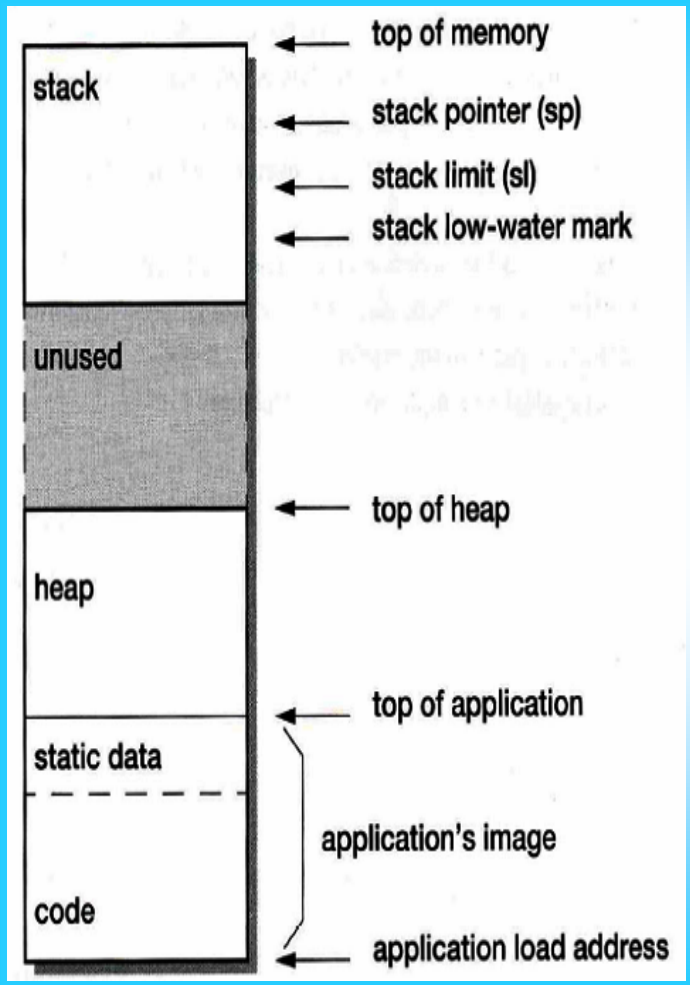
▷ 간단한 leaf 함수의 시작과 종료

```
        BL      leaf2
        ..
leaf2   STMFD   sp!, {regs, lr}    ; save registers
        ..
        LDMFD   sp!, {regs, pc}    ; restore and return
```

▷ 스택에 저장되고 복원되는 레지스터는 최소가 되어야 하고 복귀 주소가 스택에 저장되면 lr 레지스터는 임시적인 데이터를 저장하기 위해 사용될 수 있음

6.9 메모리 사용

– 주소 공간 모델



The stack

Whenever a (non-trivial) function is called, a new function frame is created on the stack containing a backtrace record, local variables, and so on. When a function returns its stack space is automatically recovered and will be reused for the next function call.

The heap

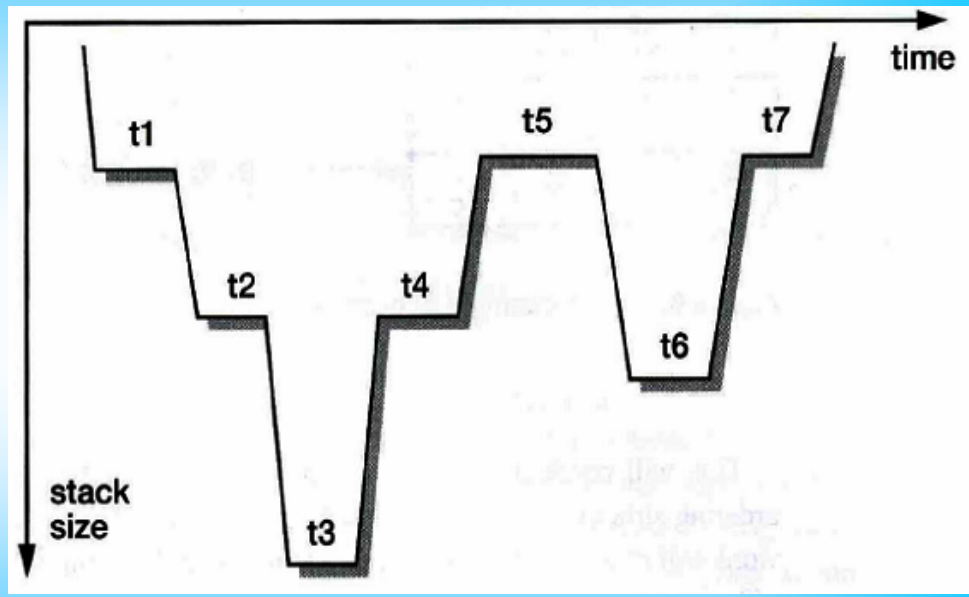
The heap is an area of memory used to satisfy program requests (malloc()) for more memory for new data structures. A program which continues to request memory over a long period of time should be careful to free up all sections that are no longer needed, otherwise the heap will grow until memory runs out.

▷ heap은 주소가 증가하는 방향으로 성장하고 stack은 주소가 감소하는 방향으로 성장, ARM은 하나의 응용에 대해 1 – 4G 바이트의 논리 공간을 할당

– 스택 동작

```
main () {
  ..                    /* t1 */
  func1 ();
  ..                    /* t5 */
  func2 ();
  ..                    /* t7 */
} /* end of main */
func1 () {
  ..                    /* t2 */
  func2 ();
  ..                    /* t4 */
} /* end of func1 */

func2 () {
  ..                    /* t3, t6 */
} /* end of func2 */
```

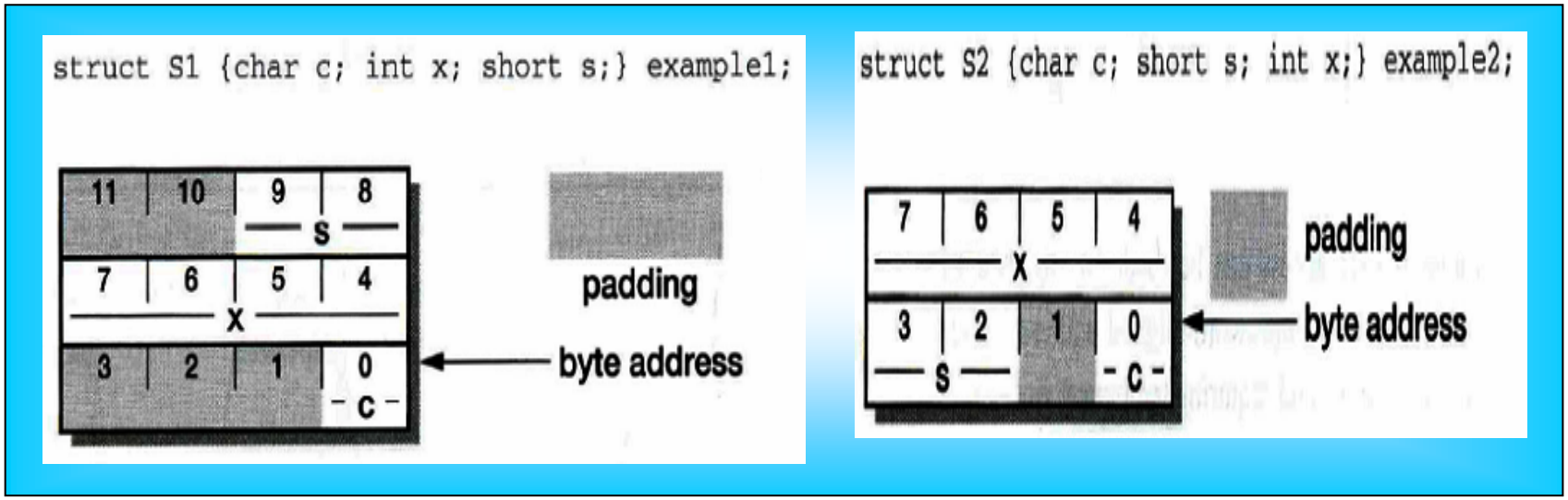


▷ 각 함수가 요청될 때 스택 공간은 레지스터에 전달되지 못한 인수, 함수 내에 사용하는 레지스터, 복귀 주소, 지역 변수들을 저장하기 위해 할당됨

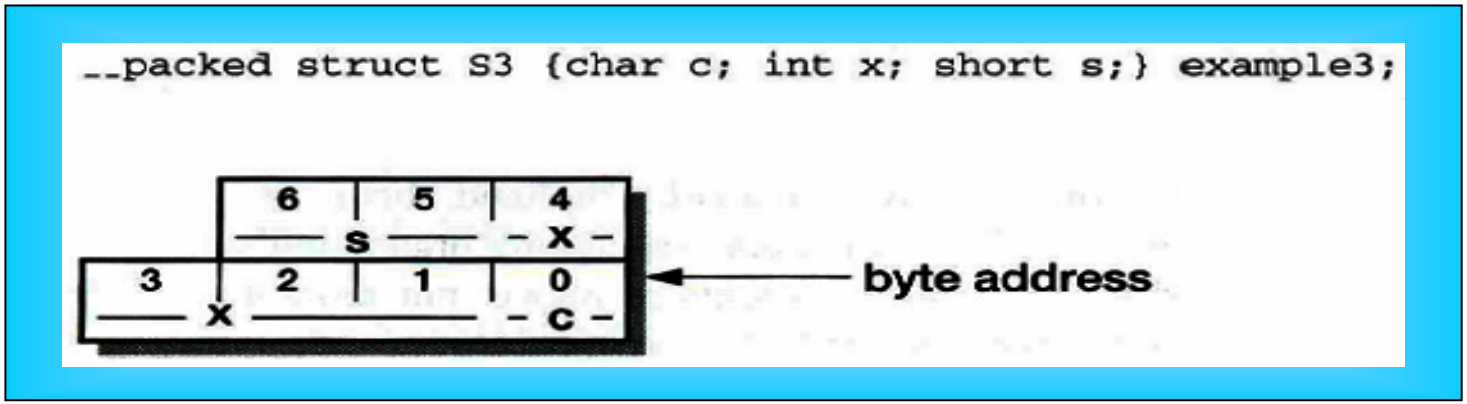
▷ 하나의 프로시져가 종료되면 사용된 지역 변수는 영원히 잃게 됨

- 데이터 alignment

▷ 다른 형태의 몇 개의 데이터 형태가 동시에 선언되면 컴파일러는 alignment를 위해 padding을 도입함, 워드 보다 작은 데이터 형태들을 워드 내로 그룹하면 메모리 효율성은 증가하고 padding 양도 감소됨



▷ ARM C 컴파일러는 모든 padding이 제거된 packed 데이터 구조를 가지는 코드를 생성할 수 있음



6.10 Run-time 환경

▷ C 프로그램은 ANSI C 라이브러리가 제공되는 환경을 요구함

▷ 작은 임베디드 시스템은 기본적인 C 함수의 구동을 위한 최소한의 stand-alone run-time 라이브러리를 제공 (나눗셈 함수들, 스택 제한 체크 함수들, 스택, heap 관리, 프로그램 시작과 종료 함수들)

▷ 이러한 최소한의 라이브러리를 위한 코드 크기는 736 바이트로 full ANSI C 라이브러리보다 매우 작음