# Experiment 3:
# Data Types, Data Structures and Functions

This experiment further consolidates the programmer's view of computer architecture. It does this by showing you how C data types, data structures and functions are represented in ARM assembly language.

## Aims

This experiment aims to:

- show the support that the ARM instruction set architecture has for dealing with data types,
- teach you how to deal with pointers and arrays in the process of hand compiling a C program into assembly language,
- illustrate how functions can be implemented using stack frames, and
- show more examples of writing and debugging assembly language programs for the ARM microprocessor.

## Preparation

It is important that you prepare for each laboratory experiment, so that you can use your time (and your partner's time) most effectively. For this particular experiment, you should do the following *before* coming in to the Laboratory:

- read through this experiment in detail, trying to understand what you will be doing,
- skim-read the sections on Programming Style in Experiments 1 and 2,
- quickly read through the relevant material from your lecture notes for this course, and
- quickly skim through *An Introduction to the GNU Assembler* and *An Introduction to the GNU Debugger*. You can find these documents on your CD-ROM.

If you are keen (and you should be!), you could also:

- browse through the Companion CD-ROM, if you haven't done so already,
- type up or modify the necessary files in this experiment, to save time in class, and
- run through the experiment at home using the simulator.

## Getting Started

Once you arrive at the Laboratory, find a spare workbench and log into the Host PC. Next, create a new directory for this experiment. Then, copy all of the files in the directory *~elec2041/unsw/elec2041/labs-src/exp3* on the Laboratory computers into this new directory. You can do all of this by opening a Unix command-line shell window and entering:

```
mkdir ~/exp3
cd ~/exp3
cp ~elec2041/cdrom/unsw/elec2041/labs-src/exp3/* .
```

Be careful to note the "**.**" at the end of the **cp** command!

If you are doing this experiment at home, you will not have a *~elec2041* directory, of course. You should use the *unsw/elec2041/labs-src/exp3* directory on your CD-ROM instead.

# Representing Data

Computers represent every piece of data in one form or another as a sequence of *binary digits (bits)*. Most programming languages, including C, operate on three types of data: *integers*, *characters* and *floating point numbers*.

## Integer Numbers

There are two main types of binary representation for integers:

- positive numbers only (known as *unsigned integers*), and
- positive and negative numbers (known as *signed integers*)

Unsigned integers only represent positive values and the number zero. Given an $n$-bit integer, the range of numbers that can be represented is from 0 to $2^n-1$.

Take 4-bit numbers as an example (ie, when $n = 4$). The range of unsigned integers that can be represented is then from 0 to $2^4-1 = 15$:

| Bin | Dec | Hex |
|---|---|---|
| 0000 | 0 | 0 |
| 0001 | 1 | 1 |
| 0010 | 2 | 2 |
| 0011 | 3 | 3 |
| 0100 | 4 | 4 |
| 0101 | 5 | 5 |
| 0110 | 6 | 6 |
| 0111 | 7 | 7 |

| Bin | Dec | Hex |
|---|---|---|
| 1000 | 8 | 8 |
| 1001 | 9 | 9 |
| 1010 | 10 | A |
| 1011 | 11 | B |
| 1100 | 12 | C |
| 1101 | 13 | D |
| 1110 | 14 | E |
| 1111 | 15 | F |

(In this table, "Bin" stands for Binary, "Dec" stands for Decimal and "Hex" stands for Hexadecimal).

Signed integers can represent positive values, negative values and the number zero. Almost all modern computers use *two's-complement arithmetic* to represent signed integers. In this format, given an $n$-bit integer, the range of numbers that can be represented is $-2^{n-1}$ to $2^{n-1} - 1$. In two's complement arithmetic, the Most Significant Bit (MSB) of the integer is used to represent the *sign:* a 0 in the MSB represents a positive number, a 1 represents a negative one.

If $n = 4$, for example, the range of numbers that can be represented is from $-2^{4-1}$ to $2^{4-1} - 1$, ie, from –8 to +7:

| Bin | Dec | Hex |
|---|---|---|
| 0000 | 0 | 0 |
| 0001 | 1 | 1 |
| 0010 | 2 | 2 |
| 0011 | 3 | 3 |
| 0100 | 4 | 4 |
| 0101 | 5 | 5 |
| 0110 | 6 | 6 |
| 0111 | 7 | 7 |

| Bin | Dec | Hex |
|---|---|---|
| 1000 | –8 | 8 |
| 1001 | –7 | 9 |
| 1010 | –6 | A |
| 1011 | –5 | B |
| 1100 | –4 | C |
| 1101 | –3 | D |
| 1110 | –2 | E |
| 1111 | –1 | F |

Almost all modern computers use either 32 or 64 for $n$. In other words, virtually all modern computers use 32-bit or 64-bit integers as their default integer size.

As you can see from the two tables above, the same binary numbers are used to represent both unsigned integers *and* signed integers. The first part of both tables is the same (that is, the representation of the numbers 0 to 7 is the same whether they are unsigned or signed). The representation for the other numbers differ, however: the binary pattern for the unsigned numbers 8 to 15 is also used to represent the signed numbers –8 to –1.

By the way, there is an easy way for you to work out the two's complement representation of any positive number: all you have to do is *complement* every bit (ie, turn every 0 into a 1 and every 1 into a 0), then add one.

For example, if $n = 4$, to work out –5, take the binary representation for +5 (0101), complement every bit (to get 1010) and add one:

| | | |
|---|---|---|
| Binary representation of positive value: | 0101 | (+5) |
| Complement every bit: | 1010 | |
| Add one: | + 1 | |
| Result in two's complement: | **1011** | (–5) |

**Question:** For $n = 6$, what value does the two's complement number 110011 represent, as a decimal number?

For 32-bit numbers, where $n = 32$, the range of numbers that can be represented as signed and unsigned integers is:

| | | | | | |
|---|---|---|---|---|---|
| For signed integers: | –2,147,483,648 | (0x80000000) | to | 2,147,483,647 | (0x7FFFFFFF) |
| For unsigned integers: | 0 | (0x00000000) | to | 4,294,967,295 | (0xFFFFFFFF) |

## Extending Unsigned and Signed Numbers

There are times that you will need to change an integer with a certain number of bits into one having the same value (and the same representation), but with a larger number of bits.

For **unsigned integers**, this is done by *zero extension:* essentially, "just sticking on zero bits to the left". Thus, to extend a 4-bit unsigned number ($n = 4$) to become an 8-bit number, just place four extra zeros to the left (ie, in front). In other words, copy the original integer *xxxx* into the lower bits, and put zero bits in the upper bits:

*xxxx* → 0000*xxxx*
eg, 1010 → 00001010

For **signed integers**, the process is called *sign extension:* the sign in the Most Significant Bit (MSB) of the original number is copied into the extended bits of the new number. Thus, to extend a 4-bit signed number ($n = 4$) to become an 8-bit number:

*sxxx* → *sssssxxx*
eg, 1010 → 11111010

## Numeric Overflow

In binary number representation, sometimes a value cannot be represented in the limited number of bits available. For example, the number 16 (10000 in binary) cannot be represented as a 4-bit unsigned number—it requires at least five bits. Similarly, the number 8 (01000) cannot be represented as a 4-bit signed number—it also requires at least five bits. (Can you see why?)

When a value cannot be represented in the number of bits available, an *overflow* is said to have occurred. Overflows can occur when doing arithmetic operations. For example, for 4-bit unsigned numbers:

| | | |
|---|---|---|
| 0011 | (3) | + |
| 1110 | (14) | |
| 1 0001 | (17) | |

This is an overflow since *five* bits are needed to represent the result 17 as an unsigned number: it cannot fit into four bits!

## Characters

A common non-integer data type that needs to be represented in a computer is a *character*. These are encoded into binary using some form of an *encoding:* a table that maps individual characters to binary sequences. Many input/output devices, including keyboards and monitors, work with 8-bit quantities. For this reason, the most common character encodings are 8 bits in size.

The standard character encoding used on almost every computer is the so-called *American Standard Code for Information Interchange* (ASCII). This character encoding defines what character each binary sequence represents. You can find this encoding on your CD-ROM in the *reference/misc* directory. If you are interested, you might also want to check the Unicode Consortium's Web page at *http://www.unicode.org/* for a vastly extended character encoding standard.

Some examples of the ASCII character encoding:

| Binary | Hex | Dec | Character |
|---|---|---|---|
| 01000001 | 0x41 | 65 | A |
| 01000010 | 0x42 | 66 | B |
| 01100001 | 0x61 | 97 | a |
| 00110000 | 0x30 | 48 | 0 (zero) |
| 00111000 | 0x38 | 56 | 8 |
| 00111001 | 0x39 | 57 | 9 |
| 00100101 | 0x25 | 37 | % |

A different bit pattern is used to represent each character that needs to be encoded. Some points to note:

- The character 'a' (0x61) is different from 'A' (0x41). In other words, upper and lower case letters are represented differently,
- The character '8' (0x38) is different from the integer 8 (0x00000008 as a 32-bit number),
- If you compare bit patterns (by treating them as integers), you will find that 'A' < 'B' (0x41 < 0x42). In fact, the whole English alphabet is in this "natural" sequence. This is good as it helps with sorting things into alphabetical order… at least, as long as you are using English and not some other language, like French!

> **Caution:** In integer arithmetic, 3 + 4 = 7. However, in ASCII, the characters '3' + '4' ≠ '7'. That is because 51 (the decimal representation of '3') plus 52 (the decimal representation of '4') is 103 (the character 'g')!
>
> You *can* add '3' and '4' together to get '7'. First, take out the "bias" of 0x30 (the character '0', zero) out of each character. This gives you the numbers 3 and 4 (since 0x34 – 0x30 = 3, and so on). Now, add them together to get 7. Finally, add back the bias of 0x30 to get the character "7" (0x37). **This only works with single digits!**

## Programming Language Support

The C programming language supports the three data representations already mentioned. In particular, to declare a signed integer variable, use the **`int`** keyword. To declare an unsigned integer, use **`unsigned int`** (or just **`unsigned`** by itself). And to declare a character variable, use **`char`**.

Consider the two C programs in Figure 1 and Figure 2:

```
int main (void)
{
    int a = 0xE0001234;
    int b = 0x00004567;

    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

**Figure 1: Program *cmp-c-s.c***

```
int main (void)
{
    unsigned a = 0xE0001234;
    unsigned b = 0x00004567;

    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

**Figure 2: Program *cmp-c-u.c***

**Question:** Which value, a or b, do each of these programs return? Why?

The values returned by the programs *cmp-c-s.c* and *cmp-c-u.c* depend on the interpretation given to the variables a (which has the value 0xE0001234) and b (0x00004567). The first program, *cmp-c-s.c*, treats a and b as the *signed* integers –536,866,252 and 17,767 respectively. The second program, on the other hand, treats a and b as the *unsigned* integers 3,758,101,044 and 17,767 respectively.

The programs in Figure 3 and Figure 4 are the hand-translated ARM assembly language versions of these two C programs. Study these two programs carefully:

```
        .text                   ; This program is the equivalent of cmp-c-s.c

_start: .global _start          ; "_start" is required by the linker
        .global main            ; "main" is our main program

        b       main            ; Start running the main program

main:   ldr     r3, =a          ; Load the address of a into R3
        ldr     r0, [r3]        ; R0 = 0xE0001234
        ldr     r1, [r3, #4]    ; R1 = 0x00004567 (NB: R3 + 4 = address of b)
        cmp     r0, r1          ; Compare a and b (a - b)
        bgt     exit            ; Return a (a > b)  [Signed branch]
        mov     r0, r1          ; Return b (a =< b)

exit:   mov     pc, lr          ; Stop the program (and return to OS)

a:      .word   0xE0001234      ; Variable "a"
b:      .word   0x00004567      ; Variable "b"

        .end
```

**Figure 3: Program *cmp-s.s***

```
        .text                   ; This program is the equivalent of cmp-c-u.c

_start: .global _start          ; "_start" is required by the linker
        .global main            ; "main" is our main program

        b       main            ; Start running the main program

main:   ldr     r3, =a          ; Load the address of a into R3
        ldr     r0, [r3]        ; R0 = 0xE0001234
        ldr     r1, [r3, #4]    ; R1 = 0x00004567 (NB: R3 + 4 = address of b)
        cmp     r0, r1          ; Compare a and b (a - b)
        bhi     exit            ; Return a (a > b)  [Unsigned branch]
        mov     r0, r1          ; Return b (a =< b)

exit:   mov     pc, lr          ; Stop the program (and return to OS)

a:      .word   0xE0001234      ; Variable "a"
b:      .word   0x00004567      ; Variable "b"

        .end
```

**Figure 4: Program *cmp-u.s***

The assembly language instructions `bgt` (branch if signed greater) and `bhi` (branch if unsigned higher) in the programs in Figure 3 and Figure 4 respectively provide the two different interpretations of the "`cmp r0, r1`" instruction.

> **Note:** To reduce code size in your programs, the ARM instruction set architecture allows you to use a wide set of *conditional instructions:* instructions that are executed only if certain conditions are met. Take, for example, the following two instructions in Figure 3:
>
> ```
> bgt     exit            ; return a (a > b)
> mov     r0, r1          ; return b (a =< b)
> ```
>
> These two instructions can be combined into the following single instruction:
>
> ```
> movle   r0, r1          ; return b (a =< b)
> ```
>
> The conditional-move instruction `movle` (move if signed less than or equal) is conditionally executed depending on the result of the previous `cmp` instruction.

**Question:** What is the equivalent conditional-move instruction that combines the two instructions "`bhi exit`" and "`mov r0, r1`" in the program *cmp-u.s* in Figure 4?

## Simple Character Manipulation

The programs in Figure 5 and Figure 6 are the C and ARM assembly language versions respectively of a program that maps characters in the range 'a'–'z' to characters in the range 'A'–'Z':

```
char main (void)
{
    char lower = 'a';       /* 'a' is used as an example of a character */
    char upper;

    upper = lower - 0x20;   /* Lower-case and upper-case ASCII characters are
                               seperated by 0x20, ie, 32 */

    return upper;           /* main() is called by the Operating System */
}
```

**Figure 5: Program *chsub-c.c***

```
        .text                   ; Executable code follows

_start: .global _start          ; "_start" is required by the linker
        .global main            ; "main" is our main program

        b       main            ; Start running the main program

main:   ldr     r3, =lower      ; Load the address of "lower" into R3
        ldrb    r0, [r3]        ; R0 = 0x00000061 (the character 'a')

        sub     r0, r0, #0x20   ; R0 = 'a' - 0x20 = 'A'

        ldr     r3, =upper      ; Load the address of "upper" into R3
        strb    r0, [r3]        ; Store the (byte) value in R0 to "upper".

exit:   mov     pc, lr          ; Stop the program (and return to the OS)

lower:  .byte   'a'             ; Variable "lower"; 'a' is used as an example
upper:  .byte   0               ; Variable "upper"

        .end
```

**Figure 6: Program *chsub-s.s***

As you can see from Figure 6, ARM assembly language supports the character data type by providing the instructions `ldrb` and `strb`, for "load byte from memory" and "store byte into memory", respectively.

**Note:** The instruction "`ldrb r0, [r3]`" loads the byte from memory into the *lowest* byte of register R0. The upper three bytes are set to zero. In other words, this instruction *zero-extends* the byte read into register R0. If you wish to *sign-extend* the byte as it is read from memory, use the `ldrsb` instruction ("load signed byte from memory").

In addition, the ARM instruction set architecture allows *half-word* (16-bit) accesses to memory by providing the `ldrh`, `ldrsh` and `strh` instructions ("load half-word", "load signed half-word" and "store half-word", respectively).

## Task 1: Character Manipulation

The program *chsub-s.s* in Figure 6 suffers from a major problem: it subtracts 0x20 from *any* byte stored at label `lower`, whether that byte is in the range 'a'–'z' or not. Write a new version of the program, and call it *toupper.s*, that converts characters in the range 'a'–'z' to their upper-case equivalent 'A'–'Z', but leaves characters outside of this range (of 'a'–'z') unchanged. Use the GNU Tools to show your working program to the Laboratory assessor.

**Hint:** The following command lines assemble and link the source code file *toupper.s*:

```
arm-elf-as -marm7tdmi --gdwarf2 -o toupper.o toupper.s
arm-elf-ld -o toupper.elf toupper.o
```

If you prefer, you can use the **`make`** command instead; the make-file is called *toupper.make*, and can be used by typing:

```
make -f toupper.make
```

You will use similar command lines for all of the other tasks in this experiment as well. Appropriate make-files have been provided, if you wish to use them. You should also remember, from previous experiments, that the GNU Debugger is started with the **`arm-elf-insight`** command.

Checkpoint 1: ..... Signature: .....

## Pointers and Arrays

An *array* refers to a collection of objects of the same type. Each element in an array has the same size (for example, one byte for characters, one word for integers), and elements are stored contiguously (ie, elements follow one another). On virtually all computers, the first element is stored at the lowest memory address of the entire array.

In the C programming language, arrays are actually used only as a short-hand notation for *pointer* operations. For example, the array declaration

```
int arr[10];
```

creates an array `arr` of 10 integers, `arr[0]` to `arr[9]`. The name `arr` is just a pointer to the first element, `arr[0]`. In other words, `*arr` is equivalent to `arr[0]`. Furthermore, element `i` of the array can be accessed by using either `arr[i]` or `*(arr + i)`.

In assembly language programming, an array can be implemented by allocating the required space in memory and by using the address of any element to access that element.

In modern computer systems, memory is organised and addressed in units of bytes. In other words, each byte has its own unique address. This ability to access individual bytes is called *byte addressing*.

The programs in Figures 7 and 8 are the C and assembly versions of the following vector calculation:

$$C[i] = \frac{A[i]}{2} + 2B[i]$$

```
int main (void)
{
    int a[4] = {-1, 2,  3, 4};
    int b[4] = { 5, 6, -7, 8};
    int c[4];
    int i;

    for (i = 0; i <= 3; i++) {
        *(c + i) = (*(a + i) >> 1) + (*(b + i) << 1);

            /* Note that this line is the same as:
                   c[i] = (a[i] >> 1) + (b[i] << 1)
               which is essentially the same as:
                   c[i] = (a[i] / 2)  + (b[i] * 2)  */
    }

    return 0;
}
```

**Figure 7: Program *matrix-c.c***

```
        .text                   ; Executable code follows

_start: .global _start          ; "_start" is required by the linker
        .global main            ; "main" is our main program

        b       main            ; Start running the main program

main:   mov     r5, #0          ; Assign i to R5; initialise for start of loop
        ldr     r2, =a          ; R2 = address of "a", ie, R2 points to a[0]
        ldr     r3, =b          ; R3 = address of array "b"
        ldr     r0, =c          ; R0 = address of array "c"

mainlp: ldr     r6, [r2, r5]    ; Load R6 with contents of a[i] (a+i = R2+R5)
        ldr     r1, [r3, r5]    ; Load R1 with contents of b[i]
        mov     r6, r6, asr #1  ; R6 = a[i] >> 1 (asr is Arithmetic Shift Right)
        mov     r1, r1, asl #1  ; R1 = b[i] << 1 (asl is Arithmetic Shift Left)
        add     r1, r6, r1      ; R1 = a[i] >> 1 (in R6) + b[i] << 1 (in R1)
        str     r1, [r0, r5]    ; Store the result into c[i] (c+i = R0+R5)
        cmp     r5, #(3 * 4)    ; Is i > 3 * 4-bytes (int takes four bytes)
        add     r5, r5, #4      ; Increment R5 (i++); does not affect cmp result
        bls     mainlp          ; If i <= 3 * 4, repeat the loop
                                ; NB: bls is "branch if less", but i has already
                                ; been incremented, so real meaning is as shown
                                ; in the comment above (ie, is "<=", not "<").

exit:   mov     pc, lr          ; Stop the program (and return to OS)

a:      .word   -1, 2,  3, 4    ; Array "a"
b:      .word    5, 6, -7, 8    ; Array "b"
c:      .word    0, 0,  0, 0    ; Array "c"

        .end
```

**Figure 8: Program *matrix-v1.s***

If you examine these programs carefully, you will find that this vector calculation uses left shift ("`<<`" or `asr`) and right shift ("`>>`" or `asl`) instead of division and multiplication: this is mainly for efficiency. Besides, the ARM processor does *not* have a native instruction for division, as you will find out later in this experiment.

Finally, note that each element of the arrays `a`, `b` and `c` is of type `int` (a signed integer). Each element, therefore, is four bytes (one word) in size. For this reason, Figure 8 increments the array index in register R5 by *four*, not one (byte).

The program in Figure 8 can be further optimised by taking advantage of the *post-indexing* variant of the `ldr` instruction, a *shift/rotate* variant of the `add` instruction and by taking advantage of the structure of the program. This optimised version is provided in Figure 9:

```
        .text                       ; Executable code follows

_start: .global _start              ; "_start" is required by the linker
        .global main                ; "main" is our main program

        b       main                ; Start running the main program

main:   ldr     r2, =a              ; R2 = address of "a", used as "counter" a+i
        add     r3, r2, #(b-a)      ; R3 = address of "b" (NB: b-a = 16 bytes)

mainlp: ldr     r6, [r2], #4        ; Load R6 with *(a+i), then make R2 = R2 + 4
        ldr     r1, [r2, #(b-a-4)]  ; R1 = *(b+i) (NB: b-a-4 = 12 bytes)
        mov     r1, r1, asl #1      ; R1 = *(b+i) << 1
        add     r1, r1, r6, asr #1  ; R1 = R1 + (R6 >> 1)
        str     r1, [r2, #(c-a-4)]  ; Store result into *(c+i). (NB: c-a-4 = 28)
        cmp     r2, r3              ; Has the end of a[] been reached, ie, is
                                    ;   it now b[0]?  Assumes "b" follows "a".
        bls     mainlp              ; No, repeat the loop

exit:   mov     pc, lr              ; Stop the program (and return to OS)

a:      .word   -1, 2,  3, 4        ; Array "a" (each array is 16 bytes in size)
b:      .word    5, 6, -7, 8        ; Array "b"
c:      .word    0, 0,  0, 0        ; Array "c"

        .end
```

**Figure 9: Program *matrix-v2.s***

**Task:** Calculate the number of instructions executed by the programs in Figure 8 and Figure 9 if each program runs through the loop $i$ times (in this particular case, $i = 4$, but you should generalise your calculation).

# Task 2: String to Integer Conversion

The program *atoi-v1.c* in Figure 10 translates a *string* of digit characters (that is, a sequence of ASCII characters '0'–'9') into the integer number the string represents. Write an ARM assembly language version of this program and call it *atoi-v1.s*. Include each of the C statements (and each part of the `for` statement) in Figure 10 as a comment in your program. Return the result `n` in register R0. Use the GNU Tools to show your working program to the Laboratory assessor.

**Hint:** Use something like "`s: .asciz "12345"`" to store the string in your program. For more information on the `.asciz` assembler directive, please see *An Introduction to the GNU Assembler*. You can find this document in an Appendix or on your CD-ROM.

**Challenge:** Can you figure out how to do the "`a = 10 * a + b`" operation in assembly language *without* using the `mul` or `mla` instructions? Hint: remember `lsl` from Experiment 2.

```
int main (void)
{
    char s[] = "12345";
    int i, n;

    n = 0;

    for (i = 0; (s[i] >= '0') && (s[i] <= '9'); i++) {
        n = 10 * n + (s[i] - '0');
    }

    return n;
}
```

**Figure 10: Program *atoi-v1.c***

Check the memory location of the string `s`. You will see something like the following in the Memory window of the GNU Debugger:

```
0x81F0:  0x34333231 0x00000035 0x00000000 0x00000000
```

where 0x81F0 is the address of the string (the exact address depends on your program, of course), and 0x34333231 and 0x00000035 correspond to the string itself. Each byte at location 0x81F0 (in this example) and the lower significant byte of the next location corresponds to one ASCII character in the string "12345". The byte 0x34 is '4', 0x33 is '3' and so on.

**Question:** As you can see, the string "12345" seems to be stored in *reverse* as "4321" "5". How do you explain this?

Checkpoint 2: ..... Signature: .....

## Functions and Procedures

Functions and procedures are used extensively in most programming languages. They allow code to be *modularised*, that is, split into separate sections, with each section carrying out one specific task. This simplifies writing programs and allows you to reuse code. It also facilitates code maintenance (when code must be modified in the future) and allows different programmers to work on different parts of the same program.

Examine the program *larger-c.c* in Figure 11. The function `larger` is called by the function `main`; `main`, in turn, is called by the operating system. The function `larger` calculates the larger of the two integer parameters passed to it, and returns that value to `main`. The function `main`, in turn, returns that value to the operating system.

```
int larger (int first, int second)
{
    if (first > second) {
        return first;
    } else {
        return second;
    }
}

int main (void)
{
    int a = 0xE0001234;
    int b = 0x00004567;
    int c;

    c = larger(a, b);

    return c;
}
```

**Figure 11: Program *larger-c.c***

If you examine the assembly language code for this program, you will find that calling the function `larger` involves the following steps:

1. Save the return address,
2. Call the function with the parameters passed appropriately,
3. Execute the actual function, and
4. Return to the caller with the appropriate return value.

The *return address* is the address of the instruction *following* the function call. In Figure 11, the return address is the address of the statement "`return c`". *Calling the function* means

jumping (branching) to the first instruction in the function, in this case, to the first instruction in `larger`. The parameters (also called *arguments*) are passed to the function by placing them in registers. *Returning to the caller* means jumping to the return address that was previously saved. The return value is passed back to the caller by placing it in a register.

Examine the hand-translated assembly language version of *larger-c.c*, provided in Figure 12. (If possible, you should consult the version on your CD-ROM instead, as many comments have been removed to save paper). Notice that the variables `a` and `b` are passed as parameters to the function `larger` in registers R0 and R1. The `bl` instruction ("branch and link") automatically saves the return address by placing it into register R14 (also called LR, the link register), then jumps to the specified label. The return value is passed to the caller in register R0; the actual "return to caller" is done by moving the value in register R14 (LR) to the program counter register PC.

```
        .text                   ; Executable code follows

_start: .global _start          ; "_start" is required by the linker
        .global main            ; "main" is our main program

        b       main            ; Start running the main program

; ------------------------------------------------------------------------
; Function: int main (void)

; This function loads the variables "a" and "b", compares them and stores the
; larger of the two in the variable "c".  The registers R0, R1 and R3 are used.

main:   ldr     r3, =a          ; Load R3 with the address of "a"
        ldr     r0, [r3]        ; R0 = 0xE0001234 (the value of "a")
        ldr     r1, [r3, #4]    ; R1 = 0x00004567 (NB: Address of "b" is R3+4)

        bl      larger          ; Call the function "larger"; store the return
                                ; address (address of next instruction) in R14
                                ; (also called LR).  Result returned in R0.

        ldr     r3, =c          ; Load R3 with the address of "c"
        str     r0, [r3]        ; Store result in "c" (0x00004567)

exit:   mov     pc, lr          ; Stop the program (Caution: bugs are present!)

; ------------------------------------------------------------------------
; Function: int larger (int first, int second)

; This function calculates the larger of the two unsigned integers in registers
; R0 and R1.  It returns the larger of the two in register R0.

larger: cmp     r0, r1          ; Compare "first" and "second" (cmp performs
                                ; first - second, but does not store the result
                                ; of first - second)
        movle   r0, r1          ; Place R1 in R0 if and only if
                                ; R0 ("first") <= R1 ("second")
        mov     pc, lr          ; Return to function main(): restore saved copy
                                ; of LR into PC.

a:      .word   0xE0001234      ; Variable "a"
b:      .word   0x00004567      ; Variable "b"
c:      .word   0               ; Variable "c"

        .end
```

**Figure 12: Program *larger-v1.s***

**Important:** A function is *not* the same as a label! In the case of assembly language programs in particular, a *function* is a higher-level construct that essentially only exists in the programmer's mind, while a *label* is just a point in the program, that is, an address. Thus, a function *may* have more than one label in it, such as shown in the function `main` in Figure 12. This is why you must always provide well-written comments to show the limits of a function, its purpose, parameters and so on.

## Task 3: Character Manipulation as a Function

Modify the program you wrote in Task 1, so that all of the "real work" is done in a function called `make_upper`, in a new program *toupper-f.s*. Make sure this function takes its input and returns its result in the register R0. Use the GNU Tools to show your working program to the Laboratory assessor.

**Hint:** You should definitely look at the file *larger-v1.s* as it appears on your CD-ROM for an example to follow.

| Checkpoint 3: ..... Signature: ..... |
|---|

## Functions and the Stack

You should have noticed by now that the program *larger-v1.s* in Figure 12 has a major problem ("bug"). When the operating system calls `main`, the return address (to the operating system) is saved in register LR (also known as register R14). However, the call to `larger` *overwrites* register LR with a new value—the address of "`ldr r3, =c`". This means that the function `main` has no way of returning to the operating system—the correct return address has been lost! This is an error that cannot be recovered from!

**Question:** What *does* happen if the instruction at the label `exit` is executed?

You will always come across this particular problem when one function calls another. The solution is to save every return address as it is generated. Unfortunately, there are times when it is impossible to predict how many times a function will be called, and in what order. For example, *recursive functions* call themselves: a recursive function `rf` might call itself, `rf`, which calls `rf`, which also calls `rf`, and so on… What is needed is a way of saving the return addresses *dynamically*, while the program is running.

There is one additional complication: return addresses need to be used (to return to the callers) in the *reverse* order of being generated (saved at the time of a function call). If function `a` calls function `b`, which in turn calls function `c`, the last return address generated (from `c` back to `b`) needs to be used *before* the return address from `b` to `a`.

The best way to save data (in this case, return addresses) that are needed in a LIFO (last-in, first-out) manner is with a *stack*. This data structure grows and shrinks dynamically as entries are *pushed* onto it or *popped* from it. A *stack pointer* is also needed, to keep track of the location of the item on the top of the stack.

The ARM instruction architecture is particularly flexible when it comes to stacks. A stack can be defined to grow towards either larger or smaller addresses (*ascending* or *descending* stacks, respectively). The stack pointer can be defined to point either to an empty location (a pointer to the next available entry) or to a full location (a pointer to the last item placed onto the stack). If you think about it, you will see that this gives four possible combinations: a *full ascending* stack, an *empty ascending* stack, a *full descending* stack and an *empty descending* stack.

## ARM Thumb Procedure Call Standard

If every function defined its own method of accessing the stack (and defined its own type of stack, too), and if every function chose its own rules for passing parameters and returning results, the net effect would be complete chaos!

It was for this very reason that the *ARM Thumb Procedure Call Standard* was defined. This standard (called the ATPCS for short) is used by compiler writers and other programmers to stipulate conventions on how the stack should be defined and how registers are to be used

for passing parameters and returning results. You can find a copy of this Standard on your CD-ROM in the *reference* directory.

As a rule, you should always use the *ARM Thumb Procedure Call Standard* when writing your programs. Doing so will allow functions in your code to be used (successfully!) by functions written by others or by a compiler. The ATPCS requires that:

- The first four parameters must be passed in registers R0–R3, in that order. These registers are also called A1–A4. Any remaining parameters must be passed on the stack and removed by the original caller.
- The registers R0–R3 (A1–A4) are *not* saved across function calls. In other words, if a particular function (called the *caller*) needs to use the values in R0–R3, but needs to make a call to another function in the mean-time, it is the *caller's* responsibility to save R0–R3.
- The registers R4–R11 and R13 must have the same values on returning from a function as they had on that function's entry. In other words, if a particular function (sometimes called the "callee") needs to modify any register in R4–R11 or R13, it is that function's responsibility to save the original value and to restore that value just before it returns. The stack is the best place to store these registers.
- The return value must be returned in register R0 (also called A1).
- The register R13 is reserved as the *system stack pointer*. This register is also called SP.
- The register R12 is reserved as the *interlink scratch register*. This register is also called IP, and can be used in the parts of code that deal with function entry and exit. Its value does not need to be preserved.
- The register R11 is reserved as the *frame pointer*. It is also called FP.
- The stack grows towards smaller addresses, and the stack pointer SP (ie, R13) points to a full location. In other words, the stack is *full descending*.

To implement these requirements, each function call and return must be accompanied by code that places or removes certain items to and from the stack. These items include the return address (in register LR, ie, R14), any parameters and various other registers.

Each function has different requirements for the number of parameters passed to it, and which registers it needs to use and therefore save. For this reason, each function must compose its own stack frame, also known as an *activation record*. A *stack frame* is space on the stack that is allocated every time a particular function is called. The frame is removed from the stack every time that function returns. Stack frames are created and destroyed on the stack dynamically, while the program is running.

The program *larger-v1.s* in Figure 12 has been rewritten to follow the ATPCS, as shown in Figure 13. Examine this program carefully, noting in particular function entry and exit code:

```
; The on-line version of this program has many more comments!

        .text                   ; Executable code follows

_start: .global _start          ; "_start" is required by the linker
        .global main            ; "main" is our main program

        b       main            ; Start running the main program

; ------------------------------------------------------------------------
; Function:    main             - Main program entry point
; Parameters:  (none)           - No parameters are passed
; Returns:     int (in A1)      - Return value: the value of "c"

; This function compares two integers stored in the variables "a" and "b".  It
; uses the function "larger" to make the comparison.  The larger of the two
; variables is stored in "c", which is returned to the operating system in
; register A1 (R0).
```

*(Continued on the next page…)*

*(Continued from the previous page...)*

```
main:   mov     ip, sp          ; Save the caller's (OS's) stack pointer into IP
        str     pc, [sp, #-4]! ; SP = SP - 4, then save current value of PC
        str     lr, [sp, #-4]! ; Decrement SP by 4, then save caller's LR
        str     ip, [sp, #-4]! ; Decrement SP by 4, save caller's SP (now in IP)
        str     fp, [sp, #-4]! ; Decrement SP by 4, save caller's FP
        sub     fp, ip, #4      ; Make FP point to one word (4 bytes) below the
                                ; caller's SP (saved in IP).  This marks the
                                ; start of the stack frame (activation record)
                                ; for the function main().
        sub     sp, sp, #8      ; Create room for two local variables on the stack

        ldr     a4, =a          ; A4 = address of "a"
        ldr     a1, [a4]        ; A1 = 0xE0001234
        ldr     a2, [a4, #4]    ; A2 = 0x00004567 (since A4+4 is address of "b")

        str     a1, [fp, #-16] ; Store value of A1 in the first local variable
        str     a2, [fp, #-20] ; Store value of A2 in the second local variable
                                ; (These local variables are not used any further
                                ; in this program, but could be...)

        bl      larger          ; Call "larger"; this automatically saves return
                                ; address in LR (R14).  Result returned in A1 (R0).

        ldr     a4, =c          ; A4 = address of "c"
        str     a1, [a4]        ; Store result (0x00004567) into c

exit:   mov     ip, fp          ; Use IP as a temporary frame pointer register
        ldr     fp, [ip, #-12] ; Restore caller's FP (ie, value of FP on entry)
        ldr     sp, [ip, #-8]  ; Restore caller's SP
        ldr     pc, [ip, #-4]  ; Return from function (uses saved value of LR)

; ------------------------------------------------------------------------------
; Function: int larger (int first, int second)

; This function compares "first" (in register A1, ie, R0) with "second" (in A2,
; ie, R1) and returns the larger of these in A1.  Registers are preserved/
; destroyed according to the ATPCS.  Only A1 is modified by this function.

larger: mov     ip, sp          ; Save caller's (main's) stack pointer into IP
        str     pc, [sp, #-4]! ; SP = SP - 4, then save current value of PC
        str     lr, [sp, #-4]! ; Decrement SP by 4, then save caller's LR
        str     ip, [sp, #-4]! ; Decrement SP by 4, save caller's SP (now in IP)
        str     fp, [sp, #-4]! ; Decrement SP by 4, save caller's FP
        sub     fp, ip, #4      ; Make FP point to one word (4 bytes) below the
                                ; caller's SP (saved in IP).  This marks the
                                ; start of the stack frame (activation record)
                                ; for the function larger().

                                ; No local variables; parameters in A1 and A2
                                ; (ie, R0 and R1 respectively)

        cmp     a1, a2          ; Compare "first" (A1) and "second" (A2)
        movle   a1, a2          ; Place A2 in A1 if and only if A1 <= A2

larger_exit:                    ; Result is now in A1 (R0)
        mov     ip, fp          ; Use IP as a temporary frame pointer register
        ldr     fp, [ip, #-12] ; Restore caller's FP (ie, value of FP on entry)
        ldr     sp, [ip, #-8]  ; Restore caller's SP
        ldr     pc, [ip, #-4]  ; Return from function (uses saved value of LR)

a:      .word   0xE0001234      ; Variable "a"
b:      .word   0x00004567      ; Variable "b"
c:      .word   0               ; Variable "c"

        .end
```

**Figure 13: Program *larger-v2.s***

By the way, if you compile *larger-c.c* in Figure 11 using level 2 optimisation (**-O2**), you will find that the GNU C Compiler does *not* generate the appropriate activation records for the

function `larger`. This is because the compiler recognises that an activation record is simply not needed (since `larger` does not call any other function, and its variables fit into registers R0–R3, ie, A1–A4), and so the compiler optimises it all away.

> The symbol "`!`" in instructions like "`str pc, [sp, #-4]!`" tells the ARM processor that *two* operations must be done: an automatic addition, then the store operation. In this particular case, "`str pc, [sp, #-4]!`" is equivalent to:
>
> ```
> add     sp, sp, #-4
> str     pc, [sp]
> ```
>
> The ARM instruction set architecture calls this the *immediate pre-indexed mode* for the store instruction.

**Note:** The action of pushing and popping values onto and off stacks is done so frequently that the ARM provides "store multiple" and "load multiple" instructions. Take, in particular, the following instructions in Figure 13 (near the labels `main` and `larger`):

```
str     pc, [sp, #-4]!
str     lr, [sp, #-4]!
str     ip, [sp, #-4]!
str     fp, [sp, #-4]!
```

These four instructions can be replaced by a single "store multiple" instruction:

```
stmfd   sp!, {fp, ip, lr, pc}
```

The `stmfd` ("store multiple, full descending") instruction stores the registers listed between braces "`{`" and "`}`" (in this case, FP, IP, LR and PC) to memory pointed to by the first operand, called the *base register* (in this case, SP). The "`!`" indicates the base register is modified at the end of storing all of the registers. The easiest way to understand all this is to look at what the instruction does:

1. Subtract 4 × number of registers listed from the base register. In this case, SP = SP – 4×4.
2. Store the listed registers to memory in ascending order: the *lowest-numbered* register is stored at the address pointed to by the base register (using its new value), the next-lowest-numbered register at the next word in memory, and so on. Note that listing the registers in a different order will have *no* effect!

   In this case, the base register is SP. Hence, register FP (R11) is stored at [SP], register IP (R12) at [SP + 4], register LR (R14) at [SP + 8] and register PC (R15) at [SP + 12].

The following set of four instructions, found near the exit code of each function (near the labels `exit` and `larger_exit`), can also be replaced:

```
mov     ip, fp
ldr     fp, [ip, #-12]
ldr     sp, [ip, #-8]
ldr     pc, [ip, #-4]
```

These can be replaced by a single "load multiple" instruction:

```
ldmea   fp, {fp, sp, pc}
```

You should examine the programs *larger-v3.s* and *larger-v4.s* on your CD-ROM for more information on how to use these instructions.

**Note:** the *ARM Thumb Procedure Call Standard* specifies that you must preserve the values of registers R4–R11 and R13 within a function. In other words, if you wish to use any of these registers, you must save its value, then restore it later; the stack is the best place to do so. If, for example, you wish to use register R5, you could save it to the stack and restore it later by using the following code fragment:

```
str     r5, [sp, #-4]!    ; Save register R5 to the stack
...                       ; Code that uses R5, potentially many lines...
ldr     r5, [sp], #4      ; Restore R5 from the stack
```

Notice that the `ldr` instruction uses what is known as *immediate post-indexed mode:* the load is done first into register R5, then register SP is automatically incremented by 4.

You can also append the registers you need to save and restore to the list of registers in the `stmfd` and `ldmea` instructions. For example, to store R4–R6, you could use "`stmfd sp!, {fp, ip, lr, pc, r4, r5, r6}`" instead of just "`stmfd sp!, {fp, ip, lr, pc}`".

## Task 4: Stack Frames in Functions

The program *atoi-v2.c* in Figure 14 is a reimplementation of the string-to-integer conversion program *atoi-v1.c* (in Figure 10). The original program has been converted into a function `atoi` that translates a string of digit characters into an integer number:

```
int atoi (char *s)
{
    int i, n;

    n = 0;

    for (i = 0; (s[i] >= '0') && (s[i] <= '9'); i++) {
        n = 10 * n + (s[i] - '0');
    }

    return n;
}

int main (void)
{
    char num[] = "12345";
    int r;

    r = atoi(num);

    return r;
}
```

**Figure 14: Program *atoi-v2.c***

Write an ARM assembly language version of this program that uses stack frames in accordance with the ATPCS, as discussed previously. You should call your program *atoi-v2.s*. Include each of the C statements in Figure 14 as a comment in your code. Use the GNU Tools to demonstrate your working program to the Laboratory assessor.

**Hint:** Make sure that you try a number of arguments to the function `atoi`! You do not need to modify your program or restart the GNU Debugger to do this: since your function `atoi` follows the ATPCS, all you need to do is open the Console window in the debugger and type something like:

```
p atoi("12")
```

Please note, however, that you need to link in the object file *malloc.o* for this to work correctly. The easiest way to do this is by using the **`make`** command:

```
make -f atoi-v2.make
```

**Note:** You can often optimise the code that implements the requirements of the ATPCS: a full stack frame is often not needed, especially if all you are writing is a function that does not call any other functions. If in doubt, write the full stack frame handling code. Otherwise, push and pop just those registers whose values must be preserved in order to conform to the ATPCS.

**Question:** What happens when you pass the string "4294967338" to the function `atoi`? Why? What about the string "4294967295"?

Checkpoint 4: ..... Signature: .....

# Task 5: Division as Iterative Subtraction

The ARM instruction set architecture does not have any direct support for dividing one number by another. If you need this operation, you will have to do it yourself by writing an appropriate function. In other words, you can use other instructions to emulate division.

Dividing a *dividend* (numerator) *x* by a *divisor* (denominator) *y* can be achieved by a series of subtractions. That is, the *quotient* (result) can be calculated by subtracting *y* from *x* as many times as is possible and counting that number of times. For example, $7 \div 2$ can be worked out as:

$$\frac{7}{2} = 7 \underbrace{- 2 - 2 - 2}_{3 \text{ times}} = 1$$

The number of subtractions, 3, is the quotient; this method is called *iterative division*.

The program *iter-div.c* in Figure 15 is a C language implementation of such an iterative division algorithm. Examine this program carefully:

```
int iterdiv (int dividend, int divisor)
{
    int quotient;

    if (dividend < 0) {
        quotient = -1;                  /* Error condition: return -1 */
    } else if (divisor <= 0) {
        quotient = -1;                  /* Error condition: return -1 */
    } else {
        quotient = 0;

        while (dividend >= divisor) {
            dividend = dividend - divisor;
            quotient = quotient + 1;
        }
    }

    return quotient;                    /* quotient = dividend / divisor */
}
```

**Figure 15: Program *iter-div.c***

Write an ARM assembly language version of this program; call it *iter-div.s*. Make sure that you include each C statement as a comment in your code in the appropriate place and that you call your function `iterdiv`. You must also follow the ATPCS. Use the GNU Tools to demonstrate your working program to the Laboratory assessor.

**Hint:** You will need to provide a suitable routine for `main`—you should have enough experience by now to do so!

Checkpoint 5: ..... Signature: .....

Write a formula that computes the number of ARM assembly language instructions executed by the function `iterdiv`, as a function of the dividend *x* and divisor *y*. Your formula should look like something like $f(x, y) = a + ((x/y) \times b)$ where *a* is the number of instructions that are *always* executed, irrespective of the values of *x* and *y*, and *b* is the number of instructions whose execution *does* depend on the values of *x* and *y*. Show your formula to the Laboratory assessor.

Checkpoint 6: ..... Signature: .....

# Positional Division Algorithm

Hand-division uses a series of *left shifts*, *magnitude checks* and *multiple subtractions* to get the final answer. For example, 3217 ÷ 16 can be calculated as:

```
          2 0 1
      ┌──────────
1 6   │ 3 2 1 7  -
        3 2 0 0
      ─────────
            1 7  -
            1 6
          ─────
              1
```

If you examine how you actually arrive at the quotient 201, you will find that you take the following steps without really needing to think about them:

1a. Shift the divisor 16 to the left as many times as is possible, until just *before* it becomes greater than the dividend 3217. This means it is left-shifted by two digits; the shifted divisor is 1600.

1b. Subtract a multiple of this shifted divisor (2 × 1600 = 3200) from the dividend, leaving 17 as the partial remainder. Also shift the multiple 2 to the left by two digits, to produce the partial quotient of 200.

2. In the second iteration, shift the new divisor 1600 right by one digit to become 160. This is greater than the partial remainder 17, so do not subtract anything.

3a. In the third iteration, shift the new divisor 160 right by one digit again, to become 16.

3b. Subtract a multiple of this shifted divisor (1 × 16 = 16) from the new dividend (the previous partial remainder) 17, leaving 1 as the new partial remainder. Add the multiple 1 to the previous partial quotient of 200, giving 201.

4. Finally, stop the iteration here, as no more right-shifts are possible. The old partial quotient of 201 becomes the actual quotient (result); the old partial remainder becomes the actual remainder.

Division in binary is much simpler, as each quotient bit is either a zero or a one: there is no need to find any multiple of the shifted divisor at all. Similarly, the iterative process of working out the partial quotient is just changing the appropriate bit position to 1 as necessary. For example, 3217 ÷ 16 can be worked out in binary as:

```
                          1 1 0 0  1 0 0 1
                      ┌──────────────────────
1 0 0 0 0             │ 1 1 0 0  1 0 0 1  0 0 0 1  -
                        1 0 0 0  0 0 0 0  0 0 0 0
                      ──────────────────────────
                          1 0 0  1 0 0 1  0 0 0 1  -
                          1 0 0  0 0 0 0  0 0 0 0
                      ──────────────────────────
                                1 0 0 1  0 0 0 1  -
                                1 0 0 0  0 0 0 0
                      ──────────────────────────
                                      1  0 0 0 1  -
                                      1  0 0 0 0
                                      ──────────
                                               1
```

The steps are similar to the ones taken for decimal division:

1a. Shift the divisor 1 0000 to the left as many times as is possible, until just *before* it becomes greater than the dividend 1100 1001 0001. This means the divisor is left-shifted by seven bits; this gives 1000 0000 0000.

1b. Subtract this shifted divisor 1000 0000 0000 from the dividend, leaving 100 1001 0001 as the partial remainder. Set the partial quotient to 1000 0000; this is the same as setting bit 7, the initial left-shift, to 1 (remember that the right-most bit is called bit 0).

2a. Shift the new divisor 1000 0000 0000 to the right by one bit to become 100 0000 0000.

2b. Subtract this shifted divisor from the new dividend (the previous partial remainder) 100 1001 0001, leaving 1001 0001 as the new partial remainder. Also add 100 0000 to

the partial quotient (ie, set bit 6 of the partial quotient to 1); the partial quotient is now 1100 0000.

3. Shift the divisor 100 0000 0000 to the right by one bit to become 10 0000 0000. This is greater than the partial remainder 1001 0001, so do *not* subtract anything (and do *not* set bit 5 of the partial quotient).

4. Shift the divisor 10 0000 0000 to the right by one bit to become 1 0000 0000. This is still greater than the partial remainder 1001 0001, so do not subtract anything and do not set bit 4 of the partial quotient.

5a. Shift the divisor 1 0000 0000 to the right by one bit to become 1000 0000.

5b. Subtract this shifted divisor from the previous partial remainder 1001 0001, leaving 1 0001 as the new partial remainder. Also add 1000 to the partial quotient (ie, set bit 3 of the partial quotient to 1); the partial quotient is now 1100 1000.

6. Shift the divisor 1000 0000 to the right by one bit to become 100 0000. This is greater than the partial remainder 1 0001, so do not subtract anything and do not set bit 2 of the partial quotient.

7. Shift the divisor 100 0000 to the right by one bit to become 10 0000. This is still greater than the partial remainder 1 0001. Thus, do not subtract anything and do not set bit 1 of the partial quotient.

8a. Shift the divisor 10 0000 to the right by one bit to become 1 0000.

8b. Subtract this shifted divisor from the previous partial remainder 1 0001, leaving 1 as the new partial remainder. Also add 1 to the partial quotient (ie, set bit 0 of the partial quotient to 1); the partial quotient is now 1100 1001.

9. Finally, stop the iteration here, as no more right shifts of the divisor are possible. The old partial quotient of 1100 1001 becomes the actual quotient (result); the old partial remainder of 1 becomes the actual remainder.

The program in Figure 16 is an implementation of this positional division algorithm:

```
int posndiv (int dividend, int divisor)
{
    int quotient;
    int bit_position = 1;

    if (dividend < 0) {
        quotient = -1;                  /* Error condition: return -1 */
    } else if (divisor <= 0) {
        quotient = -1;                  /* Error condition: return -1 */
    } else {
        quotient = 0;

        while ((dividend > divisor) && !(divisor & 0x80000000)) {
            divisor = divisor << 1;
            bit_position = bit_position << 1;
        }

        while (bit_position > 0) {
            if (dividend >= divisor) {
                dividend = dividend - divisor;
                quotient = quotient + bit_position;
            }
            divisor = divisor >> 1;
            bit_position = bit_position >> 1;
        }
    }

    return quotient;                    /* quotient = dividend / divisor */
}
```

**Figure 16: Program *posn-div.c***

## Task 6: Positional Division in Assembly Language

Write an ARM assembly language version this program and call it *posn-div-v1.s*. As before, include each C statement as a comment in your program in the appropriate place. You must also follow the ATPCS; make sure you try different arguments to the function `posndiv`! Show your working program to the Laboratory assessor.

| Checkpoint 7: ..... Signature: ..... |
|---|

Write a formula that computes the number of ARM assembly language instructions executed by the function `posndiv`. Assume that the average number of shifts for the first `while`-loop is 16. Also assume that 50% of the `if`-statements are true. Show your formula to the Laboratory assessor.

How many instructions do each of these approaches to division (iterative vs. positional) take to divide 0x3FFFFFFF by 1? By 255? What conclusion can you draw from these results regarding the two algorithms?

How would you change the C program in Figure 16 to return the *remainder* instead of the quotient? What would you need to change in your ARM assembly language program?

| Checkpoint 8: ..... Signature: ..... |
|---|

## Mixing C and Assembly Language

The GNU Linker allows you to call code written in assembly language from C and vice versa. Three things need to be done to successfully use assembly language functions from C:

1. Make sure your assembly language code follows the ATPCS,
2. Insert a line in your assembly language code to declare that the relevant function is to be global in scope. This is done using the `.global` assembler directive. For example:
   ```
   .global   posndiv
   ```
3. Insert a function prototype (a declaration) in the C code for the relevant assembly language function. Include the `extern` specifier.

As an example, the C program *div-main.c* in Figure 17 calls the function `posndiv` that you wrote in Task 6:

```
/* Main program that calls posndiv(), defined externally */

extern int posndiv (int dividend, int divisor);
    /* posndiv() is defined in another file */

int main (void)
{
    int a = 23;
    int b = 3;
    int c;

    c = posndiv(a, b);

    return c;
}
```

**Figure 17: Program *div-main.c***

Compiling and linking multiple source code files, whether they are in C or in assembly language, can be a little tricky. One major point that you must keep in mind is that a function can only be defined (ie, implemented) *once* in a given program! And this especially applies to the function `main`. In other words, you cannot try to have `main` defined in both the C program and the assembly language file.

**This is a real problem.** The assembly language programs you have written until now have all included the functions (labels) `_start`, `exit` and `main`. You will need to *remove* these labels (and associate code, of course) to be able to link to C programs.

Once you have written a suitable assembly language file, assemble it as usual:

```
arm-elf-as -marm7tdmi --gdwarf2 -o file1.o file1.s
```

Next, compile the relevant C file:

```
arm-elf-gcc -c -mcpu=arm7tdmi -O2 -g -Wall -o file2.o file2.c
```

Finally, use the GNU Linker to link the two object files *file1.o* and *file2.o*, as well as a third object file *cstart.o*, into the executable *prog.elf:*

```
arm-elf-ld -o prog.elf cstart.o file1.o file2.o
```

The reason you must link in *cstart.o* is that this file provides certain routines that allow C programs to work correctly in the Laboratory. If you like, you can examine the corresponding source code file *cstart.s*, which you will find in your *~/exp3* directory.

By the way, linking in a custom version of the C start-up routines, such as those found in *cstart.s*, is quite common in embedded systems. The start-up routines are usually far more complex when the C program using them must run under an operating system. These start-up routines are normally linked in automatically by the GNU C Compiler; you can find the relevant *.o* files in the */usr/local/lib/gcc-lib/arm-elf/vernum* directory (for some compiler version *vernum*).

## Task 7: Positional Division in C and Assembly Language

Copy the program you wrote for Task 6, *posn-div-v1.s*, into a new file, *posn-div-v2.s*. Modify this new file by removing the functions `_start` and `main` (including `exit`, if defined). Link this new program to the file *div-main.c* in Figure 17. You might like to use the **`make`** command to save you some typing; the make-file is called *posn-div-v2.make*, and can be used by typing:

```
make -f posn-div-v2.make
```

The program generated in this way is called *posn-div-v2.elf*.

Use the GNU Tools to show your working program to the Laboratory assessor.

| |
|---|
| Checkpoint 9: ..... Signature: ..... |