Example 2.1 Describe the principal features of the ARM architecture.
The main features of the ARM architecture are:
• a large set of registers, all of which can be used for most purposes;
• a load-store architecture;
• 3-address instructions (that is, the two source operand registers and the result reg ister are all independently specified);
• conditional execution of every instruction;
• the inclusion of very powerful load and store multiple register instructions;
• the 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.
If the Thumb instruction set is considered part of the ARM architecture, we could also add:
• a very dense 16-bit compressed representation of the instruction set in the Thumb architecture.

Which features does ARM have in common with many other RISC architectures?

Which features of the ARM architecture are not shared by most other RISCs? Which features of most other RISC architectures are not shared by the ARM?

Print out r1 in hexadecimal.
This is a useful little routine which dumps a register to the display in hexadecimal (base 16) notation; it can be used to help debug a program by writing out register values and checking that algorithms are producing the expected results, though in most cases using a debugger is a better way of seeing what is going on inside a program.

```
AREA Hex_Out,CODE,READONLY
SWI_WriteC EQU &0 output character in r0
SWI_Exit EQU &11 finish program
ENTRY code entry point
LDR r1, VALUE get value to print
BL HexOut call hexadecimal output
SWI SWI_Exit finish
VALUE DCD &12345678 test value
HexOut MOV r2, #8 nibble count = 8
LOOP MOV r0, r1, LSR #28 get top nibble
CMP r0, #9 0-9 or A-F?
ADDGT r0, r0, #"A"-10 ASCII alphabetic
ADDLE r0, r0, #"0" ASCII numeric
SWI SWI_WriteC print character
MOV r1, r1, LSL #4 shift left one nibble
SUBS r2, r2, #1 decrement nibble count
BNE LOOP if more do next nibble
MOV pc, r14 return
END
```

Modify the above program to output r1 in binary format. For the value loaded into r1 in the example program you should get:
00010010001101000101011001111000
Use HEXOUT as the basis of
a program to display the contents of an area of memory.

Example 3.2 **Write a subroutine to output a text string immediately following the call.**

It is often useful to be able to output a text string without having to set up a separate data area for the text (though this is inefficient if the processor has separate data and instruction caches, as does the StrongARM; in this case it is better to set up a separate data area). A call should look like:

```
        BL      TextOut
        =       "Test string",&0a,&0d,0
        ALIGN
        ..                          ; return to here
```

The issue here is that the return from the subroutine must not go directly to the value put in the link register by the call, since this would land the program in the text string. Here is a suitable subroutine and test harness:

```
        AREA    Text_Out,CODE,READONLY
SWI_WriteC      EQU     &0          ; output character in r0
SWI_Exit        EQU     &11         ; finish program
        ENTRY                       ; code entry point
        BL      TextOut             ; print following string
        =       "Test string",&0a,&0d,0
        ALIGN
        SWI     SWI_Exit            ; finish
TextOut LDRB    r0, [r14], #1       ; get next character
        CMP     r0, #0              ; test for end mark
        SWINE   SWI_WriteC          ; if not end, print..
        BNE     TextOut             ;  .. and loop
        ADD     r14, r14, #3        ; pass next word boundary
        BIC     r14, r14, #3        ; round back to boundary
        MOV     pc, r14             ; return
        END
```

This example shows r14 incrementing along the text string and then being adjusted to the next word boundary prior to the return. If the adjustment (add 3, then clear the bottom two bits) looks like slight of hand, check it; there are only four cases.

Exercise 3.2.1 Using code from this and the previous examples, write a program to dump the ARM registers in hexadecimal with formatting such as:

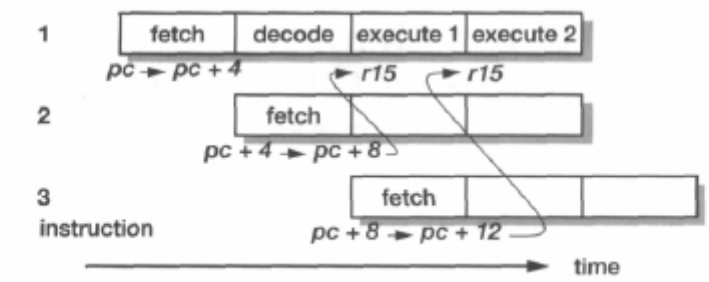
```
r0 = 12345678 r1
= 9ABCDEF0
```

Example 4.1 **Why does r15 give pc + 8 in the first cycle of an instruction and pc + 12 in subsequent cycles on an ARM7?**

This is the ARM pipeline being exposed to the programmer. Referring back to Figure 4.2 on page 77, we can see that the pc value was incremented once when the current instruction (' 1' in the figure below) was fetched and once when its successor ('2') was fetched, giving pc + 8 at the start of the first execute cycle. During the first execute cycle a third instruction ('3') is fetched, giving pc + 12 in all subsequent execute cycles.

While multi-cycle instructions interrupt the pipeline flow they do not affect this aspect of the behaviour. An instruction always fetches the next-instruction-but-one during its first execute cycle, so r15 always progresses from pc + 8 at the start of the first execute cycle to pc + 12 at the start of the second (and subsequent) execute cycle(s).

(Note that other ARM processors do not share this behaviour, so it should never be relied upon when writing ARM programs.)



Exercise 4.1.1 Draw a pipeline flow diagram along the lines of the one above to illustrate the timing of an ARM branch instruction. (The branch target is computed in the first execute cycle of the instruction and issued to memory in the following cycle.)

Exercise 4.1.2 How many execute cycles are there after the branch target calculation and before the instruction at the branch target is ready to execute? What does the processor use these execute cycles for?

Example 4.2 **Complete the ARM2 4-bit carry logic circuit outlined in Figures 4.1 1 and 4. 12 on page 89.**

The 4-bit carry look-ahead scheme uses the individual bit carry generate and propagate signals produced by the logic shown in Figure 4.12. Denoting these by G[3:0] and P[3:0], the carry-out from the top bit of a 4-bit group is given by:

Cout = G[3] + P[3].(G[2] + P[2].(G[1] + P[1].(G[0] + P[0].Cin)))

Therefore the group generate and propagate signals, G4 and P4, as used in Figure 4. 1 1 are given by:

G4 = G[3] + P[3].(G[2] + P[2].(G[1] + P[1].G[0]))

P4 =P[3].P[2].P[1].P[0]

These two signals are independent of the carry-in signal and therefore can be set up ready for its arrival, allowing the carry to propagate across the 4-bit group in just one AND-OR-INVERT gate delay.

Exercise 4.2.1 Write a logic expression for one bit of the ALU output generated by the circuit shown in Figure 4.12 in terms of the inputs and the function select lines, and hence show how all the ALU functions listed in Table 4. 1 on page 90 are generated.

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. By default this is the display terminal.

As with the assembly programming exercises, the major challenge is to establish the flow through the tools from editing the text to compiling, linking and running the program. Once this program is working, generating more complex programs is fairly straightforward (at least until the complexity reaches the point where the design of the program becomes a challenge in itself).

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.

Example 6.2 **Write the number 2001 in 32-bit binary, binary-coded decimal, ASCII and single-precision floating-point notation.**

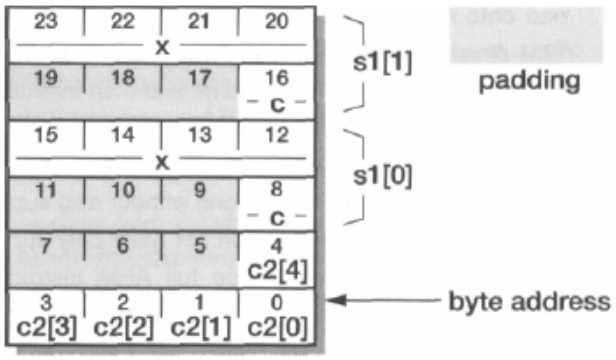
```
Binary: 2001    = 1024 + 512 + 256 + 128 + 64 + 16 + 1
                = 00000000000000000000011111010001₂
BCD:    2001    = 0010 0000 0000 0001
ASCII:  2001    = 00110010 00110000 00110000 00110001
F-P:    2001    = 1.1111010001 x 2^1010
                = 01001001 11110100 01000000 00000000
```

Exercise 6.2.1 Write a C program to convert a date presented in Roman numerals into decimal form.

Example 6.3 **Show how the following data is organized in memory:**

```
struct SI {char c; int x;}; struct
S2 {
  char c2[5];
  SI  si [2]; }
example;
```

The first structure statement only declares a type, so no memory is allocated. The second establishes a structure called 'example' comprising an array of five characters followed by an array of two structures of type SI. The structures must start on a word boundary, so the character array will be padded out to fill two words and each structure will also occupy two words. The memory organization is therefore as shown below:



Exercise 6.3.1 Show how the same structure would be organized in memory if it were packed.

Example 7.1 **Rewrite the 'Hello World' program in Section 3.4 on page 69 to use Thumb instructions. How do the ARM and Thumb code sizes compare?**

Here is the original ARM program:

```
        AREA    HelloW,CODE,READONLY
SWI_WriteC      EQU     &0          ; output character in r0
SWI_Exit        EQU     &11         ; finish program
        ENTRY                       ; code entry point
START   ADR     r1, TEXT            ; r1 -> "Hello World"
LOOP    LDRB    r0, [r1], #1        ; get the next byte
        CMP     r0, #0              ; check for text end
        SWINE   SWI_WriteC          ; if not end print ..
        BNE     LOOP                ;  .. and loop back
        SWI     SWI_Exit            ; end of execution
TEXT    =       "Hello World",&0a,&0d,0
        END                         ; end of program source
```

Most of these instructions have direct Thumb equivalents; however, some do not. The load byte instruction does not support auto-indexing and the supervisor call cannot be conditionally executed. Hence the Thumb code needs to be slightly modified:

```
        AREA    HelloW_Thumb,CODE,READONLY
SWI_WriteC      EQU    &0       ; output character in r0
SWI_Exit        EQU    &11       ; finish program
        ENTRY                      ; code entry point ;
        CODES2                     enter in ARM state
```

```
        ADR     r0, START+1          ;get Thumb entry address
        BX      r0                   ;enter Thumb area
        CODE16                       ;Thumb code follows..
START   ADR     r1, TEXT             ;r1 -> "Hello World"
LOOP    LDRB    r0, [r1]             ;get the next byte
        ADD     r1, r1, #1           ;increment pointer  **T
        CMP     r0, #0               ;check for text end
        BEQ     DONE                 ;finished?          **T
        SWI     SWI WriteC           ;if not end print . .
        B       LOOP                  ;.. and loop back
DONE    SWI     SWI_Exit             ;end of execution
        ALIGN                        ;to ensure ADR works
TEXT    DATA
                "Hello World",&0a,&0d,&00
        END
```

The two additional instructions required to compensate for the features absent from the Thumb instruction set are marked with '**T' in the above listing. The ARM code size is six instructions plus 14 bytes of data, 38 bytes in all. The Thumb code size is eight instructions plus 14 bytes of data (ignoring the preamble required to switch the processor to executing Thumb instructions), making 30 bytes in all.

This example illustrates a number of important points to bear in mind when writing Thumb code:

- The assembler needs to know when to produce ARM code and when to produce Thumb code. The 'CODES 2' and 'CODE16' directives provide this information. (These are instructions to the assembler and do not themselves cause any code to be generated.)
- Since the processor is executing ARM instructions when it calls the code, explicit provision must be made to instruct it to execute the Thumb instructions. The 'BX rO' instruction achieves this, provided that r0 has been initialized appropri ately. Note particularly that the bottom bit of r0 is set to cause the processor to execute Thumb instructions at the branch target.
- In Thumb code 'ADR' can only generate word-aligned addresses. As Thumb instructions are half-words, there is no guarantee that a location following an arbitrary number of Thumb instructions will be word-aligned. Therefore the example program has an explicit 'ALIGN' before the text string.

In order to assemble and run this program on the ARM software development toolkit, an assembler that can generate Thumb code must be invoked and the ARMula-tor must emulate a 'Thumb-aware' processor core. The default setting of the Project Manager targets an ARM6 core and generates only 32-bit ARM code. This may be changed by choosing 'Project' from the 'Options' menu within the Project Manager