# Module-4

Architectural Support for High-Level Languages: Abstraction in software design, Data types, Floating-point data types, The ARM floating-point architecture, Expressions,Conditional statements, Loops, Functions and procedures, Use of memory, Run-time environment, Examples and exercises.

# Architectural Support For High Level Languages

## Abstraction in software design

**Assembly-level abstraction**

Abstraction is important, then, at the assembly programming level, but all the responsibility for supporting the abstraction and expressing it in terms of the machine primitives rests with the programmer, who must therefore have a good understanding of those primitives and be prepared to return frequently to think at the level of the machine.

**High-level languages**

The job of supporting the abstractions used in the high-level language on the target architecture falls upon the *compiler*. Compilers are themselves extremely complex pieces of software, and the efficiency of the code they produce depends to a considerable extent on the support that the target architecture offers them to do their job.

## Data types

A computer data type can therefore be characterized by:

• the number of bits it requires;

• the ordering of those bits;

• the uses to which the group of bits is put.

**Numbers** The most basic category in the context of computation.

**Roman numerals** A number written by a human using alphabets which has a numerical value. Ex: V, XIX,MCMXCV etc

**Decimal numbers** The Roman numeral MXCV appears in decimal as:1995. Here we understand that the right-hand digit represents the number of units, the digit to its left the number of tens, then hundreds, thousands, and so on. Each time we move left one place the value of the digit is increased by a factor of 10.

**Binary coded decimal** to find a 4-bit representation for each digit.

**Binary notation** represents each decimal digit using 0 and 1.

**Hexadecimal notation** uses digits 0-9, A-F.

**Number ranges**

A computer usually reserves a fixed number of bits for a number, so if the number gets too big it cannot be represented. The ARM deals efficiently with 32-bit quantities, so the first data type that the architecture supports is the 32-bit (unsigned) integer, which has a value in the range:

0 to 4 294 967 $295_{10}$ = 0 to $\text{FFFFFFFF}_{16}$

**Signed integers**

Here the ARM supports a 2's complement binary notation where the value of the top bit is made negative; in a 32-bit signed integer all the bits have the same value as they have in the unsigned case. Now the range of numbers is:
$-2\,147\,483\,648_{10}$ to $+2\,147\,483\,647_{10} = 80000000_{16}$ to $7FFFFFFF_{16}$
The ARM, in common with most processors, uses the 2's complement notation for signed integers because adding or subtracting them requires exactly the same Boolean logic functions as are needed for unsigned integers, so there is no need to have separate instructions. The 'architectural support' for signed integers is the V flag in the program status registers which has no use when the operands are unsigned but indicates an **overflow** (out of range) error when signed operands are combined.

## Other number sizes

The natural representation of a number in the ARM is as a signed or unsigned 32-bit integer. Where a 32-bit integer is too small, larger numbers can be handled using multiple words and multiple registers. A 64-bit addition can be performed with two 32-bit additions, using the C flag in the status register to propagate the carry from the lower word to the higher word:

```
; 64-bit addition of [r1,r0] to [r3,r2]
ADDS  r2,  r2,  r0       ; add low, save carry
ADC  r3,  r3,  r1        ; add high with carry
```

## Real numbers

Real' numbers are used to represent fractions and transcendental values that are useful when dealing with physical quantities. An ARM core has no support for real data types, though ARM Limited has defined a set of types and instructions that operate on them. These instructions are either executed on a floating-point coprocessor or emulated in software.

### Printable characters

After the number, the next most basic data type is the printable character. To control a standard printer we need a way to represent all the normal characters such as the upper and lower case alphabet, decimal digits from 0 to 9, punctuation marks and a number of special characters such as £, $, %, and so on.

## ASCII

The normal way to store an ASCII character in a computer is to put the 7-bit binary code into an 8-bit byte. The most flexible way to represent characters is the 16-bit 'Unicode' which incorporates many such 8-bit character sets within a single encoding.

## ARM support for characters

The support in the ARM architecture for handling characters is the unsigned byte load and store instructions; There is nothing in the ARM architecture that reflects the particular codes defined by ASCII; any other encoding is equally well supported provided it uses no more than eight bits.

## Byte ordering

It is written to be read from left to right, but if it is read as a 32-bit word, the least significant byte is at the right. A character output routine might print characters at successive increasing byte addresses, in which case, with 'little-endian' addressing, it will print '5991'.

## High-level  languages

A high-level language defines the data types that it needs in its specification, usually without reference to any particular architecture that it may run on. Sometimes the number of bits used to represent a particular data type is architecture-dependent in order to allow a machine to use its most efficient size.

**ANSI C basic data types**

ANSI C defines the following basic data types:

• 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.**

Enumerated types (where variables have one value out of a specified set of possible values) are implemented as the smallest integer type with the necessary range of values. Bitfield types (sets of Boolean variables) are implemented within integers;

**ANSI C derived data types**

In addition, the ANSI C standard defines derived data types:

• **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 architectural support for C data types**

We have seen above that the ARM integer core provides native support for signed and unsigned 32-bit integers and for unsigned bytes, covering the C integer, long integer and unsigned character types. Pointers are implemented as native ARM addresses and are therefore supported directly.

The ARM addressing modes provide reasonable support for arrays and structures: base plus scaled index addressing allows an array of objects of a size which is 2" bytes to be scanned using a pointer to the start of the array and a loop variable as the index; base plus immediate offset addressing gives access to an object within a structure. However, additional address calculation instructions will be necessary for more complex accesses.

Current versions of the ARM include signed byte and signed and unsigned 16-bit loads and stores, providing some native support for short integer and signed character types.

## Floating-point data types

Floating-point numbers attempt to represent real numbers with uniform accuracy. A generic way to represent a real number is in the form:

$$R = a \times b^n$$

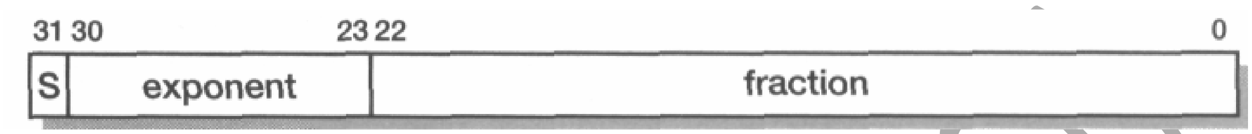
where *n* is chosen so that *a* falls within a defined range of values; *b* is usually implicit in the data type and is often equal to 2.

**IEEE 754**

There are many complex issues to resolve with the handling of floating-point numbers in computers to ensure that the results are consistent when the same program is run on different machines. The IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standard 754-1985, sometimes referred to simply as IEEE 754) which defines in considerable detail how floating-point numbers should be represented, the accuracy with which calculations should be performed, how errors should be detected and returned, and so on.

**Single precision**

The most compact representation of a floating-point number defined by IEEE 754 is the 32-bit 'single precision' format:



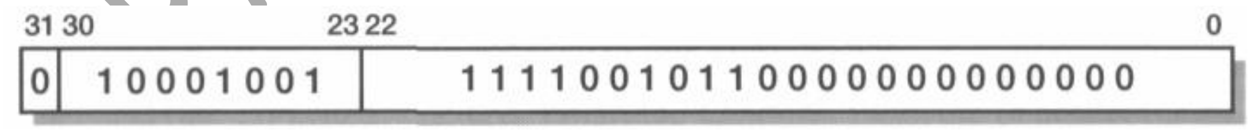
The number is made up from a **sign bit** ('S'), an **exponent** which is an unsigned integer value with a 'bias' of +127 (for normalized numbers) and a **fractional** component. A number of terms in the previous sentence may be unfamiliar. To explain them, let us look at how a number we recognize,' 1995', is converted into this format. We start from the binary representation of 1995, which has already been presented:11111001011 This is a positive number, so the S bit will be zero.

**Normalized numbers**

The first step is to **normalize** the number, which means convert it into the form shown in Equation 13 where 1 $<a<2$ and $b$ = 2. Looking at the binary form of the number, $a$ can be constrained within this range by inserting a 'binary point' (similar in interpretation to the more familiar decimal point) after the first' 1'. The implicit position of the binary point in the binary integer representation is to the right of the right-most digit, so here we have to move it left ten places.
Hence the normalized representation of 1995 is:
1995 = 1.1111001011 x $2^{10}$ Equation 14 where $a$ and $n$ are both in binary notation. When any number is normalized, the bit in front of the binary point in $a$ will be a ' 1' (otherwise the number is not normalized). Therefore there is no need to store this bit.

**Exponent bias**

Finally, since the format is required to represent very small numbers as well as very large ones, some numbers may require negative exponents in their normalized form.Rather than use a signed exponent, the standard specifies a 'bias'. This bias (+127 for single precision normalized numbers) is added to the exponent value. Hence 1995 is represented as:

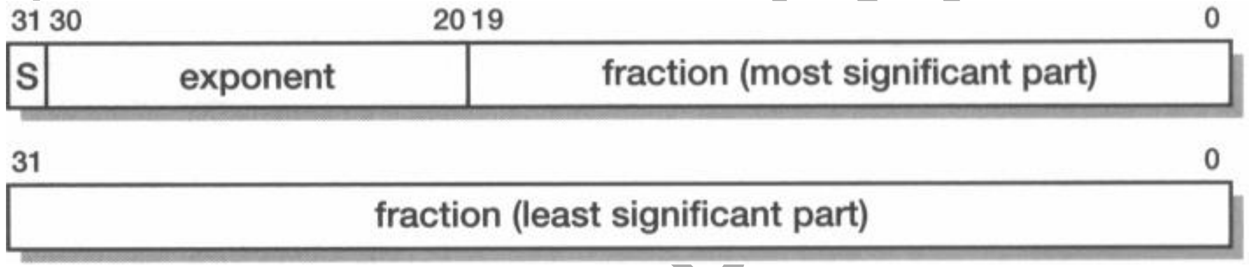


The exponent is 127+10 = 137; the fraction is zero-extended to the right to fill the 23-bit field.

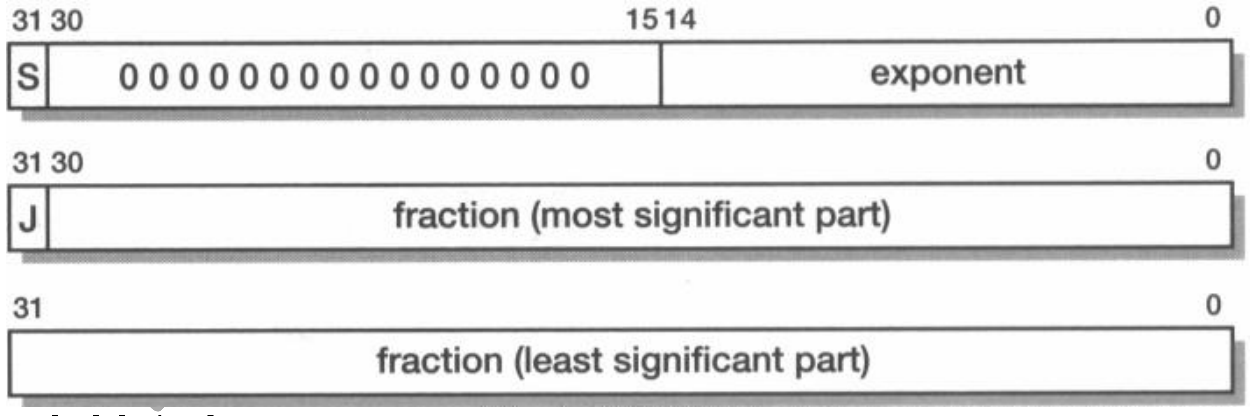
**Normalized value**

The IEEE 754 standard reserves numbers where the exponent is either zero or 255 to represent special values:
• Zero is represented by a zero exponent and fraction (but either sign value, so positive and negative zeros can be represented).
• Plus or minus infinity are represented by the maximum exponent value with a zero fraction and the appropriate sign bit.
• NaN (Not a Number) is indicated by the maximum exponent and a non-zero fraction; 'quiet' NaNs have a T in the most significant fraction bit position and 'signalling' NaNs have a '0' in that bit (but a T somewhere else, otherwise they look like infinity).
• Denormalized numbers, which are numbers that are just too small to normalize within this format, have a zero exponent, a non-zero fraction

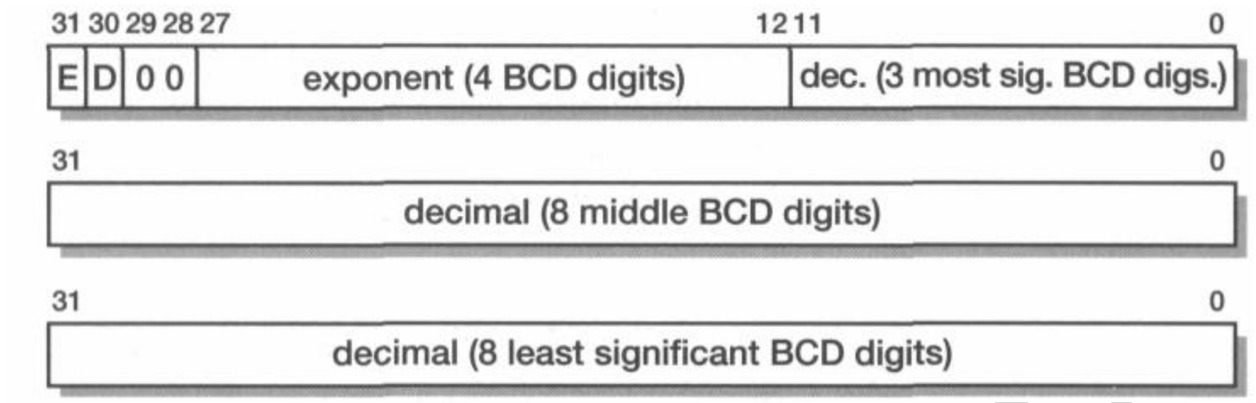
**Double precision**
For many purposes the accuracy offered by the single precision format is inadequate. Greater accuracy may be achieved by using the double precision format which uses 64 bits to store each floating-point value. The interpretation is similar to that for single precision values, but now the exponent bias for normalized numbers is +1023:


31 30
20 19
0
S
exponent
fraction (most significant part)
31
0
fraction (least significant part)

**Double extended precision**
Even greater accuracy is available from the double extended precision format, which uses 80 bits of information spread across three words. The exponent bias is 16383, and the J bit is the bit to the left of the binary point (and is a T for all normalized numbers):

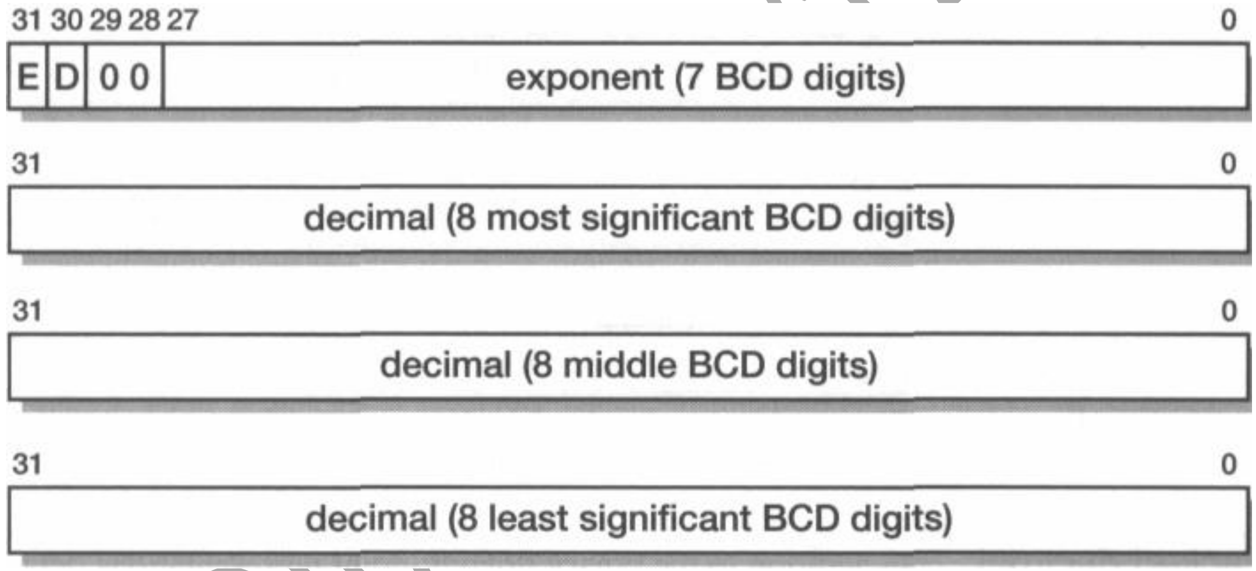

31 30
15 14
0
S
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
exponent
31 30
0
J
fraction (most significant part)
31
0
fraction (least significant part)

**Packed decimal**
In addition to the binary floating-point representations detailed above, the IEEE 754 standard also specifies packed decimal formats. Referring back to Equation 13 , in these packed formats *b* is 10 and *a* and *n* are stored in a binary coded decimal format as described in 'Binary coded decimal'. The number is normalized so that 1 < a < 10 . The packed decimal format is shown below:


31 30 29 28 27
12 11
0
E
D
0 0
exponent (4 BCD digits)
dec. (3 most sig. BCD digs.)
31
0
decimal (8 middle BCD digits)
31
0
decimal (8 least significant BCD digits)

The sign of the exponent is held in bit 31 of the first word ('E') and the sign of the decimal in bit 30 ('D').

**Extended packed decimal**
The extended packed decimal format occupies four words to give higher precision.


31 30 29 28 27
0
E
D
0 0
exponent (7 BCD digits)
31
0
decimal (8 most significant BCD digits)
31
0
decimal (8 middle BCD digits)
31
0
decimal (8 least significant BCD digits)

**ARM floatingpoint instructions**
Although there is no direct support for any of these floating-point data types in a standard ARM integer core, ARM Limited has defined a set of floating point instructions within the coprocessor instruction space. These instructions are normally implemented entirely in software through the undefined instruction trap (which collects any coprocessor instructions that are not accepted by a hardware coprocessor), but a subset may be handled in hardware by the FPA10 floating-point coprocessor.

**ARM floating point library –**
As an alternative to the ARM floating-point instruction set (and the only option for Thumb code), ARM Limited also supplies a C floating-point library which supports IEEE single and double precision formats. The C compiler has a flag to select this route which produces code that is both faster (by avoiding the need to intercept, decode and emulate the floating-point instructions) and more compact (since only the functions which are used need be included in the image) than software emulation.

# The ARM floating-point architecture

The ARM floating-point architecture presents:

• 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.

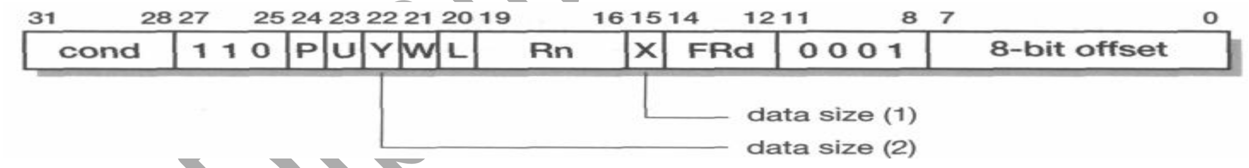
Note that the ARM coprocessor architecture allows the floating-point emulator (FPE) software to be used interchangeably with the combination of the FPA10 and the floating-point accelerator support code (FPASC), or any other hardware-software combination that supports the same set of instructions.

## FPAIO data types

The ARM FPA10 hardware floating-point accelerator supports single, double and extended double precision formats. Packed decimal formats are supported only by software. The coprocessor registers are all extended double precision, and all internal calculations are carried out in this, the highest precision format, except that there are faster versions of some instructions which do not produce the full 80-bit accuracy. However loads and stores between memory and these registers can convert the precision as required.

## Load and store floating instructions

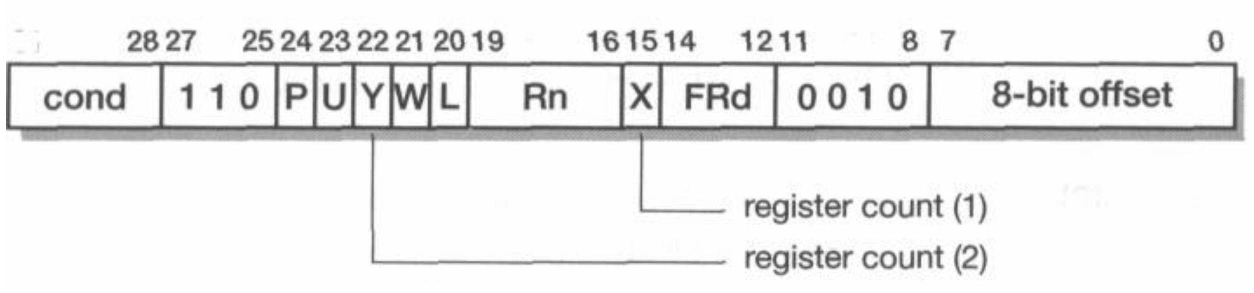
Since there are only eight floating-point registers, the register specifier field in the coprocessor data transfer instruction has a spare bit which is used here as an additional data size specifier:



The X and Y bits allow one of four precisions to be specified, choosing between single, double,double extended and packed decimal.

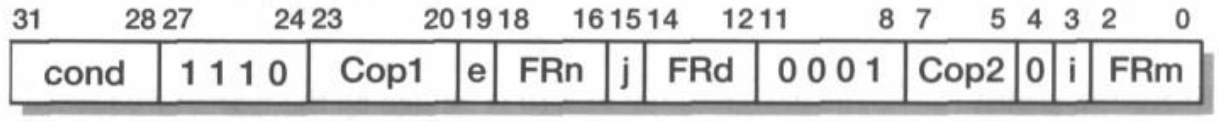
## Load and store multiple floating-point instruction

The load and store multiple floating-point registers instructions are used to save and restore the floating-point register state. Each register is saved using three memory words, 'FRd' specifies the first register to be transferred, and 'X' and 'Y' encode the number of registers transferred which can be from one to four. Note that these instructions use coprocessor number 2, whereas the other floating-point instructions use coprocessor number 1.



## Floating-point data operations

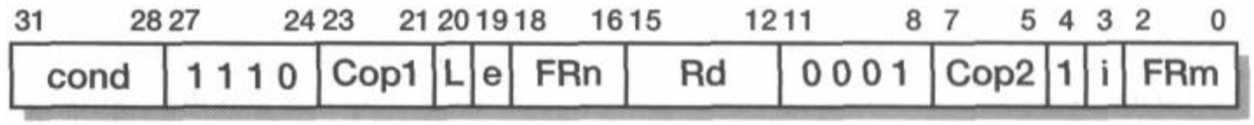
The floating-point data operations perform arithmetic functions on values in the floating point registers; their only interaction with the outside world is to confirm that they should complete through the ARM coprocessor handshake.



The instruction format has a number of opcode bits, augmented by extra bits from each of the three register specifier fields since only three bits are required to specify one of the eight floating-point registers:
• 'i' selects between a register ('FRm') or one of eight constants for the second operand.
• 'e' and 'Cop2' control the destination size and the rounding mode.
• 'j' selects between monadic (single operand) and dyadic (two operand) operations.
The instructions include simple arithmetic operations (add, subtract, multiply,divide, remainder, power), transcendental functions (log, exponential, sin, cos, tan,arcsin, arccos, arctan) and assorted others (square root, move, absolute value, round).

## Floating-point register transfer instructions

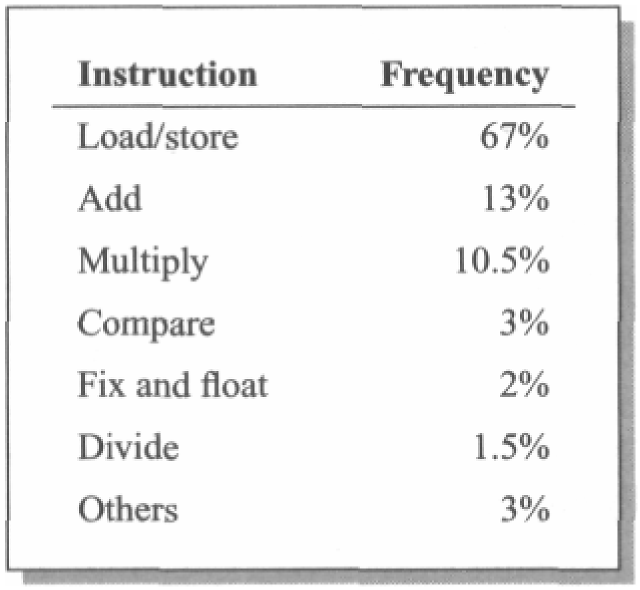
Floating-point register transfer instructions accept a value from or return a value to an ARM register transfers register. This is generally combined with a floating-point processing function.



Transfers from ARM to the floating-point unit include 'float' and writes to the floating-point status and control registers; going the other way there is 'fix' and reads of the status and control registers. The floating-point compare instructions are special cases of this instruction type, where Rd is r15. Two floating-registers are compared and the result of the comparison is returned to the N, Z, C and V flags in the ARM CPSR where they can directly control the execution of conditional instructions:
• N indicates 'less than'.
• Z indicates equality.
• C and V indicate more complex conditions, including 'unordered' comparison results which can arise when an operand is a 'NaN' (not a number).

## Floating-point instruction frequencies

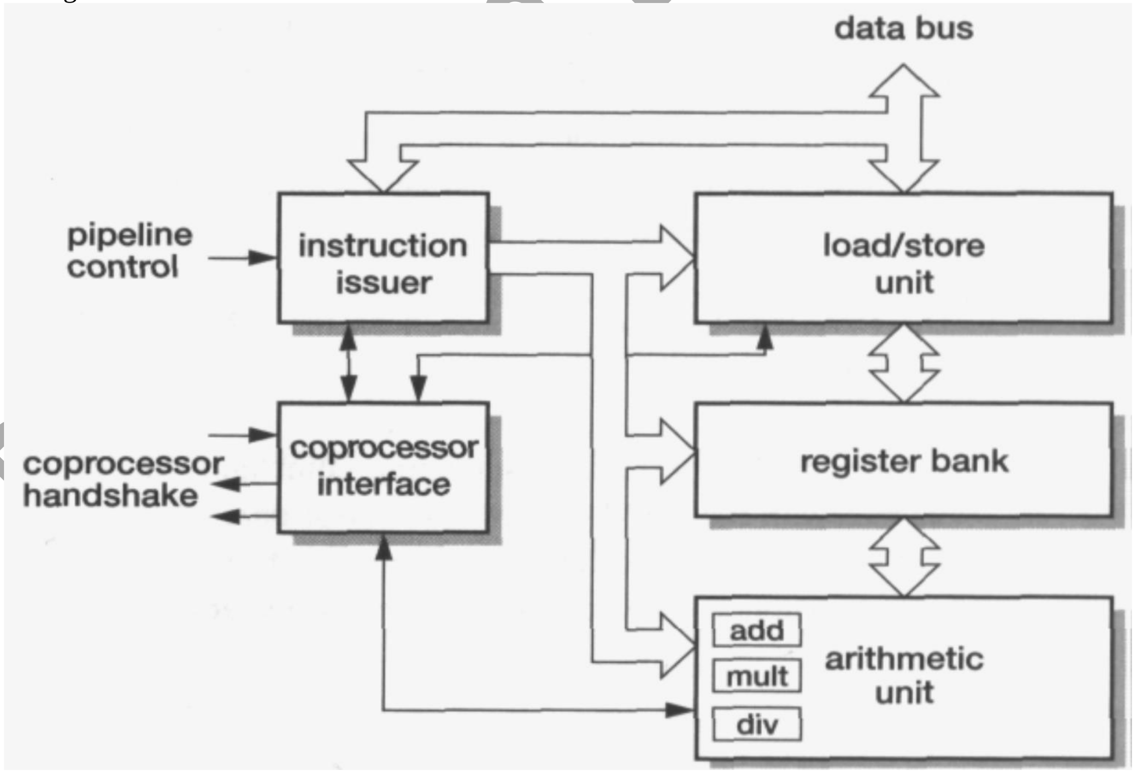
The design of the FPAIO is guided by the typical frequencies of the various floatingpoint instructions. These were measured running compiled programs using the floating-point emulator software and are summarized in Table below.
The statistics are dominated by the number of load and store operations that take place. As a result, the FPAIO has been designed to allow these to operate concurrently with internal arithmetic operations.



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

## FPA10 Organization

The major components are:
• The coprocessor pipeline follower
• The load/store unit that carries out format conversion on floating-point data types as they are loaded from and stored to memory.
• The register bank which stores eight 80-bit extended precision floating-point operands.
• The arithmetic unit which incorporates an adder, a multiplier and a divider,together with rounding and normalizing hardware.



The load/store unit operates concurrently with the arithmetic unit, enabling new operands to be loaded from memory while previously loaded operands are being processed. Hardware interlocks protect against data hazards.

**FPA10 Pipeline**

The FPA10 arithmetic unit operates in four pipeline stages:

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

A floating-point operation can begin as soon as it is detected in the instruction pipeline (that is, before the ARM handshake has occurred), but the result write-back must await the handshake.

**Floating point context switches**

The FPA registers represent additional process state which must be saved and restored across context switches. However, typically only a small number of active processes use floating-point instructions. Therefore saving and restoring the FPA registers on every switch is an unnecessary overhead. Instead, the software minimizes
the number of saves and restores by the following algorithm:

• When a process using the FPA is switched out, the FPA registers are not saved but the FPA is turned off.

• If a subsequent process executes a floating-point instruction this will trap; the trap code will save the FPA state and enable the FPA.

**FPA10 applications**

The FPA10 is used as a macrocell on the ARM7500FE chip.

# Expressions

The basic ARM integer data processing instructions implement most of the C integer arithmetic, bit-wise and shift primitives directly. Exceptions are division and remainder which require several ARM instructions.

**Register use**

Since all data processing instructions operate only on values in register, the key to the efficient evaluation of a complex expression is to get the required values into the registers in the right order and to ensure that requently used values are normally resident in registers.

**ARM support**

The 3-address instruction format used by the ARM gives the compiler the maximum flexibility in how it preserves or re-uses registers during expression evaluation. Thumb instructions are generally 2-address, which restricts the compiler's freedom to some extent, and the smaller number of general registers also makes its job harder.

**Accessing operands**

A procedure will normally work with operands that are presented in one of the following ways, and can be accessed as indicated:

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.

**Pointer arithmetic**
If the value that is passed is a pointer, an additional LDR (with an immediate offset) may be required to access an operand within the structure that it points to. Arithmetic on pointers depends on the size of the data type that the pointers are pointing to. If a pointer is incremented it changes in units of the size of the data item in bytes.
Thus:

```
int *p;
P = P + l;
```

will increase the value of p by 4 bytes. Since the size of a data type is known at compile-time, the compiler can scale constants by an appropriate amount. If a variable is used as an offset it must be scaled at run-time:

```
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
```

Where the data type is a structure with a size which is not a power of 2 bytes, a multiplication by a small constant is required. The shift and add instructions can usually produce the desired product in a small number of operations, using a temporary register where necessary.

**Arrays**
Arrays in C are little more than a shorthand notation for pointer operations, so the above comments apply here too. The declaration:

```
int a[10];
```

establishes a name, a, for the array which is just a pointer to the first element, and a reference to a [ i ] is equivalent to the pointer-plus-offset form * (a+ i); the two may be used interchangeably.

## Conditional statements
Conditional statements are executed if the Boolean result of a test is true (or false);in C these include if...else statements and switches (C 'case' statements).
**if...else**

The ARM architecture offers unusually efficient support for conditional expressions when the conditionally executed statement is small.For example, here is a C statement to find the maximum of two integers:

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

If the variables a, b and c are in registers r0, r1 and r2, the compiled code could be as simple as:

```
CMP    r0, r1    ; if (a>b)...

MOVGT          r2, r0    ;c=a
```

```
MOVLE         r2, rl    ; else c=b
```

The 'if and 'else' sequences may be a few instructions long with the same condition on each instruction (provided none of the conditionally executed instructions changes the condition codes), but beyond two or three instructions it is generally better to fall back on the more conventional solution:

```
                CMP r0, rl            ; if (a>b)...

                BLE    ELSE           ; skip clause if false

                MOV    r2, r0         ; ..c=a..

                B      ENDIF           ; skip else clause

 ELSE           MOV    r2, rl         ; ...else c=b

 ENDIF
```

Here the 'if and 'else' sequences may be any length and may use the condition codes freely (including, for example, for nested if statements) since they are not required beyond the branch immediately following the compare instruction.

**switches**

Branches are expensive on the ARM, so the absence of them from the first sequence makes it very efficient. A switch, or case, statement extends the two-way decision of an if...else statement to many ways. The standard C form of a switch statement is:

```
switch (expression) {
case constant-expression]: statements
case constant-expressio^: statementS2
…..
case constant-expression^: statements^
default: statements^
}
```

Normally each group of statements ends with a 'break' (or a 'return') to cause the switch statement to terminate, otherwise the C semantics cause execution to fall through to the next group of statements.

However this can result in slow code if the switch statement has many cases. An alternative is to use *a jump table.* In its simplest form a jump table contains a target address for each possible value of the switch expression:

```
                ; r0 contains value of expression
        ADR     r1, JUMPTABLE       ; get base of jump table
        CMP     r0, #TABLEMAX       ; check for overrun..
        LDRLS   pc, [r1,r0,LSL #2]  ;   .. if OK get pc
                ; statements_D      ;   .. otherwise default
        B       EXIT                ; break
L1      ..      ; statements_1
        B       EXIT                ; break
        ..
LN      ..      ; statements_N
EXIT    ..
```

## Loops

The C language supports three forms of loop control structure:

• for(el;e2;e3){..}
• while (el) {..}
• do {..} while (el)

Here el, e2 and e3 are expressions which evaluate to 'true' or 'false' and {..} is the body of the loop which is executed a number of times determined by the control structure. Often the body of a loop is very simple, which puts the onus on the compiler to minimize the overhead of the control structure. Each loop must have at least one branch instruction, but any more would be wasteful. for loops A typical 'for' loop uses the control expressions to manage an index:

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

The first control expression is executed once before the loop begins, the second is tested each time the loop is entered and controls whether the loop is executed, and the third is executed at the end of each pass through the loop to prepare for the next pass. This loop could be compiled as:

```
        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    ..
```

### while loops

The standard conceptual arrangement of a 'while' loop is as follows:

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

The two branches appear to be necessary since the code must allow for the case where the body is not executed at all. A little reordering produces more efficient code:

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

```
BNE LOOP
EXIT
```

This second code sequence executes the loop body and evaluates the control expression exactly the same way as the original version and has the same code size, but it executes one fewer (untaken) branch per iteration, so the second branch has been removed from the loop overhead. An even more efficient code sequence can be produced by the compiler:

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

The saving here is that one fewer branch is executed each time the complete 'while' structure is encountered (assuming that the body is executed at least once). This is a modest gain and costs extra instructions, so it is worthwhile only if performance matters much more than code size.

### do..while loops

The conceptual arrangement of a 'do..while' loop is similar to the improved 'while' loop above, but without the initial branch since the loop body is executed before the test (and is therefore always executed at least once):

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

## Functions and procedures

### Program design

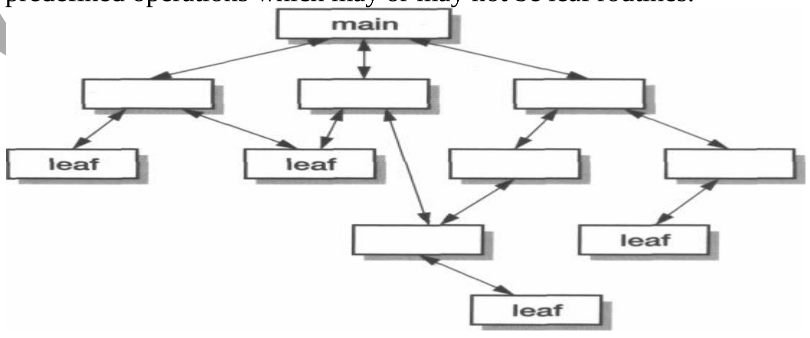
Good programming practice requires that large programs are broken down into components that are small enough to be thoroughly tested; a large, monolithic program is too complex to test fully and is likely to have 'bugs' in hidden corners that do not emerge early enough in the program's life to be fixed before the program is shipped to users. Each small software component should perform a specified operation using a well-defined interface. How it performs this operation should be of no significance to the rest of the program

### Program hierarchy

Furthermore, the full program should be designed as a **hierarchy** of components, not simply a flat list.A typical hierarchy is illustrated in Figure below. The top of the hierarchy is the program called *main.* The remaining hierarchy is fairly informal; lower-level routines may be shared by higher-level routines, calls may skip levels, and the depth may vary across the hierarchy.

### Leaf routines

At the lowest level of the hierarchy there are **leaf** routines; these are routines which do not themselves call any lower-level routines. In a typical program some of the bottom-level routines will be **library or system** functions; these are predefined operations which may or may not be leaf routines.



### Terminology

There are several terms that are used to describe components of this program structure, often imprecisely. We shall attempt to apply terms as follows:
• **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 WorldXn");
```

### C functions
Some programming languages make a clear distinction between functions and procedures, but C does not. In C all subroutines are functions, but they can have side-effects in addition to returning a value, and when the returned value is of type 'void' it is effectively suppressed and only the side-effects remain, giving a behaviour which looks just like a procedure.

### Arguments and parameters
An **argument** is an expression passed to a function call; a value received by the function is a **parameter.** C uses a strict 'call by value' semantics, so a copy is made of each argument when a function is called and, though the function may change the values of its parameters, since these are only copies of the arguments the argumentsthemselves are not affected.
The way a C function can change data within the calling program, other than by returning a single value, is when it is passed a pointer to the data as an argument. The function can then use the pointer to access and modify the data structure.

## ARM Procedure Call Standard
The ARM Procedure Call Standard (APCS) is employed by the ARM C compiler, though this is of significance to the C programmer only when the assembly-level output must be understood in detail.

### APCS register usage
• It defines particular uses for the 'general-purpose' registers.
• It defines which form of stack is used from the full/empty, ascending/descending choices supported by the ARM instruction set.
• It defines the format of a stack-based data structure used for back-tracing when debugging programs.
• It defines the function argument and result passing mechanism to be used by all externally visible functions and procedures.
• It supports the ARM shared library mechanism, which means it supports a standard way for shared code to access static data. The convention for the use of the 16 currently visible ARM registers is summarized below:
The registers are divided into three sets:
APCS register use convention.

| Register | APCS name | APCS role |
|---|---|---|
| 0 | al | 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 | vl | 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 | Ir | Link address / scratch register |
| 15 | pc | Program counter |

## APCS variants

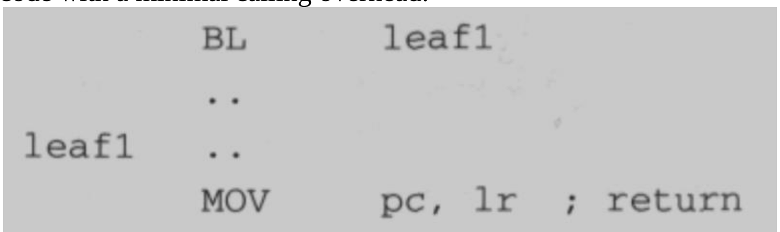
1. Four argument registers which pass values into the function.
The function need not preserve these so it can use them as scratch registers once it has used or saved its parameter values. Since they will not be preserved across any calls this function makes to other functions, they must be saved across such calls if they contain values that are needed again. Hence, they are **caller-saved** register variables when so used.
2. Five (to seven) register variables which the function must return with unchanged values. These are **callee-saved** register variables. This function must save them if it wishes to use the registers, but it can rely on functions it calls not changing them.
3. Seven (to five) registers which have a dedicated role, at least some of the time. The link register (Ir), for example, carries the return address on function entry, but if it is saved (as it must be if the function calls subfunctions) it may then be used as a scratch register.
There are several (16) different variants of the APCS which are used to generate code for a range of different systems. They support:
• 32-or 26-bit PCs.
Older ARM processors operated in a 26-bit address space and some later versions continue to support this for backwards compatibility reasons.
• Implicit or explicit stack-limit checking.
Stack overflow must be detected if code is to operate reliably. The compiler can insert instructions to perform explicit checks for overflow. Where memory management hardware is available, an ARM system can allocate memory to the stack in units of a page. If the next logical page is mapped out, a stack overflow will cause a data abort and be detected. Therefore the memory management unit can perform stack-limit checking and there is no need for the compiler to insert instructions to perform explicit checks.
• Two ways to pass floating-point arguments.
The ARM floating-point architecture specifies a set of eight floating-point registers. The APCS can use these to pass floating-point arguments into functions, and this is the most efficient solution when the system makes extensive use of floating-point variables. If, however, the system makes little or no use of floating-point types this approach incurs a small overhead which is avoided by passing floating-point arguments in the integer registers and/or on the stack.
• Re-entrant or non-re-entrant code.
Code specified as re-entrant is position-independent and addresses all data indirectly through the static base register (sb). This code can be placed in a ROM and can be shared by several client processes. Generally, code to be placed in a ROM or a shared library should be re-entrant whereas application code will not be.

## Argument Passing

A C function may have many  arguments. The APCS organizes the arguments as follows:
1. If floating-point values are passed through floating-point registers, the first four floating-point arguments are loaded into the first four floating-point registers.
2. All remaining arguments are organized into a list of words; the first four words are loaded into al to a4, then the remaining words are pushed onto the stack in reverse order.
Note that multi-word arguments, including double precision floating-point values, may be passed in integer registers, on the stack, or even split across the registers and the stack.

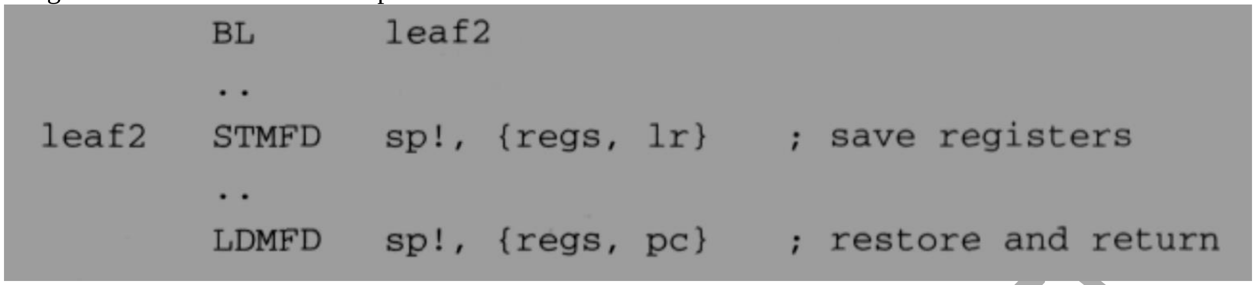
### Result return Function entry and exit

A simple result (such as an integer) is returned through a1. A more complex result is returned in memory to a location specified by an address which is effectively passed as an additional first argument to the function through al. A simple leaf function which can perform all its functions using only al to a4 can be compiled into code with a minimal calling overhead:



```
        BL      leaf1
        ..
leaf1   ..
        MOV     pc, lr  ; return
```

In typical programs somewhere around 50% of all function calls are to leaf functions,and these are often quite simple. Where registers must be saved, the function must create a stack frame. This can be compiled efficiently using ARM's load and store multiple instructions:



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

Here the number of registers which are saved and restored will be the minimum required to implement the function. Note the value saved from the link register (Ir) is returned directly to the program counter (pc), and therefore Ir is available as a scratch register in the body of the function (which it was not in the simple case above).

### Tail continued functions

More complex function entry sequences are used where needed to create stack backtrace data structures, handle floating-point arguments passed in floating-point registers, check for stack overflow, and so on. Simple functions that call another function immediately before returning often do not incur any significant call overhead; the compiler will cause the code to return directly from the continuing function. This makes veneer functions (functions that simply reorder arguments, change their type or add an extra argument) particularly efficient.

### ARM efficiency

Overall the ARM supports functions and procedures efficiently and flexibly. The various flavours of the procedure call standard match different application requirements well, and all result in efficient code. The load and store multiple register instructions are exploited to good effect by the compiler in this context; without them calls to non-leaf functions would be much more costly. The compiler is also able to make effective optimizations for leaf and tail continued functions, thereby encouraging good programming styles. Programmers can use better structured design when leaf function calls are efficient and can exploit abstraction better when veneer functions are efficient.

## Use of memory

An ARM system, like most computer systems, has its memory arranged as a linear set of logical addresses. A C program expects to have access to a fixed area of program memory (where the application image resides) and to memory to support two data areas that grow dynamically and where the compiler often cannot work out a maximum size. These dynamic data areas are:

• The stack.

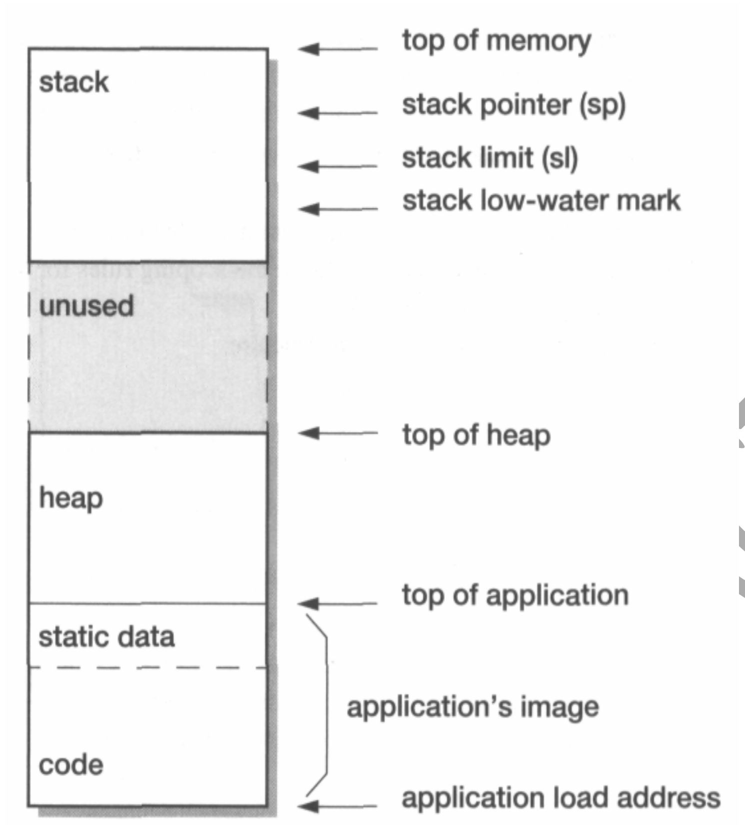
Whenever a (non-trivial) function is called, a new activation frame is created on the stack containing a backtrace record, local (non-static) variables, and so on.When a function returns its stack space is automatically recovered and will be reused for the next function call.

• 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.

### Address space model

The normal use of memory is shown in figure below where an application can use the entire memory space (or where a memory management unit can allow an application to think it has the entire memory space), the application image is loaded into the lowest address, the heap grows upwards from the top of the application and the stack grows downwards from the top of memory.



The unused memory between the top of the heap and the bottom of the stack is allocated on demand to the heap or the stack, and if it runs out the program stops due to lack of memory.

In a typical memory managed ARM system the logical space allocated to a single application will be very large, in the range of 1 to 4 Gbytes. The memory management unit will allocate additional pages, on demand, to the heap or the stack, until it runs out of pages to allocate This will usually be a long time before the top of the heap meets the bottom of the stack.

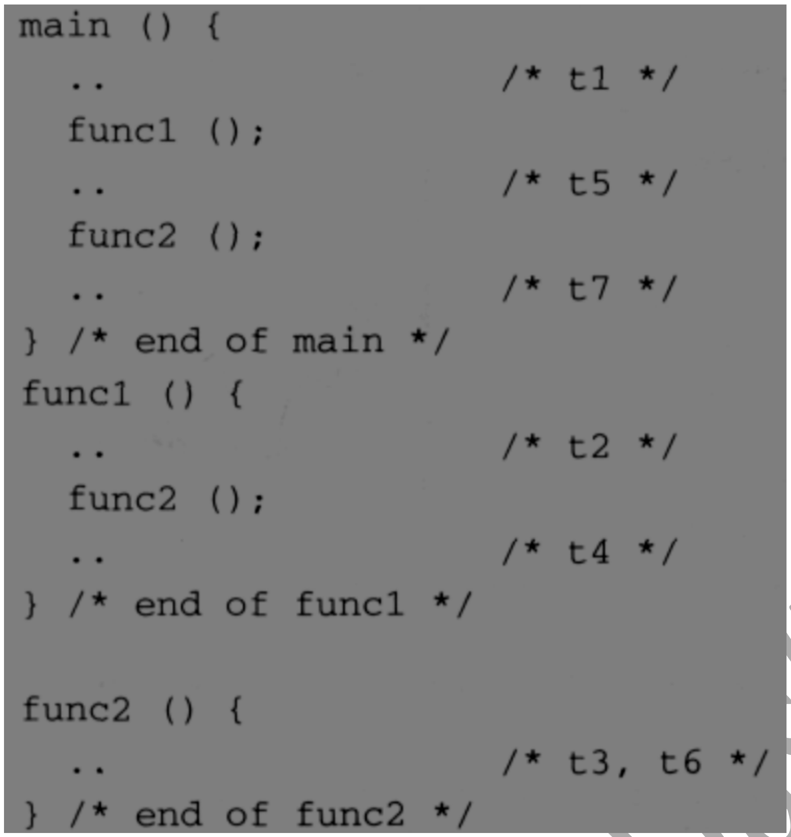
In a system with no memory management support the application will be allocated all (if it is the only application to run at the time) or part (if more than one application is to run) of the physical memory address space remaining once the operating system has had its requirements met, and then the application runs out of memory precisely when the top of the heap meets the bottom of the stack.

## Chunked stack model

Other address space models are possible, including implementing a 'chunked' stack where the stack is a series of chained chunks within the heap. This causes the application to occupy a single contiguous area of memory, which grows in one direction as required, and may be more convenient where memory is very tight.

## Stack behaviour

It is important to understand the dynamic behaviour of the stack while a program is running, since it sheds some light on the scoping rules for local variables (which are allocated space on the stack).
Consider this simple program structure:



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

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

Assuming that the compiler allocates stack space for each function call, the stack behaviour will be as shown in Figure above. At each function call, stack space is allocated for arguments, to save registers for use within the function, to save the return address and the old stack pointer, and to allocate
memory on the stack for local variables.
Note how all the stack space is recovered on function exit and reused for subsequent calls, and how the two calls to func2 () are allocated different memory areas (at times t3 and t6 in Figure 6.14 on page 183), so even if the memory used for a local variable in the first call had not been overwritten in an intervening call to another function, the old value cannot be accessed in the second call since its address is not known. Once a procedure has exited, local variable values are lost forever.

## Data storage

The various data types supported in C require differing amounts of memory to store their binary representations. The basic data types occupy a byte (chars), a half-word (short ints), a word (ints, single precision float) or multiple words (double precision floats). Derived data types (structs, arrays, unions, and so on) are defined in terms of multiple basic data types.
The ARM instruction set, in common with many other RISC processors, is most efficient at loading and storing data items when they are appropriately aligned in memory.

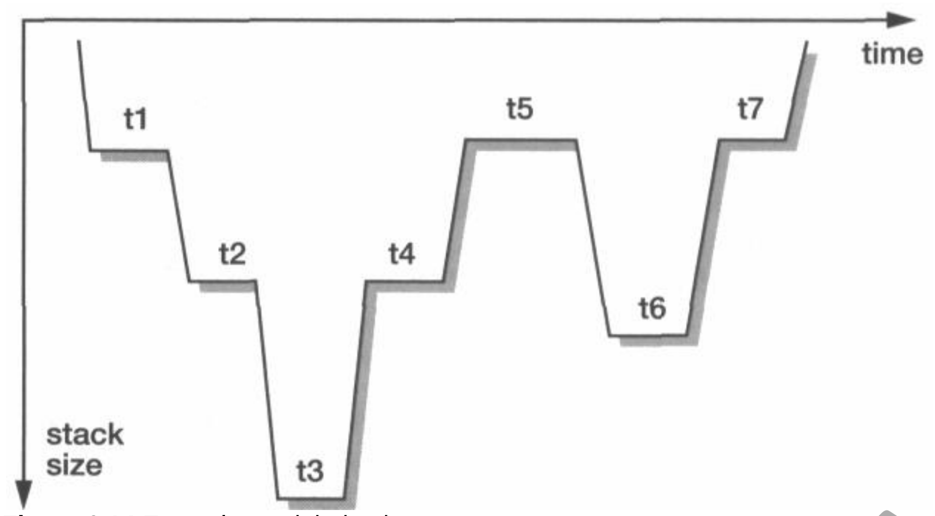


**Figure 6.14** Example stack behaviour.

A byte access can be made to any byte address with equal efficiency, but storing a word to a non-word-aligned address is very inefficient, taking up to seven ARM instructions and requiring temporary work registers.

## Data alignment

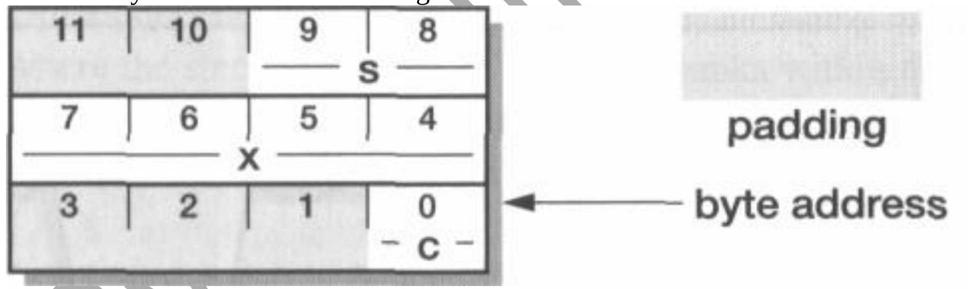
Therefore the ARM C compiler generally aligns data items on appropriate boundaries:

• Bytes are stored at any byte address.
• Half-words are stored at even byte addresses.
• Words are stored on four-byte boundaries.

Where several data items of different types are declared at the same time, the compiler will introduce **padding** where necessary to achieve this alignment:

```
struct SI {char c; int x; short s;} examplel;
```

This structure will occupy three words of memory as shown in Figure below. Arrays are laid out in memory by repeating the appropriate basic data item, obeying the alignment rules for each item. Given the data alignment rules outlined above, the programmer can help the compiler to minimize memory wastage by organizing structures appropriately. A structure with the same contents as above, but reordered as below, occupies only two memory words instead of the original three:



struct *S2* {char c; short s; int x;} example2;

This will result in the memory occupancy illustrated in Figure below. In general, ordering structure elements so that types smaller than a word can be grouped within a word will minimize the amount of padding that the compiler has to insert to maintain efficient alignment.

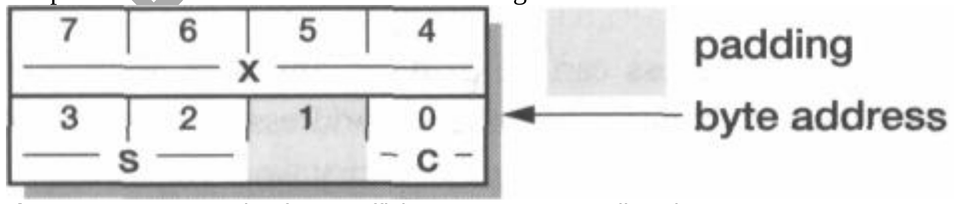
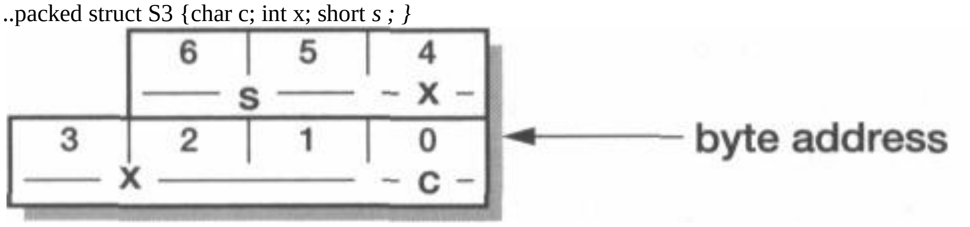


**Figure 6.16** An example of more efficient struct memory allocation.

**Packed structure**
Sometimes it is necessary to exchange data with other computers that follow different alignment conventions, or to pack data tightly to minimize memory use even though this will reduce performance. For such purposes the ARM C compiler can produce code that works with packed data structures where all the padding is removed:
..packed struct S3 {char c; int x; short *s ; }*



# Run-time environment

## Minimal run-time library

A C program requires an environment in which to operate; this is usually provided through a library of functions that the C program can call. In a PC or workstation a C programmer can expect to find the full ANSI C library, giving access to a broad range of functions such as file management, input and output (print f ()), the realtime clock, and so on.
In a small embedded system such as a mobile telephone, most of these functions are irrelevant. ARM Limited supplies a minimal stand-alone run-time library which, once ported to the target environment, allows basic C programs to run. This library therefore reflects the minimal requirements of a C program. It comprises:
• Division and remainder functions.
Since the ARM instruction set does not include divide instructions, these are implemented as library functions.
• Stack-limit checking functions.
A minimal embedded system is unlikely to have memory management hardware available for stack overflow detection; therefore these library functions are needed to ensure programs operate safely.
• Stack and heap management.
All C programs use the stack for (many) function calls, and all but the most trivial create data structures on the heap.
• Program start up.
Once the stack and heap are initialized, the program starts with a call to main ().
• Program termination.
Most programs terminate by calling _exit (); even a program which runs forever should terminate if an error is detected.

**Example 6.1 Write, compile and run a 'Hello World' program written in C.**
The following program has the required function:

```
/* Hello World in C */
#include <stdio.h>
int main() {
printf( "Hello World\n" );
return( 0 );
 }
```

The principal things to note from this example are:
• The '#include' directive which allows this program to use all the standard input and output functions available in C.
• The declaration of the 'main' procedure. Every C program must have exactly one of these as the program is run by calling it.
• The 'printf (. .}' statement calls a function provided in stdio which sends output to the standard output device.
Using the ARM software development tools, the above program should be saved as 'HelloW.c'. Then a new project should be created using the Project Manager and this file added (as the only file in the project). A click on the 'Build' button will cause the program to be compiled and linked, then the 'Go' button will run it on the ARMulator, hopefully giving the expected output in the terminal window.