COMP22712
Microcontrollers
Laboratory Manual
2020 Edition

Department of Computer Science
University of Manchester

January 10, 2020

0

Contents

1 | Introduction

1.1   Purpose

The purpose of this course is to:

- gain more experience and confidence in assembly language programming
- learn something of the structure of the low-level embedded software
- probe the hardware/software interface
- interface computers to their environment

1.2   Prerequisites

The course will require some knowledge of assembly language programming and of basic logic design. It is assumed that participants will already have taken COMP15111 and a course in basic digital design (COMP12111). Some of the tools used in these courses will be revisited in COMP22712 — you should therefore bring your own manuals and notes from these to refer to.

COMP22712 is primarily a software course, although a great deal of 'hardware literacy' is involved.

1.3   Teaching Style

This manual describes the entire course, together with some extensions and background material; we recommend you read it — thoroughly. There are no formal lectures although there will be interludes of taught material within the labs. However this is a practical subject and the only way to learn it is by having a go (and making mistakes).

Because of this style all the assessment is also done on the practical work. This means that you must not only complete the course but you must hand in any required evidence so that we know you have done it.

Submit the appropriate exercises by e-mail copied to:

```
james.garside@manchester.ac.uk
antoniu.pop@manchester.ac.uk
```

The exercises in this course are intended to introduce a set of concepts. In general it is necessary to complete the basic exercise before moving on, as some of the ideas — and frequently some of the practical output — are prerequisites for later exercises. Also many exercises include suggestions for extensions; such extensions are not necessary to the course but are there to provide a starting point if you want to learn more.

Like a magazine, this manual has a number of interspersed 'side boxes'. These contain material which is not directly relevant in that place but should come in useful about that time. Each attempts to encapsulate a particular concept which should contribute to both this laboratory and other later courses. Primarily they are there to make your life easier; there are also cases where ignoring their advice will probably waste marks.

You will find occasional 'ARM programming puzzles' filling up odd spaces in the manual. These are purely fun challenges for you to see how close to the target answer you can get. Some of these are really quite hard; if you can work out an answer without help you can feel suitably smug!

1.4   Core Philosophy

The course is focussed on **computer interfacing** with some of the techniques involved: in particular some real-time aspects such as **interrupts**. A selection of **I/O** devices are introduced and a little hardware development is included.

The programming is done in ARM **assembly language**; this does not exactly represent the usual contemporary, industrial flow, where the vast majority of the coding is likely to be in C, but usually helps enormously with the understanding of what is happening inside the machine. Translation of the principles to a programming language in future will be easy.

1.5   External References

There are a number of external references scattered through this manual. These may be to teaching resources, research projects, product data etc. and are usually to information obtainable via the World Wide Web (WWW). In general URLs are not given — WWW pages sometimes disappear and new and better ones are created — instead you should track down your own sources with the search engines of your choice. There is a very powerful research resource literally at your fingertips; use it!

1.6   Errata

Finally, this is a new edition of this manual. If/when you spot something wrong, or something not clear, please feed that back to the staff and we'll try to improve it.

Future students may be grateful to you!

2 | Organisation of the Laboratory

Organisation will be as informal as possible. All sessions are two hours long and will be performed in the Tootill 0 lab. (LF 9). Most sessions will consist of a short talk on aspects of the particular exercise followed by time spent in practical work. If required/desired there will also be time allocated to a discussion of preceding exercises and a chance for feedback.

Please make every effort to arrive on time.

Some other points:

- Certain exercises require paperwork (listings or schematics) to be handed in. You must do this or you will not be credited for that piece of work.

- Many exercises depend on previous work. You should therefore complete exercises in the order given.

- The syllabus assumes that you will spend approximately the same time working on this course outside laboratory time as you do in scheduled lab. hours. The University suggests that each 'credit' is 10 hours[1] work: 10 credits = 100 hours.
  As there are no lectures to read up on and **no examination** to revise for you should spend this time on the lab. Don't expect to keep up if you don't!

- Some exercises include suggestions for further work. These carry no extra marks but may provide some more good practice. If you have time you may attempt some of these; if not then they are a distraction so don't bother.

2.1 Assessment

Marks will be accrued for successfully demonstrating a working design, for efficiency of the solution and for presentation of listings/schematics etc. Not all exercises are formally assessed, but you should still do them!

Exercise	Sessions	Demonstrate	Submission Deadline (approx.)
1-Simple Output	1,2	Not assessed	
2-Less Simple Output	3	Not assessed	
3-Nesting Procedures	4,5	End of Session 5	12/2/2020
4-System Calls	6,7	Not assessed	
5-Counters/Timers	8,9	End of Session 10	
Consolidation	10		28/2/2020
6-Interrupts	11,12	Not assessed	
7-Keyboards	13,14	End of Session 14	13/3/2020
8-System Design/Sounds	15,16	Not assessed	
9-Project	17-18	End of Session 22	—
Easter Vacation			
9-Project (cont.)	19-22	End of Session 22	1/5/2020

Table 2.1: COMP22712 Laboratory Timetable 2020

---

[1]8 may be more realistic

2.2   Equipment Handling Precautions

- The computer building is quite susceptible to building up static electric charges on its occupants. Electrostatic Discharge (ESD) will destroy computer chips. Please make every effort to discharge yourself before handling the lab. boards and observe handling precautions as advised by the staff. If you suspect you may be carrying a charge, painless discharge can be achieved by using a coin, key etc. rather than your finger as a conductor to a ground.

- Short circuits may damage certain devices. Please remove any metal watch straps/buckles, rings etc. before handling the boards. There are no dangerous voltages or currents which will harm you here, but this is a good habit to form for when you may be handling other electrical equipment.

- Connecting and disconnecting I/O systems should be done with the power switched off.

3 | The Laboratory Equipment

3.1 ARM Programmers' Model

The ARM register map and mode encodings are shown below (fig. 3.1) for quick reference.

User	System	Supervisor	Abort	Undefined	IRQ	FIQ
		SPSR	SPSR	SPSR	SPSR	SPSR
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)
R14 (LR)	R14 (LR)	R14 (LR)	R14 (LR)	R14 (LR)	R14 (LR)	R14 (LR)
R13 (SP)	R13 (SP)	R13 (SP)	R13 (SP)	R13 (SP)	R13 (SP)	R13 (SP)
R12	R12	R12	R12	R12	R12	R12
R11	R11	R11	R11	R11	R11	R11
R10	R10	R10	R10	R10	R10	R10
R9	R9	R9	R9	R9	R9	R9
R8	R8	R8	R8	R8	R8	R8
R7	R7	R7	R7	R7	R7	R7
R6	R6	R6	R6	R6	R6	R6
R5	R5	R5	R5	R5	R5	R5
R4	R4	R4	R4	R4	R4	R4
R3	R3	R3	R3	R3	R3	R3
R2	R2	R2	R2	R2	R2	R2
R1	R1	R1	R1	R1	R1	R1
R0	R0	R0	R0	R0	R0	R0

Figure 3.1: The ARM processor architecture

Aliased registers are mapped to the bolder type to their left on this figure; mode encodings are in table 3.1.

Mode	Code	Privileged
User	1 0000	No
System	1 1111	Yes
Supervisor (SVC)	1 0011	Yes
Abort	1 0111	Yes
Undefined	1 1011	Yes
Interrupt (IRQ)	1 0010	Yes
Fast Interrupt (FIQ)	1 0001	Yes

Table 3.1: ARM modes

After reset the ARM processor's state is changed as follows:

- PC := 00000000

- Mode := Supervisor

- Interrupts are disabled (I = F = 1)

- Thumb is disabled (T = 0)

3.2   Software Development Environment

The software development environment used is Komodo, which is the same system used in COMP15111. The only real difference is that programmes will be downloaded to, and be executed on, a real ARM processor rather than a software simulation. This gives access to the real peripheral devices on a stand-alone computer.

3.2.1   Debugging environment

The interface to the laboratory board is provided by "Komodo", a monitor front-end and download tool. This allows the memory on the remote circuit board to be displayed and manipulated, files to be downloaded into memory, and programmes to be executed. The memory and processor register views are updated periodically during execution to aid understanding and debugging. The facilities are introduced gradually in other parts of this manual.

One minor difference is that the mnemonic 'SVC' is preferred to the older 'SWI'. Komodo also allows hardware designs to be downloaded into the FPGAs (Field Programmable Gate Arrays) on the experimental board.

3.3   Circuit Board

The microcontroller laboratory is intended to reflect the development environment used for embedded controllers in the early 21st century. The basic system comprises a software programmable processor — in this case an ARM — with a number of peripheral I/O systems. The microcontroller itself includes parallel I/O, a serial line, a timer and a simple interrupt controller. Much more I/O is provided using an FPGA which can be configured to suit a particular application; this means that both the software and the hardware is programmable and available to the user. The ability to use FPGAs for customisable hardware had considerable influence on the design of systems in the 1990s. As FPGAs have increased in capacity it is now feasible to use them for significant sized systems. The device used in this laboratory has a size of about 200 000 gates.

The PCB (figure 3.2) has a few I/O devices and number of expansion connectors around the board. Two connections are needed for basic operation: a 5 V DC supply must be connected to the power inlet and serial port 0 is used as the host link for controlling Komodo. There is a reset button towards the back of the board but this should not often be needed. The DIP (Dual In-line Package) switches determine the operating mode of the board and should be set to 0001 for the Komodo boot.
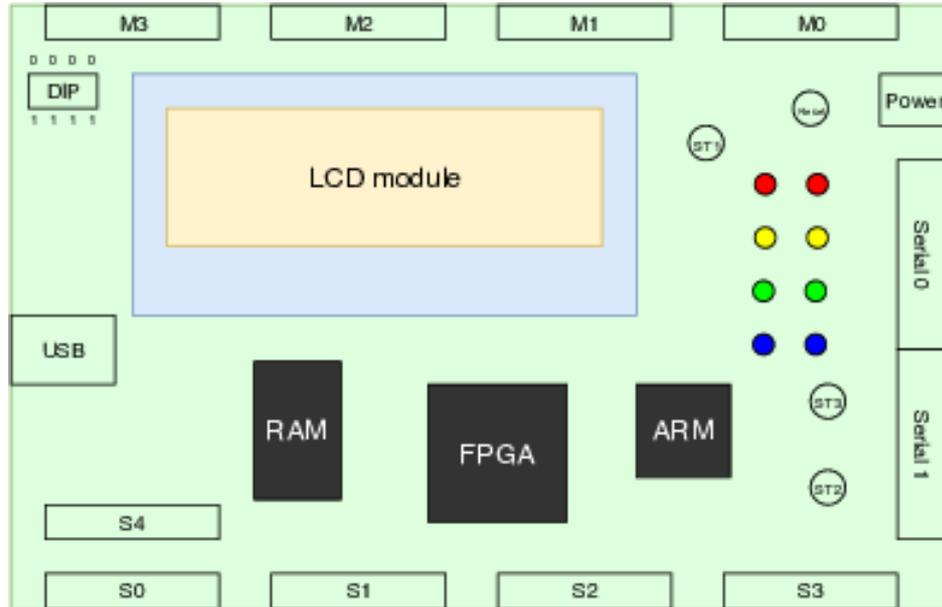
Figure 3.2: The Laboratory PCB connections

The other buttons, LEDs and LCD are introduced as they are required.

The connectors at the top and bottom allow expansion via the Microcontroller and the Spartan3 FPGA. In this lab only the connectors at the front of the board S0, S1, S2, S3 are significant. These are identical; each has sixteen I/O lines and two each of power (3.3 V) and ground outlets.

3.3.1   Lab. board memory map
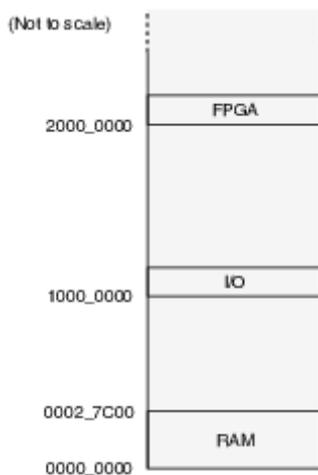
The RAM occupies a space beginning at address 00000000. The lab. boards have 159 Kbytes of RAM available, which means the highest (word) address is 00027BFC.

Remember that the ARM is a 32-bit processor with byte addressing. This means that the addresses of adjacent words are different by 4. Words must be aligned with the memory (i.e. the lowest two bits of the address must be zero).

The microcontroller's internal I/O inhabits the space from 10000000 to 1FFFFFFC. This is 8-bits wide and so only appears in the least significant byte 10000000, 10000004, etc.. The peripheral devices do not fill this space but are mapped repeatedly through the page.

The FPGA occupies addresses 2xxxxxxx and is otherwise similar to the internal I/O space. However it is accessed via an 8-bit bus and addresses are adjacent.

3.3.2   I/O port map

The internal I/O region map (base address `10000000`) is given in table 3.2.

Offset	Direction	Register	Remarks
00	R/W	PIO_A	Bidirectional data for LCD, LEDs etc.
04	R/W	PIO_B	Sundry bits plus PIO_A direction control
08	R/W	Timer	8-bit free running, incrementing at 1 kHz
0C	R/W	Timer compare	Interrupt asserted when timer equals this register
10	RO	Serial RxD	
10	WO	Serial TxD	
14	R0	Serial status	Bit 1 = Tx ready; Bit 0 = Rx ready
18	R/W	Interrupt requests	Raw interrupt requests, active high
1C	R/W	Interrupt enables	Active high
2C, 28, 24, 20	RO	Board serial number	A unique identifier for the particular PCB
20	WO	Halt port	Any write to this address will stop processor execution
3C, 38, 34, 30	—	—	Do not use

Table 3.2: Lab. board peripheral offsets

The bit maps for the I/O ports are given in table 3.3. The interrupt request bits are set or cleared by external events, but may also be written to by software.

Bit		Use		
7	Button (lower)	1 = pressed	Read only	
6	Button (upper)	1 = pressed	Read only	
5	LCD backlight	1 = on		
4	LED enable	1 = on		
3	Extra button (top left)	1 = pressed	Read only	
2	LCD R/$\overline{\text{W}}$	1 = read	also PIO_A direction	
1	LCD RS	1 = data register		
0	LCD E	1 = active		

Table 3.3: PIO_B bitmap

Bit	Use
7	Button (lower)
6	Button (upper)
5	Serial transmitter ready
4	Serial receiver ready
3	Ethernet IRQ
2	Virtex IRQ
1	Spartan IRQ
0	Timer compare

Table 3.4: Interrupt Request/Enable Bitmap

Testing Individual Bits in a Byte/Word

Suppose you have a byte into which are packed the inputs from a set of eight switches. For convenience regard these as active high inputs with Button_0 corresponding to bit zero, Button_1 to bit 1, etc.

Thus, if Button_3 (only) is pressed, the byte (say, in R0) will be &08.

Let's say you want to test if Button_3 is pressed or not. One way to do this would be:

```
CMP        R0, #&08
BEQ        Button_3_pressed
```

This works if only Button_3 is pressed, but what happens if Button_1 (for example) is pressed at the same time?

How many comparisons would you need to ensure correct operation?

CMP is an **arithmetic** function (a subtraction, in fact). In arithmetic functions there can be interactions between bits at different positions; this is useful in some circumstances, but not in others. On the other hand ("bitwise") **logical** functions treat all their bits individually. Logical functions are usually more appropriate when manipulating individual bit values.

The ARM has sixteen data processing operations: eight of these are arithmetic functions, the other eight are logical functions. See if you can classify these in the following table:

Instruction	Arith./ logical	Instruction	Arith./ logical	Instruction	Arith./ logical	Instruction	Arith./ logical
ADC		CMN		MVN		SBC	
ADD		CMP		ORR		SUB	
AND		EOR		RSB		TEQ	
BIC		MOV		RSC		TST	

What two instruction sequence can you write to replace the code fragment above, but which will detect Button_3 being pressed irrespective of the state of the other bits?

3.4 Hardware Development Environment

The CAD environment should be familiar. You should create a Cadence project for this laboratory using the usual set-up script (i.e. "mk_cadence COMP22712") and invoke it as appropriate for this laboratory (i.e. "start_cadence COMP22712").

The hardware designs required in this laboratory will need to be integrated with software too.

For rapid, successful debugging use of **simulation is highly recommended**.

The **synthesis** process is the same as in COMP12111. However downloading the FPGA configuration is not possible from here. The synthesized design can be picked up by the FPGA browser in **Komodo** and can be **download**ed into the FPGA from here. It is advisable to halt any programme execution before doing this; if the programme tries to access a part-loaded FPGA the load process will fail.

4 | Programming Style & Practice

4.1 Structure

Some inexperienced "programmers" think that the choice of language affects the quality of the programming. They bandy terms like 'structure' and sneer that assembly language is, somehow, intrinsically 'unstructured'. One of the main arguments is that there are no "IF ... THEN ... ELSE"s and "REPEAT ... UNTIL"s, only the equivalent of "IF ... GOTO". These people will typically quote Dijkstra[1] as a reference. They are wrong!

To be fair to Dijkstra this is misapplication of his thesis; he cites the ability to jump randomly from place to place in a (high-level) programme as "an invitation to make a mess". *If* this is the case it is an invitation which should be politely declined. Here we get to the point of this section.

Programming is not a function of a language; it is much more than that. The majority of the art lies in analysing a problem and devising an algorithm which will solve that problem, which is implementable, and which is reasonably efficient. This is independent of the language chosen (or imposed).

The danger in assembly language programming is that it is much less restricted than operating in the confined syntax of a language; remember the language is compiled into machine instructions too! It can be much easier to write bad code in assembler than in many languages. It can also be easier to write good code. The point is that it is up to you to ensure that your code has an underlying structure.

There is not the time or space here for a course on structured programming. However here are a few tips:

- Use **procedures** (a.k.a. "functions", "subroutines", "methods") to isolate different identifiable operations. This allows a 'top down' 'divide and conquer' approach which is useful even if the procedure is only called once. In general procedures should communicate through passed **parameters**, which can travel both ways.

  Many high-level languages define functions which can have unlimited input parameters but only return a single quantity; this can encourage bad practice. In assembly language you can pass and return as many variables as you like.

  A procedure should have a well-defined entry and exit in the code. If you are being tempted to jump into the middle of a procedure take a cold bath and then modify your structure. The use of conditional returns can also be frowned upon, although they can prove a boon to efficiency.

- Keep careful control of the **scope** of variables. This is easy within a procedure — local variables are usually nestling in registers — but it requires self discipline not to just read or alter a variable in memory because you can. Resist! Data structures should be maintained by their own library of access procedures. This means that, when it is necessary to edit the code, changes can be handled locally. It is perfectly possible to write object-oriented assembly code if you choose.

- **Loop** structures should normally have a well-defined entry and exit point. Unlike most high level languages these may not be at the same point in the loop. If they are in the same place it is usual to enter at the top and exit at the bottom (when failing to close the loop back to the top again). If the entry and exit points differ it is usually more efficient to jump in and keep the exit point at the bottom[2].

---

[1] Edsger W. Dijkstra, "Go To Statement Considered Harmful", Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148. (http://www.acm.org/classics/oct95/ — go look it up, it's quite short.)

[2] The reason for this is left as an exercise.

- Beyond the obvious issue of the length of a data item assembly language does not have explicit **types**. It is therefore your job to ensure that your variables obey your own typing rules. It is very unusual to require floating point (float, real, . . .) numbers in assembly language but integers, bytes, characters, pointers and more complex structures are common. An integer and a pointer may both be *represented* in a 32-bit word, but they are *different* things; for a start it makes sense to add integers, whereas adding addresses is nonsensical!

  You may choose to adopt a convention to help identify the type of items; for example prefixing pointer names with "p_" makes them fairly obvious.

4.2 Style

Although style is a personal thing, there are some practices which are worth observing to make your source code more readable, robust and maintainable. These are not all specific to assembly language either!

Firstly **layout**. You should be familiar with the general style of assembler source code, i.e.:

```
Label           MNEMONIC  Addresses         ; Comment
```

It is a good idea to adhere to this format — the source code looks neat and labels stand out and therefore are easy to spot. It is also sensible to break up the code with blank lines to highlight groups of related statements (very approximately equivalent to a line of source code in a high-level language).

Within this framework it is possible to enhance readability in other ways. One obvious method is to choose **sensible label names** which reflect their function. As it is often difficult to think of many different names it is usual to name the *entry point* to a module (procedure etc.) sensibly and then use derivatives of this by suffixing numbers for labels inside the module. This has the added bonus of indicating the extent of the module.

The comment field should be used to explain the meaning of the statement. It is usually possible to attach sensible comments to every line. An example of a nonsensical comment would be:

```
    ADD       R2, R2, #1        ; Increment R2
```

However:

```
    ADD       R2, R2, #1        ; Increment loop index
```

conveys significantly more information as it reminds us what `R2` currently holds.

The #1 in the preceding example is one of the few cases where it may be sensible to have immediate *numbers* appear in the address field. It is usually better to equate a number to a label and use that instead. This both conveys added information (the value 'FF' could be used for lots of things, but 'byte_mask' is clear) and allows definitions to be changed in a single place (in a separate header file or at the top of the code) rather than trawling through looking for numerous references and, probably, missing some.

Finally, as the whole programme (being structured) is broken into procedures these can also be emphasised in the source code. Some more commenting can be added to document the procedure's function, its input and output parameters, any non-local data used, any registers which are corrupted, etc. This is useful when trying to reuse or modify pieces of code, and it is the best place to keep this information in that it cannot easily become detached from the code itself.

4.2.1   Example

```
; This routine prints a string terminated by a ``cTTR'' byte
; pointed to by R0.  All register values are preserved.

Print_string    STMFD   SP!, {R0,R1}     ; Push working regs.

Print_str1      LDRB    R1, [R0], #1     ; Get byte, auto-inc.
                CMP     R1, cTTR         ; Test for terminator
                SVCNE   Print_char       ; ... if not terminator
                BNE     Print_str1       ; and loop

                LDMFD   SP!, {R0,R1}     ; Pop working regs.
                MOV     PC, LR           ; Return
```

Note: this will not work in supervisor mode.

Why not?

How could you fix this?

Copy-and-Paste is Deprecated!

Quite often, when programming, you discover that something you want to do is "rather like" something you've already done, with just a few differences. Probably your immediate reaction is to make a **copy** of the earlier code, **paste** it into the appropriate place and **edit** in the differences.

Don't do it!

If there is a lot of commonality between sections of code then they probably should be the same piece of code. This usually means some changes to the 'original' to accommodate some parameters or options and make it more flexible, often involving the extraction of a set of instructions into a procedure/function/subroutine/method/whatever. However the result is almost always a 'cleaner', better product.

Avoiding Copy-and-Paste:

Advantages

- **Readability** improves because, having understood a procedure, it can then be regarded as an abstracted function when it is encountered, rather than being deciphered (with subtle differences) each time.
  Improved documentation often results too - there's less to document!

- **Maintenance** is much easier. If (when) you are given a set of (someone else's) source files and required to "find the bug in there" you will welcome short, obvious procedures rather than 101 'variations on a theme'.

- Programme will be **smaller**. This may be vital in embedded controllers, where memory space is limited. Often speed increases too as a smaller memory image means fewer cache misses, page faults[a], etc.

Disadvantages

- The code may run slightly slower due to procedure call overheads.
  (This can be countered by using a **macro**[b] rather than a procedure.)

- Your productivity in number-of-lines-of-source-per-day falls significantly :-}
  (Productivity in terms of (better) products-faster improves though.)

---

[a]Don't worry if you don't know these terms; you will meet them in the future.
[b]A macro is a substitution where you can define your own keywords; each time a macro name is encountered the assembler (or compiler) substitutes the appropriate code.

5 | Exercise 1: Simple Output

5.1 Objectives

- Hardware/software familiarisation exercise

- Bit manipulation

- Simple I/O

5.2 Equipment

The laboratory equipment comes in two parts. First there is a software 'front end' which is provided by the Komodo software, together with the ARM software development tools and the Cadence/Xilinx hardware development tools. Secondly there is a hardware 'back end' which is provided by the experimental board (a computer in its own right), its embedded software and any daughter boards for specific, later exercises. These two parts are connected by a serial link.

Ensure that the serial line is in place (serial port 0 on PCB, serial port 2 on the PC) and the power supply is connected. First switch on the board to start the back end, then type "**start_komodo 22712**" in a shell window to start the front end. The two parts of the system should synchronise and a window with various controls should appear.
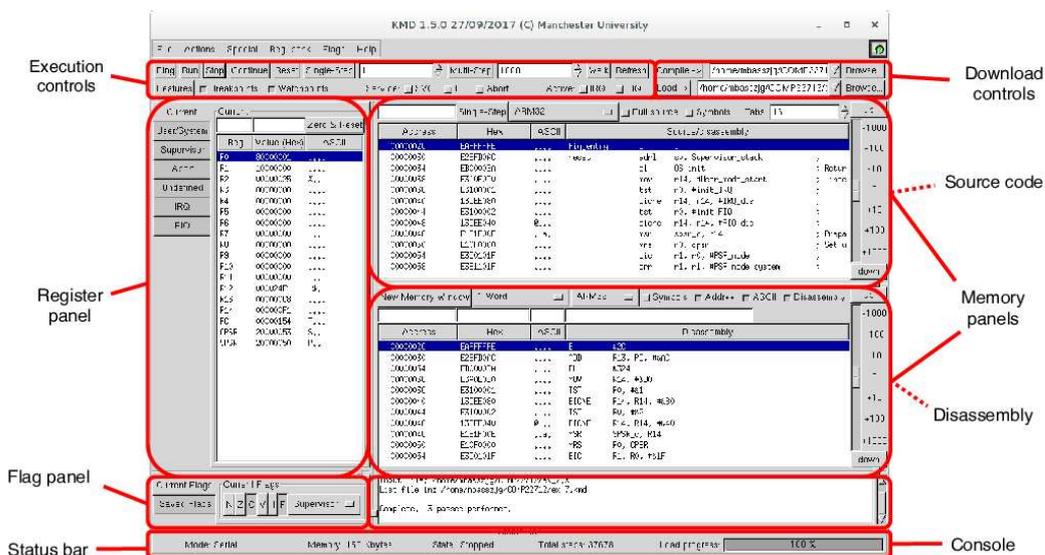

Figure 5.1: The Komodo front-end

The Komodo window (figure 5.1) has a number of areas which allow the memory and registers to be displayed and manipulated, the processor's execution to be controlled and programmes to be downloaded into both the memory and the FPGA. These are described below. **N.B. in this laboratory, you should be in "Serial" mode; check the bottom left corner**.

5.2.1 Manipulating memory

Komodo has several independent subwindows which allow the display of memory and processor registers. A register window displays a particular view of the ARM's register set. The default view is of the current register set, i.e. the mode is specified by the value in `CPSR`. These are the registers which will be accessed by almost all instructions as they execute. However it is possible to display a register view in any mode.

Clicking on a register copies is contents to the top of the subwindow, where its value can be changed by the user.

Memory views show a small area of the address space and can display the memory in various forms, primarily as different length hexadecimal numbers, ASCII and as disassembled ARM instructions. Hexadecimal is most useful for viewing data space whereas disassembly is convenient for following programmes. Extra (larger) memory windows can be created if needed.

It is possible to scroll the memory window around the address space, or move it to a particular address by typing this at the top of the window. It is also possible to define the window address in terms of a register's contents. For example, an expression such as "`PC-8`" will maintain a widow which begins eight bytes (two instructions) before the current `PC` value — this is particularly useful in association with disassembly, as the window will then scroll as the programme executes.

Note: **all numbers are hexadecimal** and no prefices/suffices are required.

5.2.2 Developing code

There is no special text editor associated with this laboratory. Develop your source files using your own preferred tool. It is, however, recommended that you first create a directory to keep your files together. Assembler source files should be identified with a ".s" extension.

Don't forget to save the file before proceeding.

5.2.3 Assembling

Assemble using the 'compile' facility provided with Komodo. This is available via the browser in the top right-hand corner. This runs a script which will assemble a single, specified file. By default the file will be loaded at address 00000000, but this may be altered for all, or part of the object code by using the `ORG` directive. The output can be downloaded directly using the browser immediately below.

This utility only allows a single file to be assembled. Later it is possible that you may want to link several **source files** into a single **object programme**. One way of doing this is to "include" other source files into the specified file using the directive:

```
INCLUDE <filename>      ; "GET" is synonymous
```

More sophisticated linking should not be needed in this lab. but if required can be done independently from Komodo using the ARM SDT, GNU tools, or any other tool you may choose.

5.2.4 Downloading

The assembled output is in 'list' file with the suffix ".`kmd`"; this is a human-readable, ASCII form. This can be located by the Komodo file **browser** and loaded into memory by pressing 'Load'.

If a new file is created in a directory the browser needs to be refreshed to pick it up. Use "./".

Komodo can also accept other input formats, notably a standard, portable binary form known as ELF (Executable and Linking Format) can be read.

5.2.5   Executing code

When the system is started the back end is in a 'Reset' state. This state can also be reached using the reset button (near the back of the PCB) or, more 'cleanly', with the reset button on the Komodo window. When the ARM is reset the `PC` is set to `00000000` and the `CPSR` is set to `000000D3`, which indicates that interrupts are disabled and the processor is in supervisor mode. Normally the ARM would immediately begin executing (from address `00000000`) but in this case the processor is halted so that the state can be observed.

There are several ways in which a programme can be executed. The simplest is to use 'Run' which simply begins normal execution. This is suitable for debugged programmes but gives very little help if there are bugs present. Another mechanism is to use the 'single step' function, which executes a single instruction before halting the processor again. This makes it convenient to observe exactly what each instruction does. When single stepping it is often convenient to display a disassembly listing of the code in a memory window.

With longer programmes stepping each instruction individually can be tedious; Komodo also allows a programmable 'multi-step' facility which is useful, for example, in stepping through larger areas of debugged code. It is also possible to 'walk' through code by multi-stepping (could be set to "1") at a user definable step rate; automatically running a few instructions per second can make some tasks much easier to follow. A set of option buttons (which default to off) allow some compound operations to be considered as a single step. Initially the only useful ones will be `BL` and `SVC` which treat procedure and system calls as one step, respectively. Warning: these detect the end of the call by checking for the return address; although they should handle nested calls (including recursion) correctly a 'badly behaved' call (one which does not return to the expected address) will not terminate[1]. If this is encountered the processor will "run" and the 'stop' button will be needed to regain control.

Any of these operations can be stopped using the relevant on-screen button.

Komodo supports other methods of suspending execution which will be described later.

5.3   Input and Output Ports

To be at all useful a computer system must have some input and output (I/O) capability. The simplest I/O is provided by a **parallel port** which essentially maps a memory location into some real hardware. In the case of an output port this means that the state of the bits stored in this 'memory' are used to control some external hardware, such as an LED (Light Emitting Diode). In the case of an input port the 'memory' is some outside world device — for example a single bit's state could come from a push button switch. The devices which inhabit this space are known as **peripherals**.

In some processors there is a special I/O address space (outside the normal memory map) with special instructions which can be used for I/O peripherals. However on most RISC processors, including the ARM, there is only a single address space and so some memory locations are sacrificed for I/O; this is not a significant problem when there is 4 Gbytes of addressable space.

It is frequently the case that peripheral devices do not use the full 32-bit bus; most common devices have only an 8-bit interface. This is reflected in this laboratory where all the I/O devices will be accessed eight bits (or fewer) at a time. When communicating with I/O ports you should therefore remember to use byte load and store instructions (**LDRB**/**STRB**).

5.3.1   Bit manipulation

Although some I/O devices are byte wide (or, occasionally, larger) many inputs and outputs are smaller — often single bits. For example the position of a switch or button can be represented with a single bit. It is usual to cluster several (functionally connected) I/O bits together into partial or full bytes to simplify the hardware requirements; an example is used in this exercise. Because the smallest quantity which can be addressed is (usually) a byte the issue of **bit addressing** must be handled in software. Bits are normally addressed such that bit 0 is the least significant bit.

---

[1] An example would be the "print following string" routine.

Action on bit	AND with ...	XOR with ...
None	1	0
Clear	0	0
Set	0	1
Toggle	1	1

A 'trick' worth knowing is that any bit manipulation can be performed using two successive bit mask ANDed and XORed with the byte in question.

This is also useful when designing hardware.

The ARM can alter a single byte in memory with a STRB instruction. However if it needs to change a single bit it cannot modify it without, potentially, changing the other seven bits in the byte. To avoid this the other bits must be written to their original value.

To do this the programmer must first find the original value of the byte. Usually this can be done by first loading the byte and altering the bit(s) concerned in a register. Individual bits can be set by ORing the value with the appropriate bit mask; bits may be cleared by ANDing with the complement of this mask or, on the ARM, using the BIC (BIt Clear) instruction. It is a simple and worthwhile exercise to learn the basic bit mask as shown in table 5.1.

Bit Number	OR (BIC) mask	AND mask
7	80 (1000 0000)	7F (0111 1111)
6	40 (0100 0000)	BF (1011 1111)
5	20 (0010 0000)	DF (1101 1111)
4	10 (0001 0000)	EF (1110 1111)
3	08 (0000 1000)	F7 (1111 0111)
2	04 (0000 0100)	FB (1111 1011)
1	02 (0000 0010)	FD (1111 1101)
0	01 (0000 0001)	FE (1111 1110)

Table 5.1: Hexadecimal (binary) bit masks

A bit can be 'toggled' (changed) — without knowing its initial value — by XORing it with a "1".

Thus to set bit 5 of an 8-bit port the following sequence could be used:

```
LDRB    R0, [port_address]
ORR     R0, R0, #&20
STRB    R0, [port_address]
```

(Remember that all ARM memory addressing must be relative to a register!)

In this first exercise a single 8-bit port is used for output. The bits within this port each correspond to an individual LED and are *active high* (i.e. a "1" turns the LED on). The port bits are read/write so the value written can always be read back.

Modern computers execute millions of instructions per second. It only takes a few instructions to change the state of a few LEDs. To make the exercise viewable by a human the steps must have 'reasonable' intervals between them. This could be done by using a **delay loop** — a construct which repeats wasting a little time many times over — but this is usually **bad** practice for several reasons[2].

Bit	LED	
7	Blue	(Right)
6	Red	(Right)
5	Amber	(Right)
4	Green	(Right)
3	Blue	(Left)
2	Red	(Left)
1	Amber	(Left)
0	Green	(Left)

---

[2]These should become apparent later.

5.4   Practical

Write a programme which cycles a set of 'traffic lights' attached to the output port at address `10000000`. Each light is controlled by a single active-high bit where 0 = light off, 1 = light on. The cycle should follow the same sequence which two sets of lights controlling a crossroads would do, and it should run continuously at a rate comfortably viewable by a lab. TA (e.g. table 5.2).

State	Left hand lights	Right hand lights	Bit pattern	Hold for about ...
0	Red	Red		1s
1	Red & Amber	Red		1s
2	Green	Red		3s
3	Amber	Red		1s
4	Red	Red		1s
5	Red	Red & Amber		1s
6	Red	Green		3s
7	Red	Amber		1s

Table 5.2: 'Traffic light' sequence

Note that states #0 and #4 *appear* the same, but are actually different states.

(The bit patterns are left for you to fill in!)

For this first experiment the timing reference can be a delay loop (an empty loop iterated about &80000 times should suffice). However, to allow this to be upgraded later, it is sensible to define a 'delay' procedure which can be called with an appropriate parameter.

A much more stable timing reference can be obtained by using a hardware timer which will be introduced in a later exercise.

Note: the lab. board is a stand alone computer. You can prove this by pulling out the serial cable, logging off etc. after your programme has been started.

5.5   ARM Procedure Calls

Procedures are called using the **BL** ("Branch and Link") instruction. This branches to a specified label but saves the **return address** (i.e. the address of the following instruction) in `R14` — also known as the **Link Register** ("`LR`"). Any existing contents of the link register are lost.

To return from the procedure this saved value must be returned to the Programme Counter ("`PC`"). This may be done with the following instruction:

```
        MOV     PC, LR          ;
```

5.6   Advanced

Modify you programme to imitate a 'Pelicon' pedestrian crossing, which has a different number of states and uses flashing lights to indicate some states.

Bit(s)	Button
7	Lower
6	Upper
5-0	Unused in this exercise

Traffic lights are not always 'dumb' and may react to input stimuli. Modify your programme so that an input can 'trip' the lights to the appropriate state. Two input buttons are available towards the lower right of the circuit board for this function, which can be read as active high bits in a port at address `10000004`. Attempts to write to these bits will be ignored.

If attempting this remember that the inputs may change the traffic light timing, but not the sequence (which would be dangerous!) and that the button presses may be momentary, and on either (or both) buttons at any time.
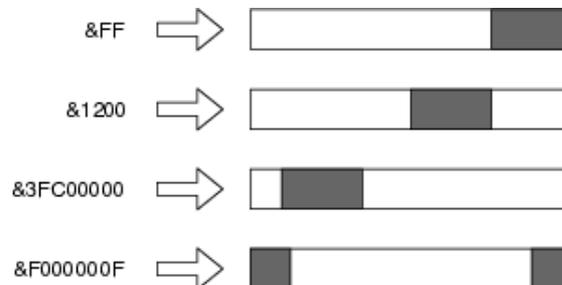
A modified state diagram might help here.

ARM immediate constants

ARM data operations have access to immediate fields which are formed by rotating an 8-bit pattern by an even number of bits around the 32-bit word; the other bits are all zeroed.

Some example values are shown below. (Note that not all constants are obtainable.)

A useful pseudo-instruction

The ARM assembler provides a pseudo-instruction to allow easy access to arbitrary immediate numbers. The operation:

```
LDR     Rd, =expression
```

can plant a 32-bit value in memory (the evaluated expression) and an instruction to load this into the specified register. Note that this requires two words and is slower than a MOV, but may be useful for getting at 'awkward' constants.

A case in point may be the address of an I/O port.

Accessing I/O ports

All ARM memory accesses are made relative to a register. I/O is mapped into the memory space, thus a register is needed to point to the I/O port. Usually several different I/O ports inhabit a small area of memory.

Rather than loading a register with the exact address of the port each time a different port is used, consider pointing a register somewhere 'nearby', and using predefined offsets to get at the ports. For example:

```
Port_area       EQU        &10000000
LED_port        EQU        &0
Buttons         EQU        &4
                ...
                MOV        R8, #Port_area
                LDRB       R0, [R8, #Buttons]
                STRB       R0, [R8, #LED_port]
```

This is faster and usually more legible. It also uses fewer registers if several ports are in use.

I/O ports in C

A 'problem' with programming in a more abstracted language is that sometimes you need to use I/O ports which are at *known*, fixed addresses. The usual solution to this in C is to set up pointers containing the relevant addresses explicitly. E.g.

```
const char* Port_A = 0x10000000;
```

This can then be used indirectly for access to the 'variable'.

```
*Port_A = value;
```

6 | Data-driven and Code-driven Algorithms

In the previous exercise you implemented a finite state machine in software.

You probably did this as a '**code-driven**' algorithm. This means that the current state is marked by the position in the code or, if you prefer, is implied by the value of PC. A code driven algorithm is illustrated below; this probably makes more decisions than your traffic lights.

```
state_0  do things
         IF inputs = 0 THEN GOTO state_1
         ELSE IF input = 1 THEN GOTO state_5
         ELSE GOTO state_2

state_1  do other things
         ...
```
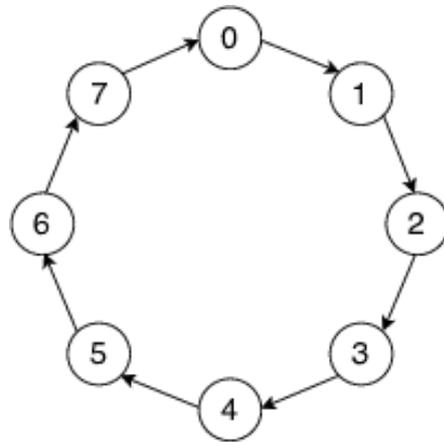
An alternative method is to write a '**data-driven**' algorithm where the state is held in a state variable:

```
loop          CASE state
              0: CALL state_0_code
                 state := state_table_0[inputs]
              1: CALL state_1_code
                 state := state_table_1[inputs]
              2: ...
              GOTO loop

state_table_0  DEFW 1, 5, 2, ...
state_table_1  DEFW ...
```

In this second example one short piece of code covers any state machine and the behaviour is **looked up** in data tables.

The code-driven example is probably more intuitive and is easier to write — at first. It will probably also be shorter (no large data tables) and faster (no indirecting through memory).

The data-driven example is more regular and is thus easier to modify, expand and maintain.

Which is the "correct" solution depends on the task in hand and the programmer's own prejudices. They are presented here simply as alternatives for consideration.

6.1 Look-up Tables

A look-up table is a data table (array) placed in memory which can be indexed by an input variable. The state tables on the preceding page are examples of look-up tables.

Look-up tables are very useful for certain tasks. For example this would be by far the most efficient method of translating a 4-bit number into a bit pattern for a seven-segment display.

```
          AND     R0, R0, #&F      ; Ensure input in range
          ADR     R1, Seg_table    ; Point at table
          LDRB    R0, [R1, R0]     ; Load pattern
          ...

Seg_table DEFB    &3F, &06, &5B, &4F
          DEFB    &66, &6D, &7D, &07
          DEFB    &7F, &6F, &77, &7C
          DEFB    &39, &5E, &79, &71
```

This is also quite fast because an arbitrary function can be calculated with a single memory load.

The look-up can be more complex than a single entry. For example a character set may be specified as a set of pixels with one byte (8 bits $\Rightarrow$ 8 pixels) per line, and the character number would need to be multiplied[1] by the number of bytes in each character. For an 8×8 character matrix:

```
Char_set  DEFB    &00, &00, &00, &00   ; Character 0
          DEFB    &00, &00, &00, &00   ; (cont.)
          DEFB    &80, &40, &20, &10   ; Character 1
          DEFB    &08, &04, &02, &01   ; (cont.)
          ...
```

This may be useful if you want to explore further in the next exercise.

Look-up tables can also be useful in evaluating functions. For example "X/Y" — where X and Y are 8-bit numbers — can be looked up as:

```
          LDR     R0, X                ;
          LDR     R1, Y                ;
          ADD     R0, R1, R0 LSL #8  ; Calculate index
          ADR     R1, Div_table        ; Point at table
          LDR     R0, [R1, R0]         ; Get result
```

This is fast, but expensive in memory requiring a $2^8 \times 2^8 = 2^{16} = 64$ Kbyte look-up table.

Worse, a 16-bit division would require 8 Gbytes, more than the ARM can address. Long division is therefore usually done iteratively.

---

[1]Easiest if the number is a power of two so the multiplication is simply a left shift.

7 | Exercise 2: Less Simple Output

7.1 Objectives

- Parallel output/handshake sequencing

- Procedures and parameter passing

- A 'real-world' device

7.1.1 The Liquid Crystal Display (LCD)

Character-based liquid crystal displays (LCDs) contain some 'intelligence' and use a (reasonably) standard interface. The interface is a parallel bus interface which comprises data and control signals. The data bus can be 4-bits or 8-bits wide; the 4-bit mode is used when there is a very limited number of I/O signals available and need not concern us here.

The interface signals are:

Name	I/O	Function
DB[7:0]	I/O	8-bit bidirectional data bus
R/$\overline{\text{W}}$	O	Read not Write 0 = write, 1 = read
RS	O	Register select (address line) 0 = control, 1 = data
E	O	Enable High to validate other signals

Table 7.1: HD44780 LCD Controller Interface Signals

The interface gives access to two 8-bit registers on-board the LCD controller. These are designated 'control' and 'data' and are selected by the state of the RS output. *Note: these registers are not* <u>directly</u> *visible to the processor.*

Like many I/O devices reading an LCD register may not give back the value last written to it. The type of operation performed is controlled by the R/$\overline{\text{W}}$ output.

To communicate with the display the control lines must be set to the appropriate values and then **strobed** using the enable line (fig. 7.1). If the enable signal is inactive the other signal states do not matter. During a transfer the signals should be stable and the usual constraints of set-up and hold times must be met.

When writing to an output port the bits will change approximately at the same time. To provide the correct timing the transfer will therefore require at least three separate output commands:

1. Set up data and command
2. Set enable line active
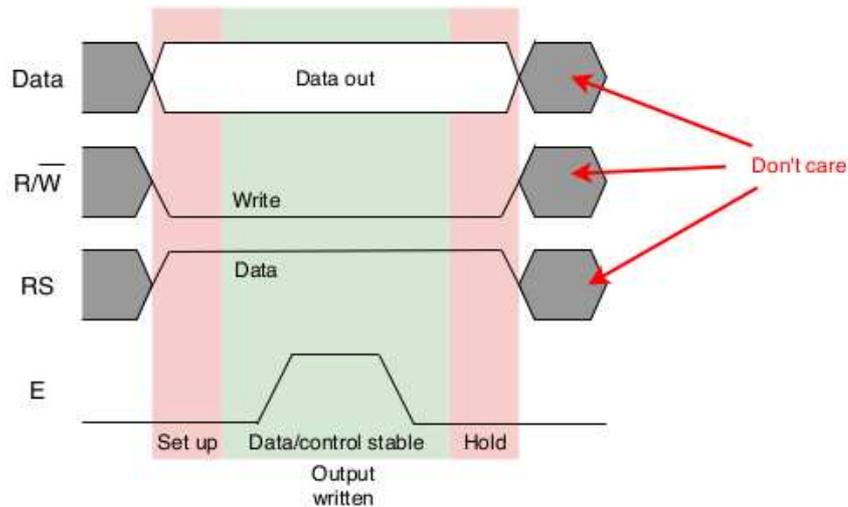3. Set enable line inactive

Figure 7.1: LCD data write cycle

Note that it is not necessary to 'remove' the data; it can remain until overwritten by the next command. However it may not be possible to complete step 1 with a single operation if, for example, only one eight bit port can be accessed at a given time (as there are ten bits to change).

It is also necessary to ensure that there is enough time *between* these steps for the LCD controller to detect and respond to the command; for example, if step 3 follows step 2 too closely the enable pulse may not be detected at all. It may be that the time between instructions is great enough to ensure this, if not the driving software must provide the appropriate delay. Fortunately, in this case, the time between adjacent instructions is longer than the minimum write pulse width.

The LCD controller is a HD44780, a standard part. It is smart enough to obey a small set of instructions. The simplest of these are:

- Print a character (and move cursor to next space)

- Clear the screen

A few other commands enable the cursor to be moved, the display to be scrolled and user defined characters to be downloaded. Data on the display controller may be found on the WWW if required[1].

Each command takes time to process. The LCD controller will generally be somewhat slower than the processor controlling it. It is potentially quite easy to try to send the next character before the previous one has been printed. As this will cause confusion it is important to wait until one command is finished before sending the next. There are two ways of achieving this:

- Wait for at least the minimum command time

- Wait until the display controller is not busy

The first option is the less efficient because:

  ⋄ absolute times are relatively hard to measure
  ⋄ the CPU could be doing something else for some (or all) of this time

The second option is better because:

  ⋄ the CPU only waits as long as necessary
  ⋄ it is self-adjusting to varying times for different commands, different LCD modules or different processor speeds

---

[1]Try: http://home.iae.nl/users/pouweha/lcd/lcd.shtml or find a site which suits you.
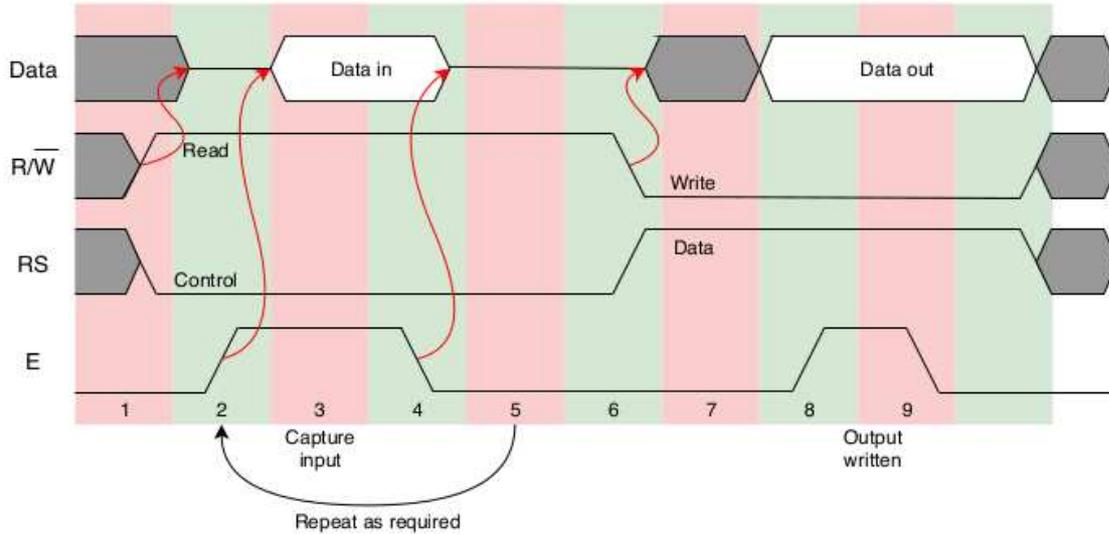
Figure 7.2: LCD character output sequence

However the second option (fig. 7.2) does require some feedback from the LCD. To obtain this the LCD control register must be read and bit 7 will then indicate the controller's status (0=idle, 1=busy).

To read the control register the appropriate command must again be sent, in this case RS=0, R/$\overline{\text{W}}$=1. Before enabling the display the data bus — which will normally be an output — must be set to be an input. This then 'floats' the bus (i.e. it becomes tristate) so that, when it is enabled, the LCD controller may drive it. If this is not done two different values may be driven from each end of the bus which will lead to an undefined value, excessive power dissipation and, possibly, damage to the components.

The sequence for writing a character now becomes:

Wait until LCD controller is idle	1	Set to read 'control' with data bus direction as input {R/$\overline{\text{W}}$=1, RS=0}	Port B
	2	Enable bus (E :=1)	Port B
	3	Read LCD status byte	Port A
	4	Disable bus (E :=0)	Port B
	5	If bit 7 of status byte was high repeat from step #2	
Write character	6	Set to write 'data' with data bus direction as output R/$\overline{\text{W}}$=0, RS=1	Port B
	7	Output desired byte onto data bus	Port A
	8	Enable bus (E :=1)	Port B
	9	Disable bus (E :=0)	Port B

The action of this on the interface signals is illustrated in figure 7.2. It all sounds like a lot of fiddling about, but each step is only one or two instructions. Of course once the routine is written it can be used to output all the characters you'll ever need.

Writing a control byte is similar — the only difference is the register addressed in step #7 (for the control register RS=0). The command for "clear screen" is 01. There are plenty of WWW sites which will help you find out about other commands if you want to do more; searching for "HD44780" is a good place to start.

7.2   The LCD Interface

The LCD uses eleven interface signals: eight of these form the data port[2], the other three being control as shown in table 7.2. These are connected to two 8-bit ports which we will call 'port A' and 'port B'. To reduce the number of I/O signals required port A (the LCD data port) is the same 8-bit port which is used for the on-board LEDs; port B (address:10000004) (table 7.2) provides the other required bits. Figure 7.3 shows the hardware configuration used.
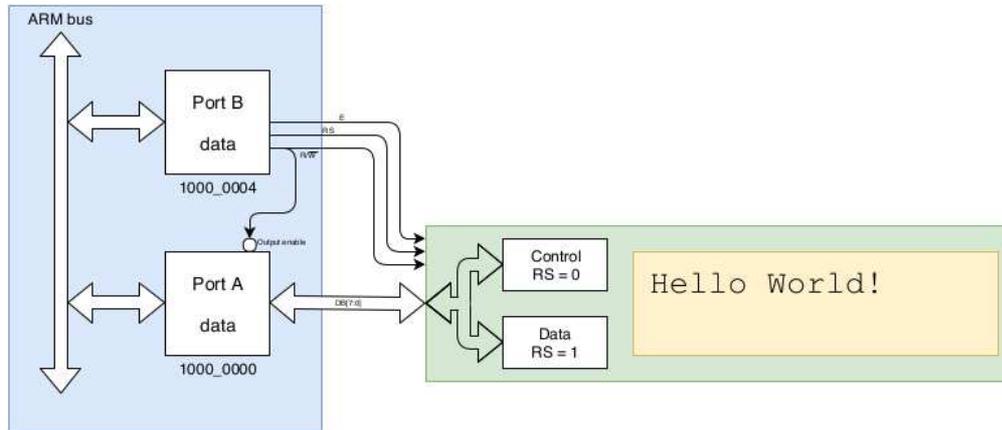

Figure 7.3: LCD controller interface

In this exercise the data port is *bidirectional* so there must be a means of disabling its output. The (active low) output enable for all eight bits is connected to port B bit 2 (LCD_R/$\overline{\text{W}}$); this ensures that port A can never be outputting at the same time as the LCD module.

Reading port A returns the values at the device pins. Thus reading port A when it is enabled as an output should return the value written to the port. Reading it when the output is disabled can return other values; if the LCD is also enabled this will drive the signals; if the LCD is disabled the port will be floating and the value will be undefined.

Note that, if the LEDs flashing as the LCD is used is distracting, they can be turned off using bit 4 of port B. The LCD backlight can be controlled using bit 5 of the same port.

Port	Bit(s)	Direction	Signal(s)	Active state
A	7-0	in/out	data[7:0]	—
B	7	in	Button (lower)	1 = button pressed Writes ignored
B	6	in	Button (upper)	1 = button pressed Writes ignored
B	5	out	LCD Backlight	0 = off 1 = on
B	4	out	LED Enable	0 = off 1 = on
B	3	in	Extra button (top left)	1 = button pressed Writes ignored
B	2	out	LCD R/$\overline{\text{W}}$	0 = write to LCD module 1 = read from LCD module
B	1	out	LCD RS	0 = control register 1 = data register
B	0	out	LCD E	0 = interface inactive 1 = interface active

Table 7.2: LCD interface ports

---

[2]It is possible to programme the interface to use only four bit data. This saves signals (and the remaining 7 fit neatly into a single 8-bit port) but means that two strobes need to be made for every transfer.

When writing characters the LCD controller accepts standard ASCII printing characters (see page 88) with a few variants such as '¥' instead of '\'. A Japanese characters set and a few Greek/mathematical symbols are also included using codes from A0-FF; you may see some of these if something goes wrong! You can also define some of your own characters.

7.3   Practical

Write "Hello world!" (or some similarly trite message) onto the LCD display.

7.4   Suggestions

Write a routine to print a single character and call it repeatedly to print the string; the character may be passed in a register as a parameter. You will need this routine (or a derivative) later, so take care to comment the code.

The procedure for outputting characters is common to all characters. Outputting control and data is the same process except for the state of one bit. It may be sensible to combine these operations into a single procedure with an extra parameter.

7.5   Advanced

Try outputting hexadecimal numbers using the same character output routine.

You might like to try writing to the bottom line of the display.

Try out some other features of the HD44780 controller, such as moving the cursor, scrolling the display, or even defining and displaying your own characters.

Define some non-printing control characters which, when 'printed', have some effects on the display.

7.5.1   Commonly used control characters

Some common ASCII character functions are:

Byte	Name	Commonly used to . . .	HD44780 operation code(s)
08	Backspace (BS)	Move the cursor left one place	
09	Horizontal Tabulate (HT)	Move the cursor right one place	
0A	Line Feed (LF)	Move the cursor down one place	
0B	Vertical Tabulate (VT)	Move the cursor up one place	
0C	Form Feed (FF)	Clear Screen	
0D	Carriage Return (CR)	Move the cursor to the start of the line	

Table 7.3: Common control characters

Note: writing these characters to the display will *not* achieve what you want. They must be **translated into codes or *sequences* of codes** which manipulate the HD44780 control register. However it is often very useful to be able to (for example) place a 'newline' code (such as '\n') directly into a string.

If you're very serious you could look up the set of **ANSI escape codes**.

Let the assembler do the work . . .

Some expressions are static — i.e. they can be evaluated at assembly ("compile") time. If this is the case they should be, saving instructions at run time (both time and space).

For example, to set two bits in a register:

```
        ORR     R0, R0, #(bit_6 OR bit_1)
```

8 | Programming Hints

Code reuse is a very good idea; it saves you time and sweat in the future. However to make code easily reusable it has to be easy to modify, even when you've forgotten the details of how it works.

Here are some tips which will assist you in writing code now and will help you in future.

8.1 Setting Constants

Most programmes will contain various immediate constants. For example a string printing routine may detect certain 'control characters' which cause actions to happen rather than characters to appear. A common example would be "carriage return" to start a new line.

A subset of the ASCII character set (see page 88) is reserved for such functions although many of these are not now used for their original 'intended' purpose. CR is character 0D.

The statement:

```
        CMP     R0, #&0D
```

can check for such a character in a string. This works fine but can be unclear. It is much more obvious to define a label at the start of a programme and use this consistently throughout.

```
CR              EQU     &0D

                ...

                CMP     R0, #CR
```

This makes the source code more readable, and distinguishes the "0D" used here from the same value used elsewhere. Even better use your own name for the constant. This allows easy modification if necessary; for example although many systems use 'CR' to move to a new line, Unix uses 'Line Feed' (LF). The excerpt below illustrates how this could be set up so that a single change can be used to modify all references in the code.

```
c_LF            EQU     &0A
c_CR            EQU     &0D
                ...
Newline         EQU     c_CR    ; Define appropriate character

                ...

                CMP     R0, #Newline
```

Note that, although the source code is longer the object code generated is the same in all these examples.

Search-and-replace with an editor is not a substitute for this. It is far too easy to modify something you didn't mean to.

Numeric constants

Komodo displays everything in hexadecimal; this is typical in a debugger.

Like most development tools the default base (or "radix") for numbers in the assembler is 10, i.e. "decimal". This is because it is the default for (most) people. However, in your source code you should use **whichever base makes your code most readable**: sometimes this will be **decimal** ("250"), sometimes it will be **hexadecimal** ("&F0") and, sometimes, **binary** (":11110000").

8.2   Header Files

It is usual to have a large number of constant definitions at the start of a piece of source code. These may include items such as control characters, your own enumerations (e.g. "TRUE" and "FALSE"), processor specific features (e.g. "`Supervisor_Mode EQU &3`"), system specific constants (e.g. "`LED_port EQU &10000000`") and application specific numbers (e.g. "`Retry_count EQU 10`").

Many of these will remain the same for various projects and can be copied when starting a new source file. However such lists grow in size and this 'maintenance' may well need applying to all the source files. Therefore it is usually sensible to keep the majority of constants (all except the application specific ones) in a separate file which can be included at assembly time.

In the ARM assembler these two directives are synonymous:

```
INCLUDE filename.s

GET     filename.s
```

In practice more than one header may be sensible. One can contain items which are immutable (such as the bit masks for the processor's flags) while another has system information such as the memory size and I/O port locations; that way the code can be ported to a different system simply by changing a single inclusion name.

It may also be sensible to split your projects into more than one file. A sensible split is to divide any 'system' code — which would encompass the ARMs exception 'vectors' and trap `SVC`s et al. — from the user-mode applications code. E.g.

```
Reset           B       Start_of_code           ; Reset (address 0)
                B       Undef_handler           ; Undefined instruction
                B       System_call             ; SVC call
                B       Prefetch_abort          ; Page fault
                B       Data_abort              ; Page fault
                NOP                             ; Unused 'vector'
                B       IRQ_service             ; Interrupt
                B       FIQ_service             ; Fast interrupt
                ...
```

Several of these will be used in future exercises.

Komodo tip

It's possible to display labels instead of addresses in a Komodo memory window; it's also possible to use labels as part of an *expression*. This can make it much easier to find/display particular memory locations of interest.

---

9 | Stacks: "Why" and "How"

This section should be revision!

9.1 Procedure Calls

We have already mentioned how a procedure can be called using the "BL" (Branch-and-Link) instruction: this is similar to a branch but it leaves a 'return address' in R14, also known as the "Link Register" (LR). The branch can therefore be 'undone' by moving this value back into the PC. This behaviour is illustrated in the following (rather pointless) code fragment.

```
11BC    ...
11C0    ADD     R1, R2, R3       ;
11C4    MOV     R4, #6           ;
11C8    BL      &1234            ; Leaves LR = 11D0
11D0    SUB     R1, R1, R0       ;
11D4    MOV     R4, #-1          ;
11D8    BL      &1234            ; Leaves LR = 11DC
11DC    ADD     R7, R7, R0       ;
11C0    ...
        ...
1230    ...
1234    MOV     R0, R6, LSL #1   ;
1238    MOV     PC, LR           ; Return
123C    ...
```

An **execution trace** of this code would show the instructions executed as follows:

```
11BC    ...
11C0    ADD     R1, R2, R3       ;
11C4    MOV     R4, #6           ;
11C8    BL      &1234            ; Leaves LR = 11D0
1234    MOV     R0, R6, LSL #   ;
1238    MOV     PC, LR           ; Return (to LR)
11D0    SUB     R1, R1, R0       ;
11D4    MOV     R4, #-1          ;
11D8    BL      &1234            ; Leaves LR = 11DC
1234    MOV     R0, R6, LSL #1   ;
1238    MOV     PC, LR           ; Return (to LR)
11DC    ADD     R7, R7, R0       ;
11C0    ...
```

Notice that the procedure (or "**subroutine**") has been 'inserted' twice, at different points in the parent code.

Also notice that a parameter has been passed (in R4) which is different for the two invocations and a result is also returned (in R0). Procedures which return values are often called "**functions**". An example of parameter passing would be a character to a 'print' routine.

9.2 Stacks

A potential problem with the ARM's procedure calling mechanism is that the BL instruction simply overwrites the contents of the LR. This means that the called procedure cannot itself call another procedure (a process known as "**nesting**") without overwriting this value.

Nesting procedures is extremely useful. To enable this the return address must first be copied from LR to somewhere where it will be preserved. The usual place is a memory location; a convenient temporary store is on a **stack**.

A stack is a data structure which is used for temporary variables. It is a "last-in, first-out" structure where (in principle) only the top item is available[1]. A stack is just a data structure and you can build as many as you want. However they are so useful that there is often support in a processor's instruction set architecture (ISA) to help with stack operations.

In principle when an item is stacked all the other elements move one 'slot' further away, and they all move back one 'slot'. In practice moving data about would be hopelessly inefficient; instead the data remain stationary and the address of the top of the stack moves. This address therefore needs to be maintained and it is kept in a **stack pointer**. Typical systems only dedicate a single register to this function and to the same stack will be used for several functions, usually just known as 'the stack'.

Of course before it is used the stack pointer must be set up at the 'start' of a free area of memory. This will typically be done as part of the system initialisation. A programme which is running doesn't care what value the address in its stack pointer is.

Stack set-up

A stack requires an area of memory and a stack pointer (SP). In assembly language a *space* in memory may be reserved using a DEFS (**DEF**ine **S**pace) directive: an argument gives the size in *bytes*. (Note that stack operations will operate on *words* so the size should be a multiple of 4.)

If the directive is *labelled* the label will be the first (lowest address) byte of the reserved space. Note that – by convention – stacks grow *down* in address space; thus SP should be initialised to the 'end' of the space or – since the SP is typically *pre*decremented – the address immediately following the space.

The SP should always be *word aligned*.
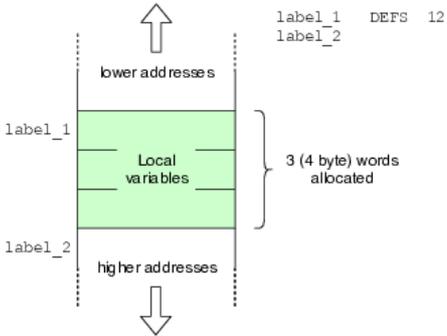
---

[1]Many card games employ such structures.

Recursion

Stacking is particularly convenient because it allows a procedure to call itself. This process — known as "**recursion**" — is sufficiently useful to allow for. (Naturally there must be some condition to terminate this!)

For the 'classic' example of recursion look up "Tower of Hanoi" (sometimes known as the "Tower of Brahma").

---

10 | Exercise 3: Nesting Procedure Calls

10.1 Objectives

- Revise the ARM call/return mechanism
- Set up a process stack

10.2 Stack Operations on an ARM Processor

Because it has so many different addressing modes it is possible for an ARM to build stacks in a number of different ways. It is important to know which model is in use because — unlike many processors — there is no explicit "PUSH" or "POP" operation; instead these are done using load and store operations. However the mnemonics have been made available, as described below.

It is recommended (though not compulsory) that the stack model known as "full descending" is used (figure 10.1). In this model the Stack Pointer (`SP`) contains the address of the last item pushed, and subsequent 'pushes' store data in lower-numbered addresses.
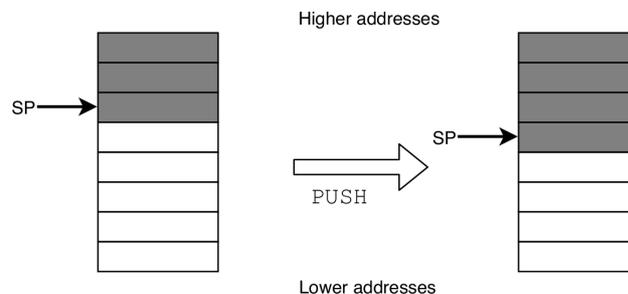
Figure 10.1: Stack push: before and after (Full Descending stack model)

In this mode the operation "`PUSH R0`" becomes:

```
STR     R0, [SP, #-4]!
```

i.e. `SP` is decremented before the data transfer. Note the '`!`' to preserve the modified `SP` value.
Conversely the "`POP R0`" operation would be:

```
LDR     R0, [SP], #4
```

i.e. the load occurs from the current `SP` address which is post-incremented. Note the `SP` value is implicitly written back here too (despite the absence of the '`!`').

In typical ARM code it is quite common to wish to push/pop several registers at the same time. This can be done using the LDM and STM (LoaD Multiple & STore Multiple) instructions.

Push and Pop

This is computer jargon.

- "Push" means 'add a quantity to the stack'

- "Pop" means 'remove a quantity to the stack' (occasionally called "pull")

- The values on the stack do not move; instead the stack pointer changes to indicate the new position of the '**top of stack**'.

```
STMFD    SP!, <registers>
...
LDMFD    SP!, <registers>
```

The list of <registers> is specified something like this: "R0, R3, R6-R9" The implication with a "full descending" stack is that the stack pointer should be initialised to the address immediately above the area reserved for the stack; the first stack push will then predecrement the address into the allocated region. For example if addresses up to 00003FFF are allocated for a stack the (full descending) SP should be initialised to 00004000 so the first word will be pushed to 00003FFC.

The assembler also accepts the synonyms:

```
PUSH     <registers>
POP      <registers>
```

10.3   Procedure Returns in ARM Code

If an ARM procedure is a 'leaf' procedure[1] which calls no other procedures then the return address may be left in the link register.

If an ARM procedure calls one or more other procedures then the return address must be preserved preferable by stacking ("pushing") the return address. A typical instruction to do this would be:

```
STMFD    SP!, {LR}        ; Push return address
```

If this is done the value must later be recovered by unstacking ("popping") the value in the corresponding manner:

```
LDMFD    SP!, {LR}        ; Pop return address
MOV      PC, LR           ; and move to PC
```

This return operation can of course be optimised to a single instruction by loading the return address directly into the programme counter, thus:

```
LDMFD    SP!, {PC}        ; Return
```

Of course STR/LDR can also be used, however STM/LDM are usually preferred because:

- other registers can be saved/restored without needing additional instructions

- it's easier to remember the correct addressing modes

It is likely that a procedure needs some "scratch" registers for its private workspace. To create space it is possible to push the original contents of these registers on entry to the procedure (at the same time as the LR) and pop them back at the end. The STM/LDM instructions are designed to make this easy.

---

[1]i.e. a furthest point in the 'call tree'.

10.4   Practical

Structure your solution to exercise 2 so that it contains two subroutines, one of which prints single characters and the other which prints a string. The string should be defined by a pointer[2] passed into the latter routine and should be output by calling the print character routine repeatedly.

Use these routines to print two (or more) different strings in the same programme.

The routines should be callable by a user who knows nothing about their contents — i.e. they should be utilities. This means that, when called, they should leave all registers unchanged and only "do what they say on the can", i.e. print a character or string.

Don't forget that you must allocate some memory for the stack and you must initialise the stack pointer before using it.

Memory can be allocated using (for example):

```
            DEFW    0, 0, 0, 0      ; Reserve four words
```

or, more practically:

```
            DEFS    100             ; Reserve 100 bytes
```

(Note that this reserves bytes, not words.)

10.5   Advanced

Try one or more of the following:

Print different strings on different lines of the display.

Print "top" or "bottom" as a response to pressing the buttons on the board.

10.6   Submission

You should submit this exercise for assessment and feedback. The submission should include code using a stack to nest procedure calls and bit manipulation to operate the LCD module. A legible, appropriately commented source file will be appreciated and credited appropriately.

---

[2]i.e. an address

EQU vs DEFW

There is sometimes some confusion between the uses of the 'EQU' and 'DEFW' (and allied) directives; these have different purposes.

EQU	DEFW
Syntax: `label equ <expr>`	Syntax: `{label} defw <expr>{,<expr>...}`
Defines a **compile time** symbol with the specified value.	Defines a (several) **word(s) in memory** with the value(s) set by the expression(s).
Label is set to **value** of expression.	**Label** (if present) set to the **address** of (the first) used location.
No presence in object code.	**Word(s) present** in object code.
Used to define **constants** etc.	Used to **reserve space** for and **initialise** variables, look-up tables etc.

Word Alignment

There is sometimes some confusion about which addresses can be used for storing different 'items'. This box describes why some restrictions exist and which apply to the ARM processor used in this lab.
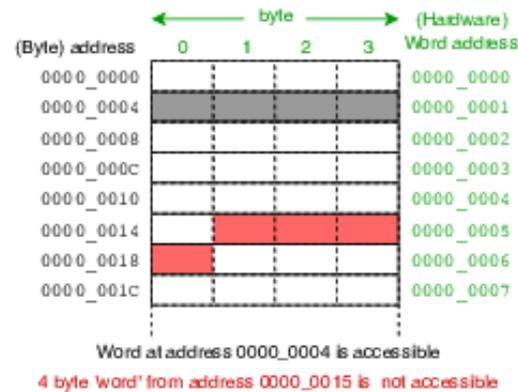
Restrictions

This example uses 32-bit words but the principle applies to any word size.

Computer memory is arranged in words to allow a machine to run quickly: if more bits can be moved from or to memory at the same time then fewer cycles are required. On the other hand, more parallel wires require more costly hardware.

In a (typical) 32-bit machine, memory will be 32 bits wide: each 32-bit word has an address. However, historically, it was convenient (and economic) to be able to use smaller variables (e.g. for characters) and the 8-bit byte became the standard. Architectures since have used word sizes of 8×2^N bits – i.e. {8, 16, 32, 64}; the 2^N multiplier makes sense since multiplying and dividing by these quantities is trivial in binary whereas multiplying and dividing by (say) 3 or 5 is much harder.

Only bytes which are *stored* together can be moved in a single cycle. Thus a word fetched

Word at address 0000_0004 is accessible

4 byte 'word' from address 0000_0015 is not accessible

from memory should comprise bytes which are stored at an appropriate byte address multiple.

Some machines *will* allow the programmer to transfer arbitrary quantities at arbitrary addresses: they do this by using additional hardware to insert extra memory cycles. This is slow and inefficient though and is generally a Bad Idea, even if it is allowed.

The ARM

ARM is a 32-bit processor and (historically) insists on word alignment. Thus:

- **bytes** can be located at any address.

- 32-bit (4 byte) **words** can be located only at addresses which are **multiples of 4**.

  - note that **instructions** are words so this restriction applies.

- 16-bit **halfwords** can be located only at addresses which are multiples of 2, but are not really required in this module.

In the assembly language, using the directive **ALIGN** will move (if necessary) to the next multiple-of-four address.

The addresses used for words are (by convention) the address of the *least significant* contained byte: thus the bottom two *bits* should always be zero.

11 | Code Organisation

11.1   What belongs where?

To some extent, how you organise your code is up to you. However there are certain rules which should be observed and others which will help you (and others) in the long run. This section briefly describes three different places where code can be located and outlines what is appropriate in each place.

Operating system functions are run in a privileged mode. These will usually be invoked via a system call, but could also be invoked by another exception such as an interrupt (q.v.). The operating system deals with the hardware and all direct hardware reads and writes should be contained within. It also provides an **abstraction layer** which is specific to the particular machine at one 'side' but has a generic interface for the user. This allows the same user code to work on many different machines.

Libraries contain commonly used functions which may be called by a number of different programmes. Keeping these functions in a library saves writing them more than once. To be generic, library functions are often quite 'plain' and often require several parameters. They run in user mode but may (and often do) call operating system routines.

User code is the 'real' application programme which contains all the 'custom' code for the particular job. It runs in user mode and calls both libraries and the operating system as necessary.

Library contents

What sort of things should go in a library? This should be user mode code — i.e. no modification of hardware except via the defined system interfaces — which is used frequently. An example here would be printing a string of ASCII characters: printing an individual character to the LCD requires hardware manipulation and is therefore an operating system task; iterating along a string can be done in user mode and, as it is often required, maybe should be a library function.

Printing a number in ASCII is another example. Here you could consider whether another parameter to specify the radix (e.g. decimal, hexadecimal, etc.) would be worthwhile.

For some other examples try typing "man string" (for example) at a Unix prompt. If the manual is installed this will list some of the standard C language libraries. You don't have to interpret all of them here!

11.2   What belongs *when*?

There should also be a clear distinction as what can be done *when*. Some calculations are *dynamic*: they rely on run-time information. For example the particular entry in a jump table is only loaded after you know which entry you want . . . this time.

However, other calculations may be done *statically*, i.e. at compilation time. For example, if you want to know how many entries there are in a table (of, say, 4 byte words) this will be a fixed quantity in the object code and the assembler (or compiler) can work it out for you.

e.g. "`#(table_end - table_start) / 4`"

Building a library

First consider if a function is likely to be used in several different programmes. If it is then it is a good candidate for being placed in a library.

Next think about how that function may be applied in the future. Are your first ideas more generally applicable or would the addition of an extra parameter (even if ignored for now) make it more useful later. Remember, once in use several programmes may rely on the same function so the interface should be fixed.

Define the interface. What goes in and what comes out? What registers (or memory locations) are used for these? Document this!

Put the function in a separate source file, with other, similar functions.

Include the library source file in your programme.

```
            INCLUDE <filename>      ; "GET" is synonymous
```

This will instantiate the code *at that position*. It may therefore be best to 'include' libraries at the end of your main source file. Call the function as normal.

[Note that you are including the library functions as if you had copied them in; therefore labels should all have unique names.]

Although you may sometimes edit the code, you must always reassemble (recompile) it before use, when this number can be calculated; it never changes in between compilations.

Clearly it is more efficient to precalculate values (once, and 'off line') whenever this is possible to avoid performing the same calculation repeatedly at run time. Always consider what might be possible at compile time and try and ensure that it is done then.

Example Programme Header

It may not matter for small programmes, but as projects grow it saves time if you can identify what is in each file and when it was last modified. This is an example of the sort of information which is useful.

```
;------------------------------------------------------
;              Traffic lights programme
;              J. Garside
;              Version 1.0
;              1st January 2003
;
; This programme emulates a set of traffic lights
; using the on-board LEDs.
;
;
; Last modified: 2/1/03 (JDG)
;
; Known bugs: None
;
;------------------------------------------------------
```

As complexity increases it is a good idea to document each procedure etc. in a similar way. This is especially useful when collaborating on projects where, for example, you can check if someone has changed a routine since you last used it.

12 | ARM Operating Modes

The ARM processor has seven operating modes. Of these six are **privileged** and are provided for operating system support whilst the seventh is **user mode** and is used for application programmes.

Up to now everything has probably been run in **supervisor mode** the default start-up mode. This mode is provided for software set up and servicing supervisor calls (`SVC`s), which are the subject of the next exercise. Ideally applications should be run in user mode.

The current mode is defined by the lower five bits of the `CPSR`. To change mode the processor must change these. For security these cannot be changed directly from user mode, so once you're there *there's no way back*. This is, of course, a security feature which prevents user code performing *privileged* operations and accesses.

Abbreviation	Mode	CPSR code (Binary)	CPSR code (Hex)
USR	User	1 0000	10
FIQ	Fast Interrupt	1 0001	11
IRQ	Interrupt	1 0010	12
SVC	Supervisor	1 0011	13
ABT	Abort	1 0111	17
UND	Undefined	1 1011	1B
SYS	System	1 1111	1F

Table 12.1: ARM operating modes

Note: the other bits in the `CPSR` are also significant and should not be changed frivolously!

To change from (say) supervisor to user mode the following sequence may be used:

```
MRS     R0, CPSR        ; Get current CPSR
BIC     R0, R0, #&0F    ; Clear low order bits
MSR     CPSR_c, R0      ; Rewrite CPSR
NOP                     ; Bug fix on some ARMs
```

The last line is a precaution against the change being delayed as it was on some early ARMs. The `NOP` ("No operation") mnemonic translates to a safe "`MOV R0, R0`".

Of the others, only (one of) the two **interrupt** modes are likely to be useful in this laboratory; these are described in more detail later in this manual.

Abort mode is present to support memory faults which are not of interest here.

Undefined mode is entered when an undefined instruction is encountered. It allows the operating system to look at the offending instruction and — maybe — take remedial action. One common example is the software emulation of floating point instructions if the floating point hardware is absent. Of course you may enter this mode inadvertently if your programme crashes and ingests something which isn't a legal instruction.

'CALL' and 'RETurn'

Most CISC (Complex Instruction Set Computer) processors provide specialised instructions for entering and exiting procedures. These normally stack and unstack the return address automatically. There are both advantages and disadvantages to this mechanism.

12.0.1   Multiple stack pointers

Each mode[1] has its own private copies of `R13` (`SP`) and `R14` (`LR`) which can be used to set up private stacks. Before using a mode it is advisable to set up a separate space for each stack and to initialise `R13` appropriately. Initially only the supervisor and user mode stacks will need to be set up, but this will grow later. Allocate 'enough' space for any anticipated stack usage; this will probably be fairly small for the supervisor stack and 'open ended' for the user stack.

As there is no return from the user mode there is another way of setting the user `R13`. **System mode** has the same registers as user mode but is privileged; it is therefore possible to switch into system mode, set the user stack pointer, and switch back to supervisor mode if necessary.

---
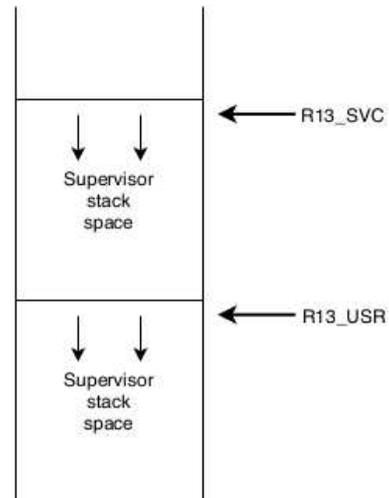
[1]except system mode.

ARM programming puzzle

Determine the magnitude of R0 ...i.e. ABS(R0). No other registers may be changed.

Target: 2 instructions

13 | Aborts

To add security to a computer it is desirable to offer protection to certain parts of the address space. For example, if there are two users of a machine, one should not be able to sabotage the code of the other. In particular, no user should be able to access the I/O space directly as this could cause all sorts of undesirable effects. In a desktop computer the I/O devices will always be protected by an Operating System (OS) — trusted code which the user cannot hack (either accidentally or deliberately) — which will have a device driver for the particular hardware.

At the machine level this means that certain memory accesses, in particular certain accesses attempted in user mode, may be disallowed. The most likely circumstance in which you may encounter an abort in this (ARM) lab. are:

- code crashes and runs 'off the end' of the RAM ⇒ prefetch abort

- attempt to read or write an I/O port while in user mode ⇒ data abort

13.1   What happens if I try and access a protected area?

Firstly the access cannot be allowed so the memory (or I/O) state is not changed. Secondly something 'wrong' is happening so the processor traps (enters an exception) to what is assumed to be safe, operating system code.

In an ARM there are two aborts defined: these are called 'prefetch abort' and 'data abort'.

- A prefetch abort occurs if the processor tries to fetch an instruction from a memory area which is disallowed.

- A data abort occurs if the processor tries to transfer data (i.e. load or store) to or from a memory area which is disallowed.

In both cases the ARM's behaviour is similar: normal execution is suspended, abort mode is entered, the (abort mode) link register is left pointing near the failing operation and execution resumes at an address near the beginning of the address space. This behaviour is similar to other ARM exceptions SVCs, interrupts, etc. which are dealt with later.

13.2   What is available here?

In the lab. system some very crude memory protection has been implemented. This causes the appropriate abort if an access to I/O space is attempted in user mode.

13.3   What might I see?

Look for the processor entering abort mode. You will not normally get into abort mode unless you have caused an abort so this indicates that you have tried to access I/O space from user mode. The `PC` will have jumped to either address 0000000C (prefetch abort) or 00000010 (data abort) which may have caused a crash. If you want to diagnose the fault better you could put something such as "B   ." at these addresses to 'halt' the processor.

LR_{abort} will indicate the position of the faulting instruction. For exact details ask the lab. staff.

13.4   What should I do about it?

Find what went wrong and make sure you only access I/O space from a privileged mode (i.e. use a `SVC` as described in exercise 4).

13.5   Aborts Continued: Some Tougher Questions

13.5.1   How might a 'real' operating system handle an abort?

This may be quite complicated; there is not space to go into all the implications here so see your operating systems course (maybe next year). The simplest option is to abandon a particular programme; in a multi-tasking system this would not halt the machine as it would be able to switch to another job.

An illegal attempt by a user to 'fiddle' with I/O devices will always result in that process being abandoned. However memory protection can also be used to support **virtual memory** systems. In a virtual memory system there may be no memory present at the requested address but it is possible to remap the address space (using more, specialised hardware) to put some there. This is usually known as **paging**. If this is the case then it is possible to correct the 'fault' indicated by the abort and rerun the instruction, this time successfully. This is why the processor must save the address of the faulting operation.

Note that, on an ARM, recovery from a data abort involves a lot of work for the OS: it must identify the failing instruction, decode it (it may be, for example, an LDM which has aborted half way through), undo any side-effects (such as base address modification, which may have happened already) work out where the failing address was and swap in the relevant page. Only then can it go back and try again. It is not for the faint hearted!

13.5.2   Are there other options for memory protection?

Typically, yes. A common function is to allow parts of the memory to be read-only. Instruction space can be protected from modification in this way; this can help prevent a crashed programme from modifying its own behaviour and, possibly, doing further damage. Data is sometimes made read-only so that the OS can detect if it has been modified: the first write causes an abort, the OS notes this, enables writes and reruns the aborting instruction. The advantage is that data known to be unmodified does not need saving when that memory is recycled.

13.5.3   Some questions for you

With the lab. system, imagine you are running a user-mode programme. What could you do to enable the direct manipulation of an I/O port? (Hint: you'll need to get into a privileged mode.)

What changes to the memory protection system are needed to prevent you escaping from user mode?

14 | Exercise 4: System Calls

14.1 Objectives

- Introduce system calls

- Supervisor/user mode operation

- Structure & reuse of calls

14.2 System Initialisation

When a system is switched on a large amount of its state is **undefined**. ROM will hold its contents (of course) but modern RAM relies on a power supply to retain data so it will be in an unknown state (unless battery-backed). More relevantly the processor's registers (including the `PC`) will be undefined which means the processor's behaviour cannot be predicted.

This would clearly be unacceptable! Therefore at power-on (and possibly at other times) a subset of the system state is **reset** to predefined values. It is not normal to attempt to define all the state (for example the RAM contents) as this would be too expensive, but registers such as the `PC` are defined, so that the processor will execute a known piece of code. If more initialisation is required (it usually is) then the software can perform this function.

14.3 ARM Initial State

Following reset the ARM's register state is undefined except for:

- `PC` (a.k.a. `R15`) which is set to `00000000`

- The control bits of the Current Program Status Register (`CPSR`) which is set to:

  - Interrupts FIQ, IRQ disabled
  - Normal ARM instruction set (not "Thumb")
  - Supervisor operating mode

If these terms are unfamiliar, don't worry just yet.

14.3.1   Mode

The ARM has a number of *modes* of operation. These define, in part, the register map and the operating privilege level. ARM has a simple privilege model in that all modes are privileged except the user mode. Privilege grants the ability to do certain things (such as alter the operating mode) which cannot be done from user mode. In a system with memory management only privileged tasks have access to certain areas of memory, such as the operating system workspace and the input/output (I/O) devices. User programmes are run in user mode which means that they cannot directly interfere with the hardware. The other restrictions ensure that they cannot change the interrupt settings or change mode to escape from these restrictions.

In the lab. system there is some limited memory management. This means that when user code wants, for example, to print something out it cannot do it directly. To do this it must enter a privileged mode (typically *Supervisor* mode). But how?

The answer is a **system call**. These are also known, according to taste as traps, restarts, software interrupts, etc.; ARM uses the term supervisor call (**SVC** for short) so we will adopt this name within this course. You may have met these by their former name of software interrupt (`SWI`); you can use this if you prefer, but ARM now recommend 'SVC'. Same thing!

> You should have used some simple **SVC services** — such as printing a character — before (in COMP15111). Note that these were provided by the emulator on your workstation and **are not available** here. **You must now write your own.**

The behaviour of a `SVC` depends on the processor being used, but is always something akin to a procedure call; the difference is that the procedure is entered with full supervisor privilege. To prevent abuse, restrictions on the procedure address mean that the instruction is not general and must jump to a (or one of a small set of) predefined adddress(es).

In an ARM a `SVC` instruction has the following behaviour:

- The current status (`CPSR`) is saved (to the supervisor `SPSR` — see later)

- The mode is switched to supervisor mode

- Normal (IRQ) interrupts are disabled

- ARM mode is entered (if not already in use)

- The address of the following instruction (i.e. the return address) is saved into the link register (`R14`); note this is $R14_{SVC}$

- The `PC` is altered to jump to address `00000008`

Some of the terminology here may still be unfamiliar!

The upshot is that all the necessary status has been saved so that the calling programme can be returned to and the processor is running a well-defined piece of (operating system) code in supervisor mode.

To complete and exit a supervisor call all this must be reversed. The normal "`MOV PC, LR`" does not affect the mode or flags so a special variant must be used; more on this later.

14.3.2   Exception vectors

These 'vectors' are jumped to when the relevant exception occurs. Because each has only a single word it is usual to place a branch instruction here to a space where the real code can be placed. (Note that this is not necessary for FIQ, the exception requiring the fastest response!) The different vectors are given in table 14.1.

Clearly the reset vector must be initialised to point to the start of the code. However it is usual to initialise all the exception vectors 'just in case'.

What behaviour should be observed if an unexpected exception occurs? Ideally there should be an attempt at error recovery, or at least reporting to the operating system. At this point however this would be over sophisticated. Some possible behaviours are:

- Restart the code (i.e. same as Reset)

- Halt

- Return, ignoring the exception

These can be varied for the different vectors of course. If you want to try and return from an exception (such as a spurious interrupt) discuss your solution first; there are some subtleties which are not immediately apparent.

Exception	Mode	Vector (address)	Link made (LR)
Reset	Supervisor	00000000	None
Undefined instruction	Undefined	00000004	R14_und = undef. instr. + 4
SVC	Supervisor	00000008	R14_svc = SVC instr. + 4
Prefetch abort	Abort	0000000C	R14_abt = aborted instr. + 4
Data abort	Abort	00000010	R14_abt = aborted instr. + 8
—	—	00000014	—
IRQ	IRQ	00000018	R14_irq = interrupted instr. + 4
FIQ	FIQ	0000001C	R14_fiq = interrupted instr. + 4

Table 14.1: ARM exception 'vector' table

14.3.3 ARM system initialisation tasks

When the processor is reset its mode and `PC` are initialised but everything else is undefined. It is therefore usual to run some initialisation code before control is passed to a user programme. This can do numerous things but here we are only concerned with a small set of operations

- Initialising exception vectors
  - Various trap routines — such as interrupt handlers — may need installing.
    At present these can all be downloaded with the programme.

- Initialising stack pointers
  - At some time it is likely that the software will require a stack. An area of memory must be allocated for this purpose and the stack pointer must be set to one end of this area.
    In practice it is normal to have a separate stack for user and supervisor code. Furthermore ARM has provision for a separate stack (i.e. a separate stack pointer, `R13`) in (almost) all of its operating modes. Any stack pointer which may be required in future should be set up now.

- Initialising any peripherals required

- Entering user mode
  - Before a 'user' programme is executed user mode should be entered.
    This should be the last thing on the agenda because user mode code cannot switch back into a privileged mode.

14.4   Context and Context Switching

The **context** of a programme is the environment in which the code runs. One component of this environment is the register contents; clearly when switching from programme to programme each needs its own set of values. (Another component is the memory map, but we are not dealing with memory management here.)

In a multi-tasking system it is usual to provide one stack for every process; during **context switching** the stack pointer will be moved from one process' stack to another. Whilst we do not (yet) want to do this, the 'obvious' context switching — between user programmes — is not the only context change which the system will undergo. For example interrupts (coming in a later exercise) will normally have their own context; ARM even provides separate stack pointers for the two classes of interrupt.

More germane here is the fact that the ARM has separate contexts for user programmes and the operating system. Thus there are two stacks which need initialising here (and more coming later). These already have separate stack pointers built into the architecture see figure 1, on page 7.

As each mode has its own, banked, stack pointer — all of which appear as `R13` — it is necessary to change mode to set up the `SP` appropriate to each context. This is done by altering the mode bits in the `CPSR`. Note that altering these bits is a **privileged** operation and cannot be done from user mode. As there is therefore 'no return' from user mode system mode is provided which gives access to the user register map whilst retaining operating system privileges.

Whilst any privileged process can write to the mode bits it is not usual to simply overwrite the entire `CPSR`. As a general rule only the bits which are of interest should be modified; other bits (such as the interrupt disable bits) should be preserved. As the programme will be unaware of their values (in general, if not in this case) they must first be read. The following code fragment illustrates a switch to IRQ mode:

```
MRS     R0, CPSR        ; Read current status
BIC     R0, R0, #&1F    ; Clear mode field
ORR     R0, R0, #&12    ; Append IRQ mode
MSR     CPSR_c, R0      ; Update CPSR
```

'MRS' & 'MSR' are special `MOV` instructions which can access the status registers; in the case of the `MSR` here the action is limited to the control byte ("_c"), although this is really superfluous here. The action of masking ('BIt Clear') and ORing leaves the other fields in the word unaltered.

14.4.1   Peripheral initialisation

In this example there is only one peripheral which needs configuring before the user code can be started, namely the LCD. This should be cleared and the interface signals left in a defined state. This is done by a hardware reset (such as power up) but you may wish to reinitialise the display each time your programme starts.

In general there may be a number of different devices which should be configured at this time. For example if interrupts are to be used (which will generally be the case) all the devices which could generate an interrupt must be initialised before interrupts on the processor can be enabled. More on this later.

14.4.2   Starting the user programme

When initialisation is complete the processor can be switched to user mode and dispatched to an application programme. This simplest method of doing this is to simply change the `CPSR` and jump to the code.

A more 'sophisticated' method is to treat the reset as simply another SVC-type call. This can set the parameters appropriately and 'return' to the user code. This is slightly more complex, but simplifies the scheduler when that is required. Basically this involves inserting the 'return' address into `LR` (`R14`) and the correct status (user mode, etc.) into the `SPSR` and returning as described below. The `SPSR` in the current mode (only) can be read and written in the same way as the `CPSR`, using `MRS`/`MSR`.

```
MOV     R14, #&D0       ; User mode, no ints.
MSR     SPSR, R14       ;
ADR     R14, User_code_start
MOVS    PC, R14         ; 'Return' to user code
```

14.5 SVC Dispatcher

It will soon become apparent that more than one different OS call is required. The ARM has only a single SVC instruction which must be used to perform all its system functions. It is therefore necessary for the called routine to determine which service is required and dispatch to the appropriate handler.

Two methods have been used to differentiate ARM SVC calls:

14.5.1 Parameter passing

A predetermined register (such as R0) can be preloaded with the number of the service required. This sacrifices a user register but is easier to process. This method is now preferred for compatibility with the Thumb instruction set (not discussed here).

- Advantages: Simple to write & decode

- Disadvantages: Extra parameter needed, destroys a register

14.5.2 Numbered SVC

The SVC instruction ("SVC xxxx") has a 24 bit field which is ignored by the processor. This can be used to indicate the type of SVC. To determine the contents of this field it is necessary to read the op. code from the service routine.

```
E F x x x x x x
```

It should be remembered that the return address is in R14 (LR) and this points one instruction (4 bytes) beyond the SVC instruction . . .

```
STR     LR, [SP, #-4]!  ; Push scratch register
LDR     R14, [LR, #-4]  ; Read SVC instruction
BIC     R14, R14, #&FF000000 ; Mask off opcode
```

R14 can then be used to select the required function.[1].

In addition to a SVC number most OS calls will require one or more other parameters. For example, a call to print a character will require the character to be printed. This would normally be passed in another register. The number and type of parameters passed is a function of the particular call made.

- Advantages: Simple to write, compact, needs no extra register

- Disadvantages: SVC number needs to be 'discovered' by the SVC handler

---

[1]Note: once saved R14 (= LR) can be used as a 'scratch' register

14.6   SVC Service Routines

14.6.1   SVC entry

The following assumes that the SVC service number is already in R0. Note that no register contents are changed here. This code is only suitable for a small number of supervisor calls due to the limits on the range of immediate numbers.

```
SVC_entry        CMP     R0, #Max_SVC     ; Check upper limit
                 BHI     SVC_unknown      ;
                 CMP     R0, #0           ;
                 BEQ     SVC_0            ;
                 CMP     R0, #1           ;
                 BEQ     SVC_1            ;
                 ...
```

This method is serviceable for a small number of different SVC calls. However as the number of possibilities grows it is more sensible to despatch through a **jump table** (see page 50).

```
SVC_entry        CMP     R0, #Max_SVC     ; Check upper limit
                 BHI     SVC_unknown      ;
XYZ              ADD     R0, PC, R0, LSL #2  ; Calc. table address
                 LDR     PC, [R0, #0]     ; #0? - see below

Jump_table       DEFW    SVC_0            ;
                 DEFW    SVC_1            ;
                 ...
```

This code fragment exploits some features of the ARM instruction set. The ADD instruction is one of those rare cases where the in-line shifter can be used; in this case it multiplies the SVC number by four to convert it to a word address. This instruction adds this, the **offset** into the table, to the R15 value. Because R15 is the address of the current instruction plus eight, the offset is added to the address of the start of the jump table. This allows the PC to be loaded directly in the following instruction with an additional offset of zero. If the jump table were located elsewhere this offset could be used to correct the value. It would be good practice[2] to calculate this value at assembly time rather than inserting the "0" explicitly. This would be done as "Jump_table - (XYZ + 8)" and would keep the value correct if editing (deliberately or inadvertently) changed the code.

14.6.2   SVC processing

> "Thou shalt not corrupt the user's state."

Like any procedure a SVC service routine will have a function, some input parameters and some output results. These should be clearly defined (this is what the comment field is for, okay?). Whilst it is possible for a service routine to corrupt other state it is generally a bad idea — it stores up trouble for later when the user has forgotten this. The best thing to do is to preserve all the values that the service routine uses except those which explicitly return a value.

This is what the stack is for.

Note in particular that a SVC **which calls a procedure** (BL) must first preserve its link register and a SVC **which calls another** SVC must first preserve its link register *and its SPSR*. These values must also be restored, of course. You will need to use the MRS and MSR instructions.

Ask, if you are uncertain about this!

---

[2]almost essential!

14.6.3 SVC exit

The SVC instruction has changed the CPSR, generously saving a copy in SPSR$_{svc}$. When exiting from the SVC this must be restored. This can be done with the instruction:

```
MOVS        PC, LR
```

The inclusion of the 'S' ('set flags bit') in an operation with the destination as PC is a special case which is used to copy the SPSR back to the CPSR (see also "Interrupt Service Exit" on page 63).

All the different kinds of SVC will have their own service routines. Using the dispatcher above it is possible simply to return at the end of a service routine; however it is worth considering exiting through a short section of return code which is common to all the service routines. This is less efficient (i.e. more instructions needed in jumping into this code) but may provide lower maintenance in the future if, for example, the SVC entry routine is changed.

14.6.4 Terminate programme

A SVC can be used to declare the completion of a user programme. Obviously this should not return. For now it is adequate to have a simple 'loop stop' (i.e. "B .") to 'suspend' execution. In a multi-tasking system, other processes would take control.

14.7 Practical

Modify your "Hello World" programme to run as an 'application' within a primitive operating system (OS). The 'OS' should initialise itself (including the supervisor stack) and anything the application might need (specifically the user stack) before dispatching control to the user code. The user code should communicate with the peripherals using operating system calls (i.e. SVCs).

Calls which may be useful now might include:

- Print character
- Print string
- Terminate programme

Clearly a 'print string' call makes the user programme somewhat trivial (you might like to print more than one message to bulk out the code) however it may be useful in future. If you do provide it note that it is sensible (structured!) for it to call the 'print character' routine to save code duplication.

Later we can, by modifying only the print character routine, redirect the message to a completely different place.

(The programming hints on page 50 may be useful here.)

Komodo tip

As programmes get longer it gets inconvenient to try to follow their progress manually. Instead of entering the address in a memory sub-window you can use a register name; the window will then update to start at this pointer and move if the register value changes.

An expression can be used instead, thus (say) "PC - &C" will show the last few instructions as well and is a convenient way to follow programme execution.

Similarly, R13 (or "SP") may be used to follow the stack pointer, etc.

15 | Programming Hints

15.1 Multi-way Branches

A common construct in programming is the so called "case" or "switch" statement which is a multi-way branch depending on some variable or expression. This is also useful in assembly language, where it is normally implemented as a jump table. An example in ARM code is given below:

```
            CMP     R0, #Table_max          ;
            BHS     Out_of_range            ;
            ADR     R1, Jump_table          ;
            LDR     PC, [R1, R0, LSL #2]     ;

Jump_table  DEFW    Routine_0
            DEFW    Routine_1
            DEFW    Routine_2
            DEFW    Routine_3
            DEFW    ...
```

Note that:

- the offset (in R0) is multiplied by 4 (LSL #2) when indexing the table to allow for the addresses in the table being four bytes long.

- this code jumps to the routines which must know where to jump back to. To change these into procedure calls ("BL") it is necessary to save a 'return address' before loading the PC, e.g.

```
            ADR     LR, Place_to_return_to_afterwards
```

Such a code sequence is useful, for example, in **dispatching** different supervisor calls to the appropriate **service routines**.

15.2 Allocating Variables

Variables can be held in registers or in memory. The act of 'declaring' a variable is necessary to reserve some space in memory to hold that particular value. The easiest way to reserve space in assembly language is simply to 'define' the relevant item (usually a 32-bit word) in the source file using "DEFW". Typically — as the value is initially unknown — the word would be defined to be zero, but this is arbitrary.

```
            LDR     R1, thingy      ; Load variable
            ADD     R1, R1, #1      ; increment
            STR     R1, thingy      ; and store back
            ...
thingy      DEFW    0               ; Arbitrary variable
```

(In practice this uses PC-relative addressing . . . if you care!)

If an explicit pointer to the variable is required the following directives should serve:

```
ADR     R1, thingy      ;'nearby' variables
ADR1    R1, thingy      ; synonymous
ADR2    R1, thingy      ; more distant (2 words)
...                     ; etc.
ADRL    R1, thingy      ;'distant' variables
```

Breakpoints

A breakpoint is a 'marked' instruction which stops execution when it is encountered. Execution is stopped *before* the instruction can change the state of the system. They are used as a debugging aid; for example a breakpoint set just before a suspect piece of code allows the programme to be 'run' to that point and then single stepped. Alternatively a breakpoint could be used to stop execution if the processor has jumped to an address it should not have reached.

The ARM boards support up to 8 breakpoints and each has three possible states:

- deleted (or never set)

- active (can cause a trap if encountered)

- inactive (will not cause a trap but the programmed values are retained)

In addition a global flag must be set before any breakpoints are considered.

The easiest method of setting/clearing a breakpoint is to 'double click' on an address in a memory window. (They can also be controlled from a pop-up window.) Breakpoints are shown in orange in the memory display.

Note that a breakpoint is ineffective if it is the first instruction encountered after a user 'run' command; this allows the user to elect to 'continue' from a breakpoint without it causing interference unless it is encountered again.

Nesting system calls

Occasionally you may wish to nest system calls. This makes sense if (for example) a call to open a file printed out the filename [a]: there, clearly, a system function nests inside another. It is important that all the return information is preserved so that the original context can, eventually, be restored. Usually this is done by pushing it onto the stack.

On an ARM a user call (using `BL`) may need to preserve the link register (`LR`) value before a nested `BL` overwrites it. A `SVC` has its own `LR` so this is not needed for a single supervisor call from user mode, but if `SVC`s are nested the supervisor `LR` must be preserved before the second `SVC` occurs.

However a `SVC` has also saved its caller's mode (et al.) in its `SPSR`. The next `SVC` overwrites so this, too, must be saved before `SVC`s can be nested. If this is omitted the eventual return could be in the wrong mode and that could result in mayhem.

---

[a]It's unlikely that you'd want to do this. It's just an example, okay?

16 | ARM Processor Flags

In the ARM code sequence:

```
CMP     R0, #10
BLT     sandwich
```

the branch depends on the outcome of the comparison. How does the second instruction know what the first has done? The answer lies in a small set of status flags which reside in the CPSR.

There are four basic status flags (see table 16.1) and these inhabit the four most significant bits of the CPSR. In Komodo these can be viewed (and changed) either as part of the CPSR or individually as buttons (underneath).

Name	Abbreviation	CPSR bit	Meaning
Negative (sign)	N	31	Set if the result was negative (two's complement, i.e. bit 31 of result)
Zero	Z	30	Set if the result was zero
Carry	C	29	The carry output of the ALU following an **arithmetic** (ADD, SUB, CMP, . . .) operation; or The last bit to 'fall off' the shifter if a shift is used in a **logical** (AND, ORR, TST, . . .) operation
Overflow	V	28	Set if the result is 'wrong' due to exceeding the representable number range, assuming two's complement signed numbers. e.g. $2147483647 + 1 \Rightarrow$ -2147483648

Table 16.1: ARM flags

These can be tested individually (e.g. BEQ branches if the Z flag is set) or in some predefined logical combinations. You can look up how (e.g.) 'LT' is evaluated if you're sufficiently interested, but it doesn't matter much at the moment.

The primary method of testing the flags is by using conditional evaluation. ARM allows such predication on any instruction; many processors only allow conditional branches. Note that the flags indicate the status as if the calculation is both signed (e.g. 'GT') and unsigned (e.g. 'HI'); you can choose the interpretation afterwards.

A second use of the carry flag is to act as the 33rd bit in a calculation when needed. For example, if adding two 64-bit numbers R1:R0 and R3:R2, the carry allows an 'overflow' from the lower 32 bits to be included in the second half of the calculation.

```
ADDS    R0, R0, R2      ; Must set flags
ADC     R1, R1, R3      ; Add with Carry
```

The carry flag can also be used as the 33rd bit in shift operations.

Unlike most processors — which typically set flags on any evaluation — ARM gives you the option of setting flags on any data operation, by adding an 'S' to the mnemonic. The only exceptions are the operations which have no register result {'CMP', 'CMN', 'TST', 'TEQ'} which would do nothing if they didn't set the flags.

Not all processors use flags to contain this information: some use general purpose registers or specialised comparison operations. However flags in a status register are not uncommon: another example are Intel's ubiquitous IA-32/IA-64.

16.1 ARM Condition Codes

For reference, the ARM's condition codes are tabulated below (table 16.2). (Don't try and learn them all!)

Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal/equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned Higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No Overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1001	LE	Signed less or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never	none

Table 16.2: ARM condition codes

You may observe that these are in complementary pairs (e.g. EQ and NE). This makes implementation easier. [In detail, bit 28 of the opcode need only be XORed with the rest of the comparison so the hardware is kept simple.]

The ALways code is assumed if no code is specified in the assembly language. Because the codes come in pairs, there is also a NeVer code; this wastes $\frac{1}{16}$ of the instruction space. (On more recent ARM devices this code has been redefined as another ALways code with some new instructions, which therefore *cannot* be conditional).

ARM programming puzzle

Multiply the word in `R3` by 7. No other registers may be used.

Note: '`MUL`' won't work because all its operands must already be in registers.

Target: 1 instruction

---

17 | Exercise 5: Counters and Timers

17.1 Objectives

- Introduce the timer as a peripheral device
  - Some more peripheral programming

- Frequency division by software

- Cursor control on LCD

When operating under real-time[1] constraints it is important to be able to measure the passage of time. Time can be measured (and delays imposed) by totalling the time it takes a section of code to execute. This is a very poor way of producing a delay because:

- Execution time may vary, for example due to cache hits or misses
  - this may be very unpredictable

- Interrupts (see later) could insert extra 'invisible' delays into the code

- If in a multi-tasking environment the code could be suspended for an arbitrary time

- The code may be ported to a machine with a different clock frequency

Thus if a real time reference is needed the programme must have access to a separate, fixed-frequency clock which will not be affected by other system activity. This is normally provided by a hardware peripheral known as a 'timer'.

Definitions:

Counter: a hardware circuit which increments (or decrements) according to an external stimulus.

Timer: a counter circuit which counts the pulses of a regular, clock input

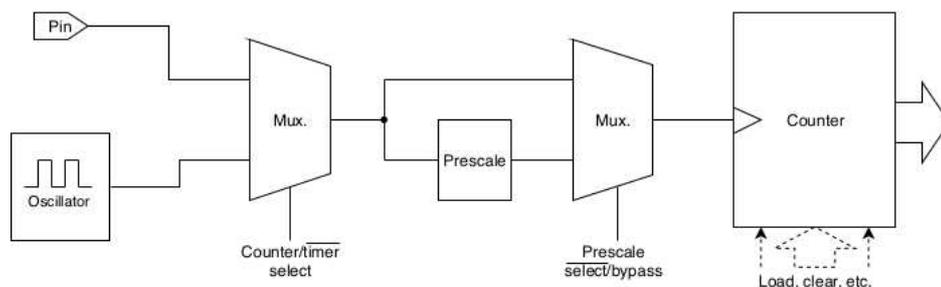**Prescaler:** a divider (modulo N counter) used to reduce the number of counts a timer needs to make

Figure 17.1: Typical counter/timer circuit

---

[1]i.e. the response of the system must meet some real, worst-case constraints.

A single peripheral often functions as both a counter and a timer (figure 17.1). In practice timers are far more common than simple event counters, so the following description applies primarily to them. A timer can be seen as a subsystem which counts a known (but — likely — programmable) number of clock pulses to indicate a defined interval of time. Because clocks in computer systems are often significantly faster than that required a prescaler — typically dividing by a power of two[2] — is often an option to slow the counter down to a 'sensible' rate.

---

[2]A good example is the common 32 kHz 'watch' crystal — its true frequency is 32.786 kHz which, if divided by two 15 times gives a 1 Hz output.


Types of Timer

Because the hardware required is relatively simple, timers are often supported in hardware. There are several possible implementations, some examples of which are:

Free-running: a value is incremented or decremented on each clock pulse. This integer will have a fixed length (typically 8- or 16-bit) and will cycle modulo its word length. Usually the value will be readable by software and thus an interval can be calculated from the difference between the current and previous readings (beware the wrap-around). Note that there will always be some uncertainty due to the clock resolution, but this is not cumulative because the timer runs freely. Sometimes the counter will be writeable by the processor, although this may introduce cumulative errors.

Often free-running timers have one or more comparison registers; these can be set up to a value which the counter will reach in the future and will produce an output (usually an interrupt) at that time. The lab. board has such a system.

One-shot: a slightly more sophisticated timer which counts a preprogrammed number of clock pulses and then signals the fact and stops. Typically the counter will count down and stop at zero. Usually it is able to provide an interrupt. This is useful when some activity is required a known time after an input; for example timing the ignition in a car engine at a known, but probably variable, time in the engine's cycle.

The figure below shows both a free-running and a one-shot timer in 'continuous' use. In each case the timer requests attention and waits for the processor to detect this and service it. Note how the free running timer is able to maintain a regular rhythm whereas the one-shot loses time whilst waiting for the processor.
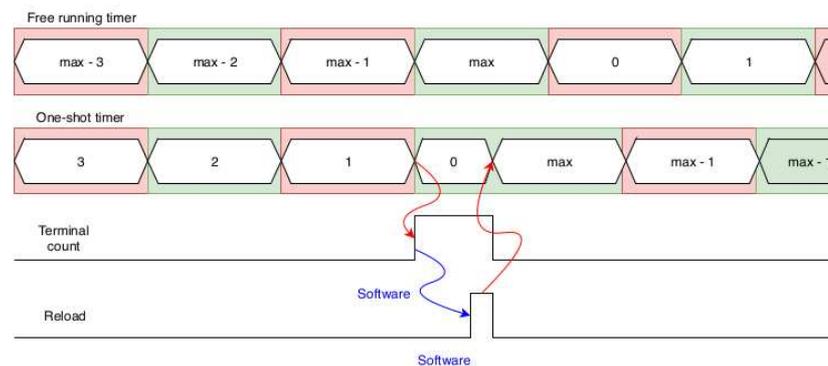
Reloadable: similar in function to a 'one-shot' but they do not stop at zero; instead they load themselves with a value retained in a separate register. This means that they can be programmed as modulo-N counters and then can be left to run freely. (The count can, of course, be changed.) They have an advantage over 'one-shot' timers in applications where time should be measured continuously in that they continue without processor intervention.

As well as providing processor inputs such as regular interrupts reloadable counters are used for programmable clocks within the system. For example one such timer can be used to divide a system clock down to the **baud** (q.v.) clock used on a serial line. This subsequently runs without further (direct) interaction with the processor.

17.2   Practical

Part 1: Write a programme which increments a memory location every 100 ms (one tenth of a second) using the hardware timer as a reference. Make this into a 'stopwatch' such that the lower button on the circuit board starts the count and the upper button causes it to pause. Holding the upper (stop) button down for more than one second should reset the counter to zero.

   Hint: a state diagram might help here.

   The timer provided (`10000008`) has no prescaler, is free-running and increments at 1 kHz; it is 8 bits wide. This frequency therefore needs to be divided down by 1000 to achieve a 1 Hz output.

   Note that an 8-bit counter only has 256 different states.

   Part 2: If time permits, modify your solution to part 1 so the display is a four digit decimal counter running at one increment per second using the LCD as an output. It is suggested that you store the count in BCD (Binary Coded Decimal) to make printing easier — just use a standard hexadecimal print routine. Alternatively a binary-to decimal conversion routine is provided in `/opt/info/courses/COMP22712/Code_examples/bcd_convert.s`

   To provide 'hardware abstraction' — i.e. to divorce the software from the particular hardware used — the timer should only be read using a system call (i.e. a `SVC`). This means that if the programme can be ported to a different system only the `SVC` calls need to be changed.

   As an alternative approach you might consider having a `SVC` which waits for a certain time, passed as a parameter.

   In this exercise the counter can be **polled** until a desired value is reached. This is, of course, inefficient; the following exercise changes the mechanism to using am interrupt.

17.2.1   Submission

You should submit this exercise for assessment and feedback. The submission should include access to the timer using system calls, plus any appropriate techniques from earlier exercises. **Assuming you have done the second part, above, this is the only part you need to submit.**

17.3   Hexadecimal Print Routine

Probably the easiest way to print a number (in, say, R0) in hexadecimal is to first assume a routine to print the least significant digit. This is then called the required number of times with the appropriate value shifted/rotated into the correct position. Try it on paper, first.

```
PrintHex8      Rotate R0 right 4 places
               CALL PrintHex4
               Rotate R0 left 4 places (i.e. restore it)
               CALL PrintHex4
               RETURN

PrintHex4      Save R0
               Mask off everything except lower 4 bits
               IF R0 \textgreater~9 then add 'A' - 10
               ELSE add '0' (ASCII conversion)
               CALL PrintCharacter
               Restore R0 and RETURN
```

Note: the ASCII '6' is not the byte 6, but the byte &36. See p.88 for a complete ASCII table.

Binary Coded Decimal (BCD)

Binary Coded Decimal is a representation where decimal digits are coded into bit fields. Because there are (obviously!) 10 decimal digits each digit uses four bits. Thus, a byte containing binary `0010_0011` would represent 23_{10} rather than $23_{16} \equiv 35_{10}$.

BCD was once widely used to avoid division operations for display purposes.

17.4   Decimal Printing

The difficulty with printing in decimal is that the number to be output must be divided down first. (This is also true in hex, but dividing by 16 is much easier). The ARM does not have a general-purpose "DIV" instruction so you would need to write a function to do this.

Assuming division is available, here are two different ways of printing a 16-bit unsigned number in decimal. Step through these (on paper) to see how they work.

17.4.1   Decimal print — method #1

Start with a table representing the digit positions; for a 16-bit number (0-65535) this would be:

```
DEFW    10000   ; Note these are decimal
DEFW     1000        ; numbers
DEFW     100
DEFW     10
DEFW     1
```

Divide the number to be printed by the first entry in the table, keeping both quotient and remainder. Print the quotient, which must be a single digit. Repeat the process successively using the remainder as input with subsequent lines until the end of the table.

17.4.2   Decimal print — method #2

Divide the number by 10; stack the remainder and repeat until the number reaches zero. This pushes the digits to be printed onto the stack in reverse order. Now pop and print the stacked digits so they appear in the correct order. This method has the advantage that it will work on an arbitrary sized number; it is also easier to suppress leading zeros if that is required. If you try this, make sure that the number 0 prints correctly though!

How to stop the processor from a programme

The ARM's behaviour under Komodo is normally controlled by the 'front panel'. However, occasionally, you may want to a programme to stop the processor itself.

To allow for this a port at address `&10000020` is provided; a write to this port will immediately halt execution; the value written is irrelevant.

---

18 | Calling Conventions

To enable routines — possibly built by different tools or even in different languages — to work together a common interface standard must be employed. This is simply a software convention which programmers or compilers are expected to obey.

For example, you could decide that all processor registers will hold the same values on exit from a 'BL' which they had on entry; this would usually entail saving (probably stacking) and restoring the values so that the routine could use registers internally. There is a cost in time and energy in each operation.

There is no compulsion to use any particular convention in this lab; this is simply mentioned for future interest.

18.1 'Classic' Compiler Calling Convention

Figure 18.1 shows a loose interpretation of the sort of **stack frame** layout a complier might use. (This omits some features such as links to other stack frames which may be included.)

It is assumed here the stack grows 'down'. In this view all the arguments are passed on the stack; see section 18.2 for a slightly different view.

First the arguments are pushed to the stack by the calling routine. The call is then made which might (e.g. x86) automatically stack the return address; alternatively the return address can be pushed as the first act of the called routine (e.g. ARM). The called routine may also push some processor registers to give itself some working space later. Finally the called routine will decrement SP further to open up some space for its **local variables**; these will inherit an undefined value of whatever was on the stack previously[1].

When returning, the process is reversed: SP is incremented, the working registers are restored to their values for the called routine, the actual return (jump) is made and SP is incremented again to get rid of the arguments. That final step can sometimes be done as part of the return – if the language convention and the processor instruction set allow it – or may be done by the caller after the return.
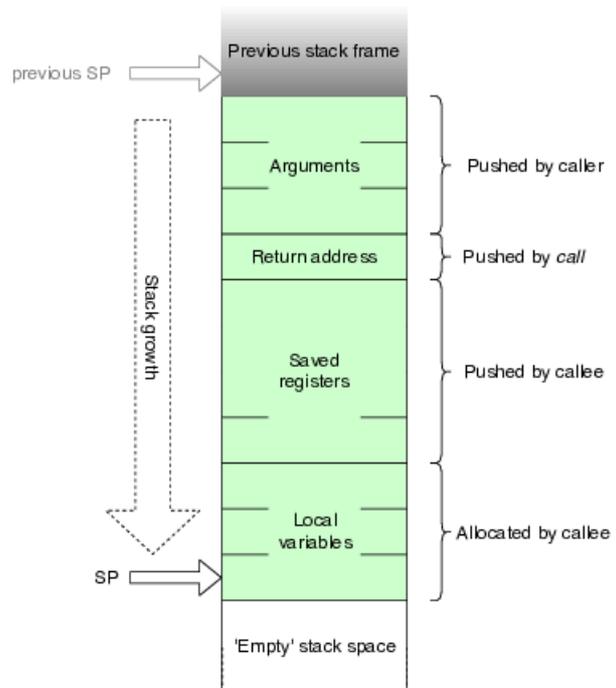
Figure 18.1: Example Stack Layout

---

[1] Try looking some time

18.1.1 Note about C language (etc.)

Some languages insist that procedures are always supplied with the expected number of arguments. C does not do this: think of 'printf(...)' for example!

This forces a convention that the arguments are pushed to the stack in *reverse* order, so that the first listed argument is the last to be pushed. This means that the first argument is at an offset from the stack pointer determined by the *called* routine and can always be found.

It also means that a calling routine can supply as many arguments as it chooses; if it supplies fewer than expected the called routine can find other values on the stack.

A further implication is that the calling routine must be responsible for *removing* the arguments when the call returns, since the called routine can never be sure what was passed. Typically the arguments are not wanted – C does not return modified values – so rather than popping them all (which means expensive memory read operations) a simple add to the SP is used.

```
push    {r0-r2}             ; Three arguments
bl      subroutine          ;
add     sp, sp, #12         ; Three 4-byte values (in this example)
```

18.2 ARM Procedure Call Standard (APCS) — Simplified Version

APCS is still a software specification, used by ARM's own compilers. Because languages such as C typically pass (a 'few') arguments into procedures and functions it takes this into account and reserves some registers for the job. Because the number of arguments is not bounded it cannot necessarily pass them all in registers.

Register(s)	On entry	On exit
R0	First argument	Function return value else undefined
R1-R3	Second to fourth arguments	Undefined
R4-R11	Caller's own variables	Preserved
R12	Scratch register	Undefined
SP	Stack Pointer (Full Descending)	Restored
LR	Set as return address	Undefined
Flags	Not defined	Undefined

Table 18.1: Simple APCS register model

If the procedure uses more than four arguments, the excess are pushed onto the stack before the call ('BL') is made; they must be deallocated after the return.

You can find the complete and up-to-date APCS document (ARM Ref. "ARM IHI 0042") on line if you're sufficiently interested.

ARM programming puzzle

Test if the value in R0 is an exact integer power of 2. Other registers may be used.

Target: 2 instructions (quite difficult!)

19 | Exercise 6: Interrupts

19.1 Objectives

- Introduce interrupts

- Read buttons to change display

- Invention of user interface

19.2 What are Interrupts?

Interrupts are a mechanism by which hardware can 'call' software.

The usual model of computer operation is that the user's algorithm is implemented in software. Typically this is broken down into procedures which are called to perform ever simpler functions. The lowest level of calls — such as executing instructions — can be thought of as being implemented in the hardware.

An interrupt is initiated by a dedicated hardware signal to the processor. The processor monitors this signal and, if it is active, is capable of suspending 'normal' execution and running an **Interrupt Service Routine (ISR)**. Effectively it is as if a procedure call has been inserted *between* two instructions in the normal code.

ARM has two separate, independent interrupt inputs called "Interrupt Request" (IRQ) and "Fast Interrupt Request" (FIQ). These exhibit similar behaviour although they have their own operating modes. FIQ is a higher priority signal, so it will be serviced in preference to IRQ or, indeed, may pre-empt an IRQ already being serviced.

19.3 Why use Interrupts?

Most processors will only run a single **thread** (stream of instructions) at a given time. It is often desirable to try to do two or more tasks at the same time (or at least appear to do so). Often this facility can be added with a very small hardware cost.

To illustrate by example think of an office worker writing a letter, a single task. He would also like to answer incoming telephone calls. One way to do this would be to pick up the telephone after typing each line to see if anyone is waiting; he could check to see if anyone is outside the door at the same time ...

This periodic checking is known as **polling** and is clearly inefficient. Firstly it involves a lot of unnecessary effort; most of the time there will be no one there. Secondly a caller could be ignored if our author gets stuck on a particular sentence. Thirdly the 'write letter' process has to 'know' about all the other possible tasks that could happen in advance.

By adding a bell to the telephone it is possible to allow it to interrupt other tasks. Our heroine can concentrate on her composition without considering callers, but can still know immediately when one wants attention.

19.4 Examples of Interrupt Processes

Waiting. As has already been seen it is quite common for a processor to have to wait for hardware to become ready. This might be for a fixed time or whilst waiting for external hardware to become ready (e.g. printer off line). In any case polling the hardware is a waste of time the processor could spend doing something else.

Clocks. A clock interrupt can provide a regular timing reference whilst other processes are running. More on this below.

Input. An interrupt can allow the machine to register an 'unexpected' input, such as the user pressing a key or moving the mouse.

19.5 ARM Interrupt Behaviour

If the IRQ signal is active — and enabled — the interrupt service routine will be called after the current instruction has completed. The ARM inserts interrupts cleanly between instructions, just as if a procedure call had been inserted into the instruction stream.

The following actions are then performed:

- `R14`$_{irq}$:= address of the next instruction + 4

- `SPSR`$_{irq}$:= `CPSR`

- `CPSR` mode is set to IRQ mode

- IRQ is disabled in the `CPSR` (note a bit is **set** to **disable** the interrupt on ARM processors)

- The Thumb bit is cleared (if set)

- `PC` := `00000018`

(The FIQ response is similar to the IRQ response (for "IRQ" read "FIQ") except that both IRQ and FIQ are disabled and the `PC` is set to `0000001C`.)

Normally address `00000018` will contain a branch to the actual service code because the subsequent address is the entry point for the FIQ ISR[1].

Note: the only user-accessible registers changed are `PC` and `CPSR`, copies of both of which are preserved in special interrupt registers[2]. Note that, unlike a "`BL`" which changes only the `PC`, an interrupt is an exception — like a "`SVC`" — and changes the operating mode too. Both must be restored to exit the ISR correctly.

19.6 Interrupt Enable/Disable

It is occasionally desirable to be able to ignore interrupts. The ARM processor contains two bits which, when set, prevent IRQ and FIQ from having any effect, i.e. they act as **interrupt disable**[3] bits. These are visible in the `CPSR` as bits 7 and 6, respectively. Because they are in the control byte of the `CPSR` the processor can only modify these when in a privileged mode. (Note: the ARM is unusual in having the '1' state as interrupts disabled; most processors have these bits in the opposite sense.)

As the hardware state is undefined at start up these bits are set by a processor reset. No interrupts will be serviced until they are cleared in software.

These bits reside inside the processor. In addition to these it is usual to have separate interrupt enable bits to control the individual interrupt sources which prevent an interrupt request reaching the processor at all (figure 19.1).

On the laboratory board the interrupts and their enables are implemented on the microcontroller although they can be expanded through the FPGAs. There are eight interrupt sources as shown in table 19.1, although some of these are of no interest here. *Note that the interrupt enable states are* <u>*undefined*</u> *following a reset.*
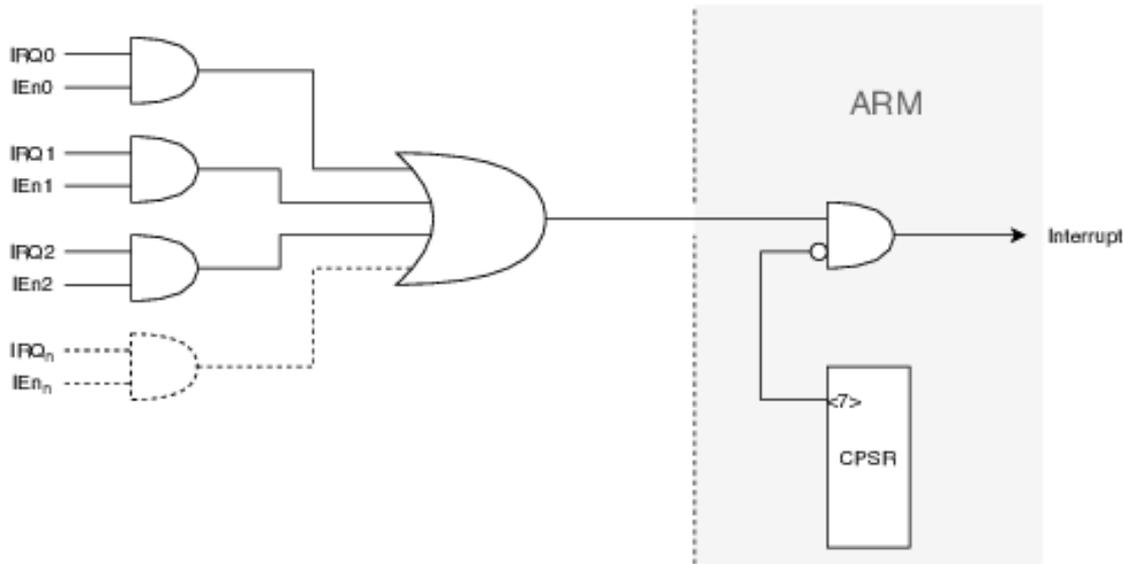
Figure 19.1: Typical Interrupt Configuration

Bit number	Function
0	Timer compare
1	Spartan FPGA
2	Virtex FPGA (Not fitted here)
3	Ethernet interface (Not fitted here)
4	Serial RxD ready
5	Serial TxD available
6	Upper button
7	Lower button

Table 19.1: Interrupt Bit Assignments

The interrupt bits are visible through port `10000018` and the (active high) interrupt enable bits through port `1000001C`. If any interrupt request and the corresponding interrupt enable are active then a processor interrupt (IRQ) will be requested. The processor will service this only if it too has interrupts enabled, i.e. bit 7 of the `CPSR` is 0 (note unusual sense).

19.7   State Preservation

"Thou *really* shalt not corrupt the user's state."

The interrupt entry sequence preserves the minimum of the processor's state necessary. Although the interrupt mode has some of its own, private registers (more in the case or FIQ) it is essential that the service routine does not change any of the state visible to the user process (for example the contents of `R0`).

Usually an interrupt service routine will require some working registers; if this is the case the values in these registers **must** first be saved (the IRQ routine can have its own stack for this purpose) and restored before the user programme is resumed. If this is not done the ISR will still work but the user programme will begin to behave erratically as its variables are randomly corrupted.

Sooner or later interrupt stacks will be required so the `SP` (`R13`) in the IRQ and FIQ modes should be initialised to point to their own reserved areas within the start-up code.

---

[1] The FIQ is the last 'vector' in the table and therefore its ISR can begin at address `0000001C` rather than branching first. This can save time for the most urgent interrupt(s).

[2] In fact the `PC` has been modified, but the original value is recoverable.

[3] This is somewhat unusual: the majority of common microprocessors have interrupt *enable* bits, i.e. in the opposite sense.

19.8   Interrupt Service Routine

During the interrupt service routine it is necessary to acknowledge the interrupt source. If this is not done then the interrupting device will continue to clamour for attention and will cause another interrupt as soon as interrupts are re-enabled — typically immediately on return from the ISR.

Some processor/hardware combinations will acknowledge the interrupt automatically when the ISR is entered; the ARM does not do this, so acknowledgement must be performed by software within the ISR. This involves clearing the bit in port `10000018` which corresponds to the bit being serviced.) Other bits should be left alone until their request is also serviced.)

19.9   Interrupt Service Exit

When the interrupt service is complete the machine state must be restored. Any registers which have been corrupted must be reloaded by the code itself; finally the interrupt entry sequence must be reversed.

Note that the address saved in $R14_{irq}$ is not the address of the next instruction. To return to the correct place the saved `PC` must be decremented by 4. In addition the mode must be restored, interrupts re-enabled etc. This is all achieved with the instruction:

```
SUBS    PC, LR, #4
```

The subtract copies the corrected return address back to the `PC`. The inclusion of the 'S' bit, normally used to set the flags, has a special meaning when the instruction destination is the `PC` and it causes the current mode's `SPSR` to be copied back to the `CPSR`. Because this was saved on interrupt entry it restores the remainder of the processor state. (Remember that the interrupt could have occurred in any mode, not just user mode.) Compare this with "SVC exit" on page 49.

An alternative method of restoring the `CPSR` using "LDM^" can also be used. In this case it is sensible to correct the return address before stacking it.

```
ISR_entry       SUB     LR, LR, #4      ; Correct return addr.
                STMFD   SP!, {R0-R2,LR} ; Save working regs.
                                        ; and return address
                ...
                LDMFD   SP!, {R0-R2,PC}^; Restore & return
```

If the "hat" ('^') is appended to an `LDM` with the `PC` in the register list the current `SPSR` is copied to the `CPSR` during the instruction. (If `PC` is not in the list, or if the instruction is an `STM` the behaviour is different.)

19.10   Practical

Part 1: Experiment with interrupts before running them 'real time' by single-stepping (or 'walking') through an Interrupt Service Routine (ISR). The two user buttons can be used to assert the interrupt signal if they have been enabled. This can then be used to illustrate how an external stimulus can cause a change in a programme's behaviour. Don't forget to enable interrupts on the processor too!

Note that, if running at full speed, a button press may cause several interrupts due to key bounce.

Part 2: Convert your counter (displayed on the LCD) to an interrupt-driven system. When running this should be independent of any user programme which can be run separately.

19.11   Timer Compare

The board includes a timer compare register (visible at address `1000000C`) which can be set to any desired 8-bit value. Each time the counter is incremented the appropriate interrupt bit is modified; therefore when the counter matches the compare register an interrupt may be generated. This will 'naturally' occur once every 256 ms although the compare register can be changed to shorten the interval to the *next* interrupt request.

Note that the interrupt request register can be modified by software to clear the request when the interrupt has been serviced (e.g. during the ISR).

19.12   Frequency Division

The timer interrupts arrive at a maximum frequency of 1 kHz (i.e. 1 ms intervals) and a minimum frequency of about 3.9 Hz (i.e. 256 ms intervals). The counter or clock should only *visibly* increment at 1 s intervals. However there is nothing to say that you can't keep a more accurate clock without displaying the less significant digits!

19.13   Debugging Checklist

Did you remember to:

- set the interrupt 'vector' to the interrupt service routine?

- set up an interrupt stack (if required)?

- enable the required interrupt?

- enable interrupts on the processor (i.e. *clear* the interrupt disable flag)

- preserve all the processor state during the ISR?

- clear the interrupt so it is serviced only once?

- restore the mode (and hence re-enable interrupts) when returning from the ISR?

19.14   Advanced

Reformat the output to make a digital clock. Allow the user to set the time using some input buttons (devise your own way to do this).

Don't forget the buttons may 'bounce', so read through the next exercise first.

Komodo tip

In the same way that a `BL` or `SVC` can be treated as a single step, an interrupt service routine can be made transparent to the user. By enabling the relevant option in Komodo ("Active: IRQ") an interrupt will be serviced invisibly, even if the processor is otherwise halted.

This may be useful in debugging 'foreground' code where interrupts must remain active in real time. For example a clock setting programme may be single-stepped even while the clock continues to be updated.

<div style="border:1px solid black; padding:1em">

Interrupt 'Chains'

Typically a system will have a number of interrupts which are linked into a 'chain'. This occurs (for example) as peripheral **device drivers** are added to a system; each may require its own interrupt.

A common method of dealing with this is to have a code fragment which can grab a recognised interrupt and service it, or pass on control if it is not recognised.

```
        Check my hardware interrupt status
        IF interrupt\_active THEN GOTO service routine
        ELSE GOTO next interrupt possibility
```

Eventually the interrupt should be recognised, serviced and cleared. Cautious programmers may choose to finish with an unrecognised interrupt service routine which will allow for an interrupt input which may have glitched.

Although to date the code has been static this is not the case in a real system. Normally the OS will have initialised the interrupt 'vector' to anything it requires first and subsequent modifications should not disable these functions. To add another interrupt service routine the following operations must be performed:

```
        Install our handler in memory
        Read the current 'vector'
        Make this the 'next possibility' in our handler
        Change the vector to point to our handler
```

An interrupt will then first come to the new handler — where it may be recognised and processed — and subsequently passed to the 'original' handlers.

N.B. Adding (or removing) interrupt handlers is a privileged operation and should be provided as an operating system call.

Note that, on an ARM, the 'vectors' are actually instructions not addresses and thus may require some additional work to synthesize. For example the offset in a branch instruction (the lower 24 bits) is multiplied by 4 (to make a word address), sign extended and added to the PC (address of branch instruction *plus 8*) to find the destination. This process must be reversed to build the instruction.

Branch instructions are not the only (or even most common) instructions which occupy the 'vectors' however. "LDR PC, vector" is quite common. Although this requires another 32-bit word to act as a (true) vector this can be more useful because it is not restricted in its address range like a branch instruction.

Optional *paper* exercise

Design a piece of code which links into an existing interrupt chain when the existing 'vector' could be *either* a branch or LDR PC instruction — and you have to discover which.

(You don't need to produce a perfectly correct implementation of this, just solve the problems.)

</div>

20 | ARM Shifts and Rotates

ARM supports four basic shift types (illustrated below):

Name	Abbr.	Distance	Effect
Logical shift left	LSL	0-31	Move the bit pattern towards the more significant end of the register. Fill at the right with zeros. Lose some bits at the left. The last bit lost can be caught in the carry flag.
Logical shift right	LSR	1-32	Move the bit pattern towards the less significant end of the register. Fill at the left with zeros. Lose some bits at the right.
Arithmetic shift right	ASR	1-32	Move the bit pattern towards the less significant end of the register. Fill at the left with copies of the original sign bit (i.e. bit 31). Lose some bits at the right.
Rotate right	ROR	1-31	Move the bit pattern towards the less significant end of the register. Fill at the left with any bits 'lost' from the right.

Also, a single place right rotation extended through the carry flag ('RRX') is possible. (A similar, 33-bit left rotation can be produced with "ADCS Rn, Rn, Rn".)

ARM is unusual in that its shifter is in series with the ALU; it is thus possible to perform a shift and an ALU operation with a single instruction. If a shift (only) is desired, that instruction can be a 'MOV'. More recently alternative mnemonics are also accepted, such as: LSL   R$_{dest}$, R$_{src}$, #<dist>.

Shifts are useful in isolating and moving bit fields. They are also used in multiplication and division. Multiplying or dividing by 2N is particularly easy (and, hence, fast).

ARM's ability to shift and operate also makes some other operations easy.
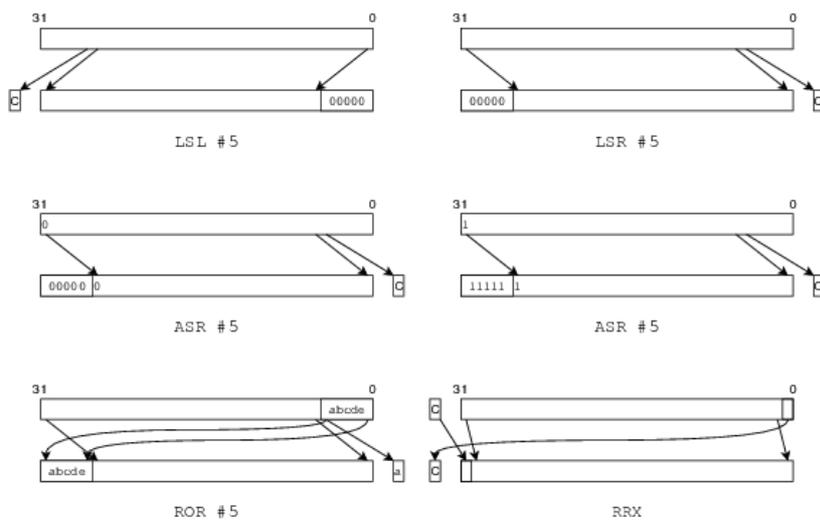
20.1 Shifts and Carries

When a data processing operation is executed the flags may be set. It is typically clear what is meant by 'Zero' (a.k.a. 'equals' when comparing); the sign bit ('N' for Negative in ARM terminology) is also obvious — a copy of the most significant bit.

Overflow ('V') and Carry ('C') have meaning only for *arithmetic* operations (ADD, SUB, CMP etc.) where the values are treated as numbers; they have no meaning in *logical* operations (AND, ORR, MOV etc.) where each bit in the word is processed independently.

In ARM, even when setting the flags, the overflow flag is unchanged on a logical operation.

However many processors — including ARM — use the carry flag as an extension when employing shifts. Here it is the last bit to have 'fallen off' the register in the shift or rotation operation.

Because ARM can shift and apply an ALU operation at the same time there must be a rule to dictate the source of the carry. Assuming a flag-setting operation, the carry flag will be set as follows:

```
if arithmetic operation             //ADD, SUB, CMP ...
  carry is taken from the ALU
else                                 //logical operation)
  if no shift takes place
    carry is unchanged
  else
    carry set to bit 'falling off' shift
```

The ability to 'catch' the carry flag enables some other programming 'tricks' which can lead to more efficient code. Here's an alternative bit test sequence:

```
movs    r0, r0, ror #5  ; Bit 4 into carry
bcs     its_a_one       ; and Branch Carry Set
```

Shift puzzles

Puzzle: what is the effect of this operation?

```
ADD     R0, R0, R0, LSL #2
```

Puzzle: what does this fragment do?

```
            mov     r1, #0          ; Initialise result
loop        movs    r0, r0, lsr #1  ; LSB into carry
            adc     r1, r1, #0      ;
            bne     loop            ; if R0 != 0
            ...
```

Puzzle: or how about this one?

```
            mov     r1, #0
            mov     r2, #32
again       movs    r0, r0, lsr #1
            adc     r1, r1, r1      ; Think about this one!
            subs    r2, r2, #1
            bne     again
            ...
```

<div style="border:1px solid">

Loop Unrolling

Take a piece of code:

```
            MOV     R0, #0          ; Value
            MOV     R1, #8          ; Count
repeat      STR     R0, [R1], #4    ; Zero memory
            SUBS    R1, R1, #1      ; Several times
            BHI     repeat          ;
```

Assuming (falsely) no prefetch this executes in 34 cycles — 8 of which are the actual data store cycles. The 'loop overhead' accounts for 16 cycles.

If a prefetch depth of 2 is assumed (typical for many ARMs) the code's execution time is 48 cycles (only the last iteration doesn't waste cycles refilling the pipeline).

If this is time critical code it can be significantly speeded up by **unrolling** the loop:

```
            MOV     R0, #0          ; Value
            STR     R0, [R1], #4    ;
            STR     R0, [R1], #4    ;
            STR     R0, [R1], #4    ;
            STR     R0, [R1], #4    ;
            STR     R0, [R1], #4    ;
            STR     R0, [R1], #4    ;
            STR     R0, [R1], #4    ;
            STR     R0, [R1], #4    ;
```

The cost is now 17 cycles, a 3× speed-up! The penalty is that the code has grown to about twice its original length, and would be much longer for more iterations. The speed-up is also less spectacular if the 'body' of the loop is longer. This can be exploited by partially unrolling the loop; e.g. in the above example looping four times and storing two words each time costs only one more instruction but only takes 32 cycles (on this model) — a 1.5× speed-up for a trivial cost.

Loop unrolling (or "in-lining") is normally undesirable, but occasionally useful for small loops in time-critical code.

Note that the number of iterations can still be varied ... by calculating and jumping to the appropriate entry point in the code.

** BEWARE **

Counting cycles in this manner is a simplistic approach to system timing. Timing can be heavily influenced by other factors, especially memory speed.

For example, if the code is cached, unrolling a loop will use more cache space which may cause more cache misses. As a cache miss could easily cost (say) 100 instruction times, this 'speed' technique could make the system much slower instead.

</div>

21 | Exercise 7: Key Debouncing and Keyboard Scanning

21.1 Objectives

- Programming around 'real world' problems

- I/O matrix techniques

21.2 Key Bounce

Key bounce is a mechanical effect which causes switch contacts to open and close repeatedly as the switch's state is changed from open to closed, or vice versa. The number of bounces and the time taken for the bounce to die out are not deterministic, but the total time of bouncing is bounded. The maximum bounce period depends on the switch being used and the amount of wear it has had, but a typical period would be about 5 ms. This is a short time in human terms, but a long time for a computer. It is therefore generally not possible to simply sample a switch's state and note every change from open to closed as a button press. The switch must be debounced.

In other labs. key debouncing has been performed in hardware. This works, but increases the cost of the interface. Software debouncing is a cheaper solution.

21.3 Debouncing

Debouncing typically involves waiting for the relevant time; for example it could be noted that a button which was formerly open is now closed and a delay could be imposed before that button is sampled again. This is a fairly simple mechanism to conceive, but requires a time stamp for each button to say when it can/cannot be examined.

Another method of debouncing in software involves the repeated sampling of the key — at decent intervals — but only considering it to have changed state when it has been in a new state for a certain number of consecutive samples. For example a key state could be read as a bit every 1 ms which is shifted into a byte to keep a rolling sample of the last 8 ms of state. The key is 'pressed' when the byte is FF and released when it is 00. For any other state the key is regarded to not have changed.

Another variant of this method is to keep a counter for each key, incrementing it if the key is pressed and decrementing it if it is not. The counter can 'saturate' at 00 and a maximum pre-chosen to indicate when the bounce period has finished. Other variations are also possible.

With any of these methods it is also necessary to note the state of the key before the operation begins. This value changes only when debounce is complete; in the first debouncing method it would be set when the byte reached FF and would remain set until the byte had returned to 00. This provided the necessary hysteresis. When the key is first pressed an event may be generated to cause some effect (such as printing a character).
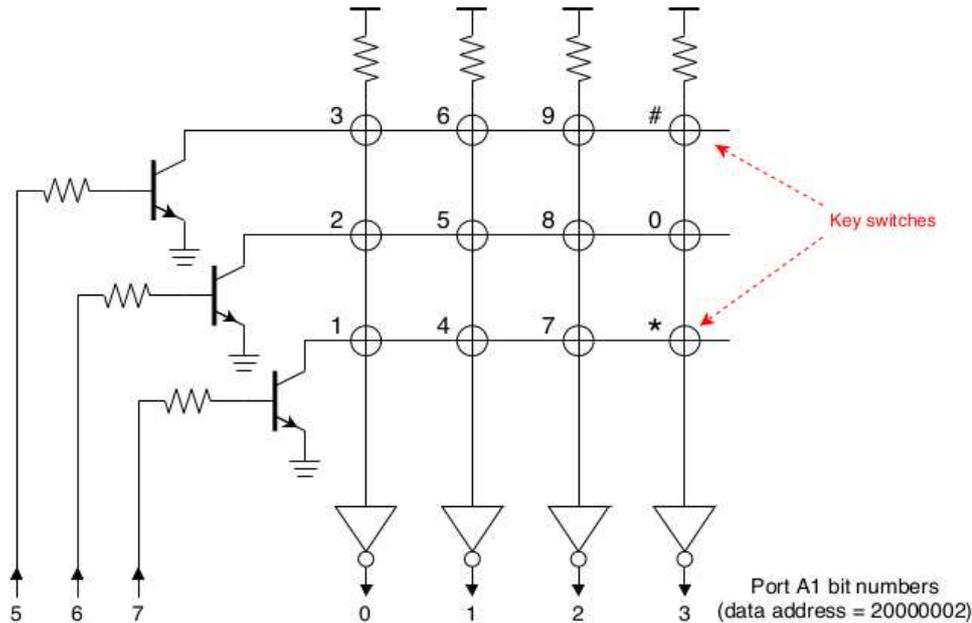
Figure 21.1: Keyboard Matrix

Can you devise a more efficient method to tell if the key has just been pressed? Discuss possibility with the lab. staff (and others) and see if you can find an elegant solution.

The process outlined above needs to be instantiated for every key. The result is a 'map' of the debounced key state which is kept in memory (often sacrificing one byte per key for convenience). This is useful for certain purposes — such as games where key-pressed-or-not is all that is required — but to type into the machine it is also necessary to detect the 'edge' of a key being depressed for most keys[1].

The user interface to the keyboard normally comes down to a "get character" call which waits for the next key press and returns the appropriate code. This is a system call.

In a very simple system (such as here) it may be possible to begin scanning the keyboard on demand. Thus, when input is required, a number of scans (sufficiently separated in time) can be performed and repeated until a key code is returned.

21.4   Keyboards

A simple keyboard would have each switch connected to its own input port bit. However for a typical computer keyboard (which has over one hundred keys) this is too expensive in both silicon and wiring.

Instead it is normal to place keys on a two dimensional matrix; 100 keys can be placed on a 10×10 grid and serviced with 20 I/O lines instead of 100, a considerable saving! Figure 21.1 shows the principle of this on a smaller scale. Note: all the inputs and outputs to this circuit are active high.

To scan a row of the keyboard the corresponding input is activated and all others are inactivated. This means that exactly one of the 'horizontal' wires is driven low.

If no keys are pressed then all the 'vertical' wires are pulled high by the resistors. However if one or more keys on the selected row is pressed the corresponding output will be activated via the switch. This provides the state of some of the keys.

A complete keyboard scan is performed by activating each of the inputs in turn (all others are set inactive) and latching the output pattern for each row.

Note that this scan is done electrically and is not affected by the mechanical bouncing of the switch contacts. It is perfectly possible to sample a particular key on successive scans and debounce it appropriately, as described in the preceding section.

---

[1]Exceptions are keys such as Shift, Ctrl, Alt, . . . which act in concert with other keys.

21.5 FPGA Peripherals

The Spartan-3 FPGA is used to provide the majority of the I/O in this lab. The FPGA can be loaded with a user-selected configuration so that the I/O can be customised for particular applications. However this is not yet necessary as there is a default configuration loaded on start up.

The default configuration has eight, bit programmable 8-bit PIOs which are connected to the central eight connections of each row of the I/O connectors at the front of the board. The keyboard interface should be plugged into the left, front connector which corresponds to PIO #S0.

The FPGA is mapped into an 8-bit wide area of memory occupying the address space 2xxxxxxx. Only byte transfers are sensibly defined. Seven address lines — A[6:0] — are provided and all bytes are significant, so the address space repeats every 32 bytes.

The default PIO address map is:

Offset	Register	Offset	Register	Connector	Port
0	Data	1	Control	S0	Front, left outer, lower
2	Data	3	Control	S0	Front, left outer, upper
4	Data	5	Control	S1	Front, left centre, lower
6	Data	7	Control	S1	Front, left centre, upper
8	Data	9	Control	S2	Front, right centre, lower
A	Data	B	Control	S2	Front, right centre, upper
C	Data	D	Control	S3	Front, right outer, lower
E	Data	F	Control	S3	Front, right outer, upper

Each PIO is programmed via two registers (fig. 21.2): one holds data which should be output, the other defines the direction of the pin. A bit value of "0" in the direction latch programmes the corresponding bit in that PIO to be an output; a value of "1" programmes the bit in that PIO to be an input. The direction latch resets (at hardware reset) to a value "FF" so that all the pins default to being inputs. This register can be read back.

Why is defaulting to inputs (at reset) the sensible behaviour for the PIO bit?

The data register contents is output on any pins defined as outputs. Bits programmed as inputs are ignored. When read this register returns the values at the pins themselves, so input bits should give external input states whilst output bits should be as programmed.

Figure 21.2: PIO Bit Logic

21.6   Practical

Attach one of the small, matrix keyboards to the lab. board and use it to input digits which appear on the LCD display.

21.6.1   Advanced

Modify your system to make a simple adding machine, totalling and displaying the entered numbers.

21.6.2   Submission

You should submit this exercise for assessment and feedback. The submission should use interrupts to scan the keyboard on a regular basis and not rely on software delays.

21.7   Key Translation

Note that the 'code' you read by scanning the keyboard is related to the physical position of the key and is not the number written on it. This code needs to be translated to a corresponding ASCII code for printing. For a hint on an efficient way to do this, re-read "Look-up tables" on page 23.

21.8   'Real' Keyboard Scanning

In a 'real' system it is normal to scan the keyboard using regular (timer) interrupts (q.v.) to decouple the keyboard from the application programme. During each scan the key states would be read and fed to the debounce software. This would then update the keyboard map as required. This allows characters to be typed even if an application is not 'paying attention', a function known as '**type ahead**'.

If a key has just been pressed (and debounced) the corresponding code needs to be passed to the relevant output stream. This is done by inserting the character into a software First-In, First-Out (FIFO) buffer. In a typical keyboard driver the last key pressed and the time it was observed would also be recorded to allow **auto-repeat**, i.e. the last key sent can be sampled again and inserted again if it is still pressed (i.e. held down) at a certain future time.

Another function usually provided by the keyboard driver is **rollover**. This feature updates the 'last key pressed' when a new key is depressed, even if the first key has not yet been released. This feature allows faster typing for those sufficiently proficient.

> Try it on your workstation! Press and hold a key and then press another, the output character should change. If the second key is released first the first key will not be seen until it too has been released and pressed again.

With keyboards scanned under interrupts the system call to read a key reduces to getting the next character from the FIFO (or waiting if the FIFO is empty) and returning to the application.

Multistep bug?

When interrupts are used 'multistep' sometimes does not count all the steps asked for. This is because a requested step may become an interrupt entry action which is not a real instruction.

Is this a bug? What do you think?

Note that the difference in the number of steps requested and instructions executed gives the number of interrupts taken during the sequence. This may be useful in itself.

Software FIFOs

A FIFO (First-In, First-Out) buffer, or "queue" is an 'elastic' pipe, capable of storing items and releasing them in the same order as they were entered.

In software FIFOs are normally implemented as circular (or 'cyclic') buffers, where the data is kept in an area of memory which is logically wrapped around so that walking off one end brings you back in at the other. The head and tail of the queue is indicated by two pointers which obey these rules. This form of buffering is efficient in that the data never have to be moved in memory; if the head and tail were fixed the data would constantly be shuffling down the memory — a very time consuming process.

In operation items (such as key codes) are inserted at the head and the head pointer is advanced (wrapping if necessary). Items (if any is available) are removed from the tail which advances in a similar fashion in the same direction. The buffer management software must keep track of pointers so that the tail cannot overtake the head if the buffer is empty. Furthermore the head must not 'lap' the tail because then items would be overwritten before they have been read.
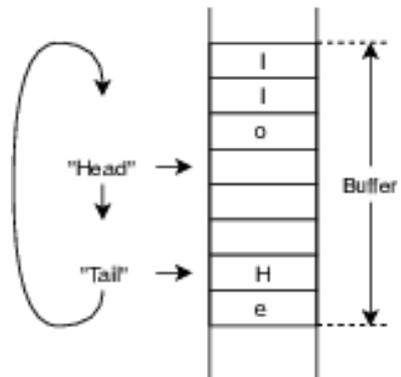
22 | Relocatability

It is usually an advantage if code can be executed wherever it is located in memory, i.e. it is **relocatable** or **position independent**. Depending on the processor used this is not always feasible; for example a processor may only have jump instructions which transfer control to a fixed address. However most modern processors are fairly unrestricted and, with care, code can be fully relocatable. (Note that this does not imply that code can be moved during execution.)

For example the ARM branch instructions are all **relative branches**[1], i.e. the target address is calculated as an **offset** from the current position. Thus whatever address the code is placed at the branch will still be to the correct target. In fact a branch instruction could be written as:

```
ADD     PC, PC, #offset
```

although the offset range is larger than that allowed in 'normal' immediate operands and is sign extended to 32 bits. 'Normal' branches on an ARM are therefore relocatable.

Other forms of branches are usually position independent too; for example a subroutine return will go back to the calling routine — wherever that was. The exceptions are when an address is calculated or loaded from memory where extra care is needed if position independence is to be maintained.

Example: the branch instruction only provides a 24-bit (word aligned) offset, allowing branches backwards or forwards in the address space of about 32 Mbytes (8 Minstructions)). This is enough for almost all purposes, but if a more distant branch is required one way of providing this would be:

```
            LDR     PC, long_branch ; Load target absolute
long_branch DEFW    target
            ...

target      ...
```

This creates a word 'variable' (really a constant) which contains the address of the distant routine. The value can be read in a position independent way — "`long_branch`" will be translated by the assembler into "`[R15, #-4]`" which refers to the next location because R15 is the current position + 8. **However** the value "`target`" is an absolute number which is set by the assembler and thus will not alter if the code is moved.

The following sequence gets around this, at a price:

```
              LDR     R0, long_branch ; Load target offset
branch_instr  ADD     PC, PC, R0      ;
long_branch   DEFW    (target - (branch_instr + 8))
              ...

target        ...
```

---

[1]Although not universally observed the usual convention is that "branch" refers to a flow control change made relative to the current position while "jump" refers to one to an absolute address.

Here the *relative* offset between the 'branch instruction' (ADD) and the target has been stored and added to the PC explicitly; the "+ 8" is needed to allow for the 'PC+8' effect. This is relocatable although slightly slower and requiring an extra register.

As well as code, data should be position independent. Most data are stored in one of four places:

- On the stack (dynamic)

- On the heap (dynamic)

- Within the code (static, often read only such as constants, strings, data tables, . . .)

- Another static space (e.g. global variables)

Dynamically allocated variables are not a worry here; by definition their addresses are defined at **run time**, usually in some space allocated by the operating system.

Data may be embedded in the code. These can be accessed via **PC relative** addressing as in the previous example, so they need not present a problem. (Note however that there is a 4 Kbyte limit on the offset from the PC to the variable of interest which can encourage the slightly dubious practice of interleaving data items between procedures[2].)

It may be feasible to use this technique for variables as well as constants but this practice is deprecated as it leads to problems in placing the programme in ROM.

To access to such constants (or variables) the programmer can write:

```
            LDR       R0, My_constant
            ADD       ...
            SUB       ...
            ADD       ...
            MOV       PC, LR

My_constant     DEFW      &12345678
```

Which generates the code:

```
            LDR         R0, [PC, #&C]
            ADD         ...
            SUB         ...
            ADD         ...
            MOV         PC, LR
```

The number "&C" when added to the value 'PC + 8' at the point the instruction is executed will give the value "My_constant". (Also see the boxed text on page 21.)

If a **pointer** to a label (such as a pointer to a string) is required instead then the directive "ADR" can be used. This is not an instruction itself, but will assemble to something which will generate the correct address. For example:

```
Myself          ADR     R3, Myself
```

Could assemble to:

```
            SUB     R3, PC, #8
```

---

[2]For example, finely interleaving code and data can lead to inefficient cache usage if code and data are cached separately, as they are on many high-performance processors.

Where `R3` is given the value (PC+8) - 8.

If the range of this operation is insufficient the directive 'ADRL' allows a sequence of instructions to be substituted; the length of the sequence, in instructions, can be specified using `ADR?` where '?' = {1, 2, 3, 4}.

The ARM assembler also allows the position of the current instruction to be determined by the value ".". Thus, for example, a 'loop stop' — an instruction which branches back to itself — can be written as:

```
        B       .                       ; Stop here!
```

The hardest problem is finding a place in RAM for the static variables which can be accessed globally rather than — for example — relative to the stack pointer. There is no easy answer to this. The commonest solution is probably to dedicate a register to point to a fixed area of memory (allocated by the OS when the programme starts) and use fixed offsets from that. Sadly this 'loses' a register.

22.1   Optional Exercise

This code sequence for a jump table was given on page 50.

```
            CMP     R0, #Table_max          ;
            BHS     Out_of_range            ;
            ADR     R1, Jump_table          ;
            LDR     PC, [R1, R0, LSL #2]    ;

Jump_table  DEFW    Routine_0
            DEFW    Routine_1
            DEFW    Routine_2
            DEFW    Routine_3
            DEFW    ...
```

This code is not relocatable; the addresses in the table are the absolute addresses of the various routines and so the jump targets will not move with the rest of the programme.

Write some ARM code which dispatches from a relocatable jump table.

ARM programming puzzle

Given a word in R0, swap the byte order (i.e. endian swap).

E.g. change R = &12345678 to R0 = 78563412

Other registers may be used.

Target: 4 instructions (Very hard!)

You can find a solution in the Internet.

---

23 | Exercise 8: System Design and Squeaky Noises

23.1   Objectives

- Configure and download hardware: build a system crossing the hardware/software 'divide'.

- Introduce system-level buses and address decoding

- Introduce the piezo-electric buzzer

All the exercises up to this point have used pre-prepared hardware. This exercise requires additional circuitry to be designed and integrated with an existing system.

Computer Systems

The "three box" model of a computer (fig. 23.1) divides a system into a Central Processor Unit (**CPU**), **Memory** and Input/Output (**I/O**), all connected by a **bus**. In this case the bus comprises address, data and control signals which may be bundled separately, thus inside this general bus resides an address bus and a data bus — the other signals are sometimes referred to as a control bus, but form a less coherent set and we will treat them individually.
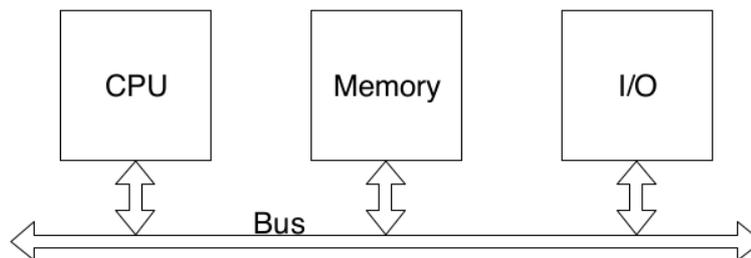

Figure 23.1: The "Three Box" Computer Model

In addition to these three large "boxes" there is a small amount of logic used in their interconnection, often known as the **glue logic**. This performs mundane functions such as distributing the system clock and decoding addresses.

To produce a complete System-on-Chip (SoC) all these boxes must be integrated into a single design. However within this laboratory the CPU and memory "boxes" are already established and only the I/O subsystem needs development. Much of the I/O is mapped into the Xilinx 'Spartan' FPGA and is therefore 'soft', although a simple configuration is pre-loaded when the board is (hardware) reset.

Internally an I/O subsystem comprises a number of devices, together with some 'glue' to hold them together[1]. A typical I/O subsystem is shown in figure 23.2. Here a particular area of the address space has been decoded externally (within the system 'glue') and is used to enable one of the various devices. Which device is selected depends on the particular address which is decoded within the internal 'glue'. Note that it is not necessary to fill the entire address space.

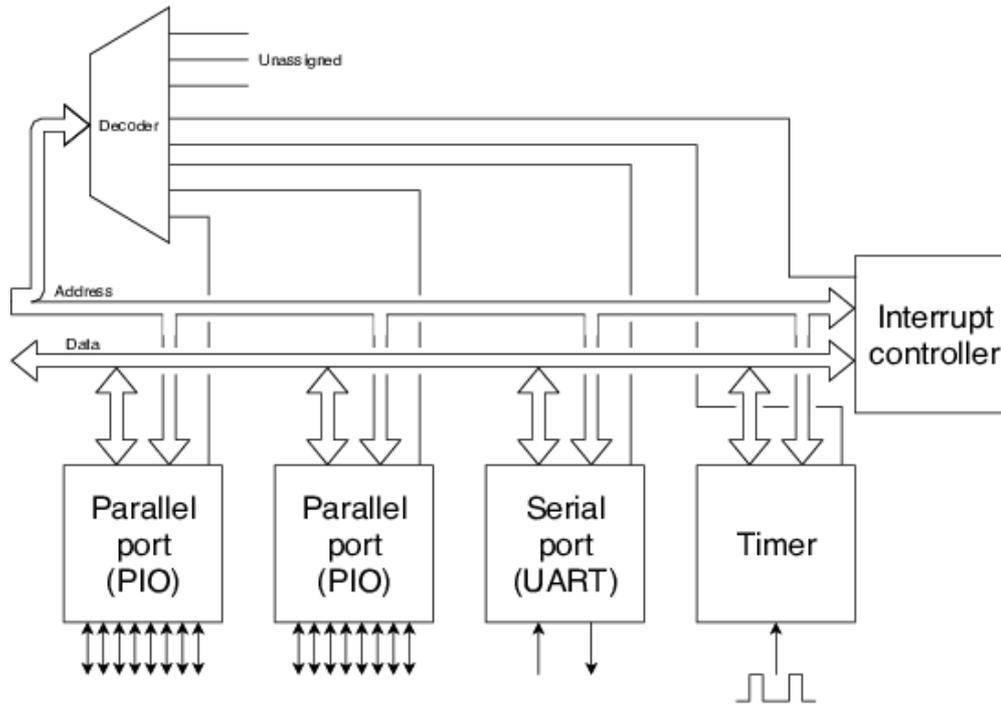[1]The memory subsystem is very similar internally.

Figure 23.2: Typical Computer I/O Subsystem

A customised I/O system can be produced by first creating (or choosing from an existing library) the peripherals which are required. Ideally these should have a common interface, such as a tristate data bus with the same timing characteristics, to minimise the glue logic complexity. A typical peripheral device will have several registers and so will require more than one address in the available space; this corresponds to one or more address lines being required. It is usual to choose the least significant address lines for this.

The remaining address signals, together with the qualifying 'chip select' can then be decoded to produce the individual device select signals. Note that an unselected device should be completely passive, neither driving nor listening to data transactions!

An example configuration (loaded by default) is available in the `ENGLAB_22712` library as `PIO_8`.

23.2   The Spartan I/O Subsystem

The Spartan FPGA has a 8-bit interface to the ARM processor: the data bus is 8-bits wide and this byte used for transfers at all addresses in the available range. Only `LDRB`/`STRB` instructions should be used within this region; any address can be used.

The bus interface signals are given in table 23.1 and are defined in a library component "`Bus_8`". The device is selected by any processor address in the range `20000000-2FFFFFFF` although only a (repeating) set of 128 bytes can be addressed as only seven address signals are available.

Signal	Function	Direction	Notes
D(7:0)	Data bus	Bidirectional	
A(6:0)	Address bus	Input	
CS	Chip select	Input	Active (high) when processor address is 2xxx xxxx
$\overline{\text{Wr}}$	Write enable	Input	Active low
$\overline{\text{Rd}}$	Read Enable	Input	Active low

Table 23.1: ARM/Spartan Bus Interface Signals

When CS and $\overline{\text{Wr}}$ are active together the data bus value should be copied into the addressed location (if such exists). When CS and Rd are active together the addressed location (if such exists) should drive the data bus and return a value to the processor. Typically the chosen addresses for the ports would be 20000000, 20000001, ... 2000001F. *It is a really good plan* to equate (EQU) these addresses to a meaningful label in a header file and use only the label within the programme body. This confers the following advantages:

- References to the ports become more obviously understandable

- Only a single block of references needs updating if the system is reconfigured

23.3   Piezo-electric Buzzers

Piezo-electric crystals are materials which change shape when an electric field is applied[2]. As switching a voltage on an output is a relatively easy thing to do they are widely used by, for example, mobile telephone manufacturers to make irritating squeaky noises. This can be done most simply by grounding one terminal and switching an output bit from 0 to 1 and back to 0 again at the desired frequency. By changing the switching frequency different tones can be played.

A louder noise can be produced by increasing the applied field. With purely digital electronics this can be difficult, but a doubling in amplitude can be achieved by driving both terminals of the device in antiphase[3]. A really sophisticated driver could use voltage steps to output something smoother than a square wave, although this requires considerably more effort.

Note that piezo-electric buzzers are not very responsive at low frequencies so sound 'best' at a few kilohertz. (They also have particular resonant harmonics, which you may hear.)

23.4   Cadence Revisited

Because the laboratory boards have all their peripherals in an FPGA it is possible for the user to define and customise these. Previous exercises have used precompiled peripherals; in this exercise you add your own.

Create a Cadence project for this laboratory using the usual setup script (i.e. "`mk_cadence COMP22712`") and invoke it as appropriate for this laboratory (i.e. "`start_cadence COMP22712`"). The files created will be placed in their own hierarchy: $\sim$/`Cadence/COMP22712/...`

23.4.1   ENGLAB_22712

A component library, '`ENGLAB_22712`' is available which defines a number of pre-existing components. Amongst these is a standard, simple `PIO` (Parallel Input/Output device) which allows the direction and value (if an output) of each bit of an 8-bit byte to be defined. The PIOs are interfaced to an extension of the processor's bus inside the FPGA and each instantiation connects to a set of **I/O pins**. Up to eight PIOs can be employed as there are 64 uncommitted I/O lines. The I/O pin symbols determine the physical connection of the I/O signal.

Some other pin definition symbols are available which overlap with the PIO I/O pins. These may be useful for interfacing to various other devices; for example the buzzer output pins overlap with **GppSA0** (used in **PortSA0**). Note that each pin can only be used once in a given design; if it is duplicated the design will not synthesize.

In addition to the external connections there are some other I/O definitions. These are the signals used on the board such as **clock** inputs and **interrupt** outputs.

A **bus interface** ('`Bus_8`') is provided which allows the processor's bus to be used inside the FPGA. A tristate, bidirectional data bus is retained as a default.

The output pins connected to the piezo-electric buzzer pin are defined by the component `Buzzer`.

---

[2]They also produce an electric field if they change shape, hence their use in spark igniters, for example.
[3]i.e. at the same time, but in opposite directions.

Komodo tip

Komodo not only runs your programme but also keeps the display updated. This second process will interfere with the smooth running of the 'real' programme. If you try to time your output tones in software you will probably be able to hear this.

Although this 'display update' is enabled/disabled automatically when a programme is run/stopped it is also possible to control this independently using the '**Refresh**' button on the front panel.

23.5   Practical

Either modify the existing I/O controller (hardware) to include an output to drive the piezo buzzer or make your own driver from scratch (perhaps using `PIO_8` as a template. Make this play a regular tone, or — preferably — a tune. The simplest way to do this is to add an addressable output port (i.e. a latch) and drive the relevant outputs in software, using a system timer. However, at the available frequencies the resolution (1 ms) is poor.

A more capable, if slightly harder (i.e. involving more hardware), method is to use a programmable clock divider which can be set at different frequencies by the CPU and will then generate the tone on its own.

Such an element ("`Divider`") is provided in the ENGLAB library. This counter divides its input clock by twice (the input value plus 1). Thus if programmed with the value 5 the output period will be $2*(5+1) = 12$ times the input clock period.

A 1 MHz clock and a 50 MHz clock are available on the "`Clocks`" symbol (`Clk_1MHz` and `Clk_50MHz`, respectively). A buzzer symbol is also available to define the output pads.

Standard (1 kHz) timer interrupts can then be used to control the duration of the note.

23.6   Installing Your Hardware Design

Your own customisation of the FPGA can be produced and simulated using the familiar Cadence design suite. As in other labs. design synthesis is provided by the Xilinx Synthesis option in the Verilog Simulation Environment; downloading can be performed with Komodo.

A correctly synthesized FPGA design will produce a file:

∼/Cadence/COMP22712/xilinx_compile/<design_name>.bit

Remember, any synthesis/error reports may be found in:

∼/Cadence/COMP22712/xilinx_compile/<design_name>.<ext>

Downloading is performed using the "**Features**" extension to Komodo. Clicking this pops up a window which gives access to a browser where the relevant file can be selected. (Ensure the selected device is "Spartan XCS10XL", the default option.) Clicking "**Download**" will then send the synthesized design down to the lab. board.

ARM programming puzzle

Sign extend a 10-bit number in `R8`.

Target: 2 instructions

---

24 | Exercise 9: Project

Choose **one** of the following projects (**or invent** your own) and develop it, as far as you can in the time remaining, using techniques you have learnt in earlier exercises. Concentrate on demonstrating principles gleaned from earlier exercises rather than spending too much effort on elaborate applications code. For example, projects involving user-developed hardware as well as software will be credited more favourably.

Whichever project you choose, use your own imagination to think of ways in which it can (sensibly) develop and try to implement those too. Alternatively, define a project of your own of similar scope. Remember the lab. staff are there to help and advise on project work so whatever you plan to do, talk it over first.

24.1 Practical and Submission

You should submit this exercise for assessment and feedback. Submit everything that you have done for this exercise; the more appropriate techniques you can demonstrate, the better.

24.1.1 Suggestions

Calculator Build a calculator capable of adding, subtracting, multiplying and dividing (and maybe more?) decimal numbers.
If you're short of keys you can add an extra keyboard or use the normal calculator "2nd function" approach.

Alarm clock Make a clock capable of displaying (and being set to the correct time). Include an alarm facility which can be set by the user and will 'go off' at the correct time.

Morse code trainer Modify your "Hello World" programme to 'print' its output as Morse code.
Extend your keyboard input routines to enable the sending of "text messages".

Electronic organ Write a programme which plays notes as specified by buttons on the keypad.

Dimmer switch Control the brightness of your LEDs using interrupts and Pulse Width Modulation[1] (PWM).

Music Player Write an interpreter ('virtual machine') which reads a list of notes (pitch & duration — possibly volume too) from an input file and plays the tune so described. Remember you will need codes for 'rests' and (hopefully!) 'end of tune'.
For a quick test there are some files available in:
`/opt/info/courses/COMP22712/Code_examples/Tunes/`
Remember that you could add more buzzers to create "harmony".

---

[1] If you don't know about this, Google will no doubt help.

Hints for writing an tune interpreter:

- Simplify your task by translating notes to frequency/period with a look-up table.
- Note that, as an octave is a factor of two, the frequency can be transposed by an octave with a left or right shift.
- Remember that two notes of the same pitch may need a separator.
- Don't forget a 'silent' setting, for rests and the end of the tune!

Acoustics for non-musicians

An "octave" is a factor of 2 in frequency and is divided into eight notes (but is really twelve semitone divisions). In modern tuning — known as "equal temperament" — each of these divisions is the same on a logarithmic scale. This means each semitone has a frequency about 6% different from an adjacent one.

$$factor = 2^{\frac{1}{12}} \approx 1.059$$

Starting a scale at 1 kHz the semitones have the frequencies as shown in the table below. The bold text denotes the notes of the major scale.

Note	Frequency (Hz)	Period (μs)
DoâĂš	**2000**	**500**
Ti	**1888**	**530**
	1782	561
La	**1682**	**595**
	1587	630
So	**1498**	**667**
	1414	707
Fa	**1335**	**749**
Mi	**1260**	**794**
	1189	841
Re	**1122**	**891**
	1059	944
Do	**1000**	**1000**

The period of the waveform (the time taken to complete a whole cycle) is also given.

25 | Component Libraries

25.1   ENGLAB_22712

The following components may be of interest to the system builder:

Bus_8 (_16)	The processor interface to address and control inputs and bidirectional (tristate) data signals.
Clocks	Clock inputs including 50 MHz (CLK_sel) and 1 MHz derived from this.
Interrupts	Two interrupt outputs IRQ, FIQ by which interrupts can be requested on the CPU.
PIO_8	Simple I/O system with eight 8-bit PIOs (default FPGA configuration)
PIO	The functional part of a simple PIO; not a stand-alone component.
Port<xx>	Sets of I/O pads and their drivers, mapped to specific I/O locations.
Buzzer	Buzzer pins; *alternative* to the PIO 'S0A'.
Buttons	Button inputs common with the microcontroller port.
Divider	A programmable clock divider by 2 x (N + 1).
Decoder	A partial address decoder for building I/O systems in the FPGA.
Downcount8	8-bit down counter.
Timer8	8-bit timer with prescale.
One_shot	Single shot timer which may be used to generate interrupts.

Where <xx> is from {0a, 0b, 1a, 1b, 2a, 2b, 3a, 3b} and each corresponds to a row of one of the front expansion connectors on the PCB.

There are also some components in this library which are used to support these cells. The names of these should either be self-evident or from the Xilinx macro library (e.g. CB8CE = Counter, Binary, 8-bit with Clear and Enable).

25.2   SPARTAN-3

This is the library supplied by the device manufacturer with the development tools. It includes the standard set of simple gates (AND, OR etc.) and various variants on edge-triggered flip flops. Some of the most commonly used examples are embolded in the list, below:

25.2.1   Basic gates (etc.)

GND, **VCC**, PULLDOWN, PULLUP
INV
AND2, AND2B1, AND2B2, AND3, AND3B1, AND3B2, AND3B3, AND4, AND4B1, AND4B2, AND4B3, AND4B4, AND5, AND5B1, AND5B2, AND5B3, AND5B4, AND5B5
NAND2, NAND2B1, NAND2B2, NAND3, NAND3B1, NAND3B2, NAND3B3, NAND4, NAND4B1, NAND4B2, NAND4B3, NAND4B4, NAND5, NAND5B1, NAND5B2, NAND5B3, NAND5B4, NAND5B5
NOR2, NOR2B1, NOR2B2, NOR3, NOR3B1, NOR3B2, NOR3B3, NOR4, NOR4B1, NOR4B2, NOR4B3, NOR4B4, NOR5, NOR5B1, NOR5B2, NOR5B3, NOR5B4, NOR5B5
OR2, OR2B1, OR2B2, OR3, OR3B1, OR3B2, OR3B3, OR4, OR4B1, OR4B2, OR4B3, OR4B4, OR5, OR5B1, OR5B2, OR5B3, OR5B4, OR5B5
XNOR2, XNOR3, XNOR4, XNOR5, **XOR2**, XOR3, XOR4, XOR5

25.2.2 Buffers, including clock buffers

BUF, BUFCF, BUFG, BUFGCE, BUFGCE_1, BUFGDLL, BUFGMUX, BUFGMUX_1, BUFGP

25.2.3 Flip-flops & latches

FD, FDC, **FDCE**, FDCE_1, FDCP, FDCPE, FDCPE_1, FDCP_1, FDC_1, FDDRCPE, FDDRRSE,
FDE, FDE_1, FDP, FDPE, FDPE_1, FDP_1, FDR, FDRE, FDRE_1, FDRS, FDRSE, FDRSE_1,
FDRS_1, FDR_1, FDS, FDSE, FDSE_1, FDS_1, FD_1
LD, LDC, LDCE, LDCE_1, LDCP, LDCPE, LDCPE_1, LDCP_1, LDC_1, LDE, LDE_1, LDP, LDPE,
LDPE_1, LDP_1, LD_1

25.2.4 Arithmetic support

MUXCY, MUXCY_D, MUXCY_L, MUXF5, MUXF5_D, MUXF5_L, MUXF6, MUXF6_D, MUXF6_L,
MUXF7, MUXF7_D, MUXF7_L, MUXF8, MUXF8_D, MUXF8_L
XORCY, XORCY_D, XORCY_L
MULT18X18, MULT18X18S, MULT_AND

25.2.5 Memory

RAM16X1D, RAM16X1D_1, RAM16X1S, RAM16X1S_1, RAM16X2S, RAM16X4S, RAM16X8S,
RAM32X1S, RAM32X1S_1, RAM32X2S, RAM32X4S, RAM32X8S, RAM64X1S, RAM64X1S_1,
RAM64X2S, RAMB16_S1, RAMB16_S18, RAMB16_S18_S18, RAMB16_S18_S36, RAMB16_S1_S1,
RAMB16_S1_S18, RAMB16_S1_S2, RAMB16_S1_S36, RAMB16_S1_S4, RAMB16_S1_S9,
RAMB16_S2, RAMB16_S2_S18, RAMB16_S2_S2, RAMB16_S2_S36, RAMB16_S2_S4,
RAMB16_S2_S9, RAMB16_S36, RAMB16_S36_S36,
RAMB16_S4, RAMB16_S4_S18, RAMB16_S4_S36, RAMB16_S4_S4, RAMB16_S4_S9, RAMB16_S9,
RAMB16_S9_S18, RAMB16_S9_S36, RAMB16_S9_S9
ROM128X1, ROM16X1, ROM256X1, ROM32X1, ROM64X1

25.2.6 I/O pads, buffers and latches

IBUF, IBUFDS, IBUFG, IBUFGDS, IFDDRCPE, IFDDRRSE, IOBUF, IOBUFDS, IOPAD, IPAD,
KEEPER, OBUF, OBUFDS, OBUFT, OBUFTDS, OFDDRCPE, OFDDRRSE, OFDDRTCPE, OFD-
DRTRSE, OPAD

25.2.7 Debug, documentation & miscellaneous

BSCAN_SPARTAN3, CAPTURE_SPARTAN3, CONFIG, DCM
STARTBUF_SPARTAN3, STARTUP_SPARTAN3, TBLOCK, TITLE, FMAP
LUT1, LUT1_D, LUT1_L, LUT2, LUT2_D, LUT2_L, LUT3, LUT3_D, LUT3_L, LUT4, LUT4_D,
LUT4_L

26 | ARM Assembly Language Mnemonics and Directives

These are example mnemonics as a quick reference guide. This is not a fully comprehensive list. "cc" shows the position of the condition code (if any).

Data Operations

```
ADDccS  Rd, Rn, Rm/#nn
ADCcc   Rd, Rn, Rm/#nn
SUBcc   Rd, Rn, Rm/#nn
SBCcc   Rd, Rn, Rm/#nn
RSBccS  Rd, Rn, Rm/#nn
RSCcc   Rd, Rn, Rm/#nn

ANDcc   Rd, Rn, Rm/#nn
ORRcc   Rd, Rn, Rm/#nn
EORcc   Rd, Rn, Rm/#nn
BICcc   Rd, Rn, Rm/#nn

MOVcc   Rd, Rm/#nn
MVNcc   Rd, Rm/#nn

CMPcc   Rn, Rm/#nn
CMNcc   Rn, Rm/#nn
TSTcc   Rn, Rm/#nn
TEQcc   Rn, Rm/#nn

MULcc   Rd, Rm, Rs
MLAccS  Rd, Rm, Rs, Rn
```

Loads and Stores

```
LDRcc   Rd, [Rn]             ; Reg. Indirect
LDRccH  Rd, [Rn, Rm/\#nn]    ; Pre-indexed
LDRccB  Rd, [Rn], Rm/\#nn    ; Post Indexed
STRcc   Rd, [Rn, Rm/\#nn]!   ; Pre-ind. w/back

LDMccFD Rd, {register list}  ; No w/back
STMccFD Rd!, {register list} ; With w/back
```

Control Transfer

```
Bcc     Label
BLcc    Procedure_start
SVCcc   Number
```

Special Control Operations

```
        MRScc   Rd, CPSR
        MRScc   Rd, SPSR
        MSRcc   CPSR_f, #nn         ; Sets flags
        MSRcc   SPSR_c, Rm          ; Sets mode in SPSR
```

Assembler Supplied 'Pseudo Operations'

```
        NOP                         ; No operation
        MOV     Rd, \#-nn           ; assembles to MVN
        CMP     Rn, \#-nn           ; assembles to CMN

        ADR     Rd, label           ; e.g. ADD Rd, PC, #nn
        ADRL    Rd, label           ; 'enough' instructions

        LDR     Rd, =nnnnnnnn       ; Load constant
```

Note: the last operation (`LDR`) places the constant (nnnnnnnn) somewhere 'convenient' in memory (probably at the end of the code) and plants an instruction which loads this value. The pseudo-operation therefore generates two words, which are unlikely to be adjacent in the memory map. If the constant is 'short' enough, the assembler will substitute a `MOV` instruction. *Try experimenting!*

Directives

Directive mnemonics are case insensitive.

```
        ORG     &1000               ; Set assembly address
        ALIGN                       ; Next 4-byte boundary
        ALIGN   N                   ; Next N-byte boundary

        DEFB    0, 1, 2, "bytes"    ; Define bytes
        DEFH    &ABCD, 2+2          ; Define halfword(s)
        DEFW    &12345678           ; Define word(s)

label   EQU     value               ; Set value of "Label"
        DEFS    &20                 ; Reserve &20 bytes

        INCLUDE <filename>          ; What it says!
```

Condition Codes

cc	Meaning	Coding	Flag condition
EQ	Equal	0	Z
NE	Not Equal	1	$\overline{Z}$
CS/HS	Carry Set/Higher or Same (unsigned)	2	C
CC/LO	Carry Clear/Lower (unsigned)	3	$\overline{C}$
MI	Minus (negative)	4	N
PL	Plus (positive)	5	$\overline{N}$
VS	Overflow Set	6	V
VC	Overflow Clear	7	$\overline{V}$
HI	Higher (unsigned)	8	$C \wedge \overline{Z}$
LS	Lower or Same (unsigned)	9	$\overline{C} \vee Z$
GE	Greater than or Equal (signed)	A	$\overline{N \oplus V}$
LT	Less Than (signed)	B	$N \oplus V$
GT	Greater Than (signed)	C	$(\overline{N \oplus V}) \wedge \overline{Z}$
LE	Less than or Equal (signed)	D	$(N \oplus V) \vee Z$
AL	Always	E	TRUE

Expression Operators

It is possible, and often valuable, to allow the assembler to work out some values (such as the length of strings and tables). Thus instead of a simple number an integer expression can be used. The following is a summary of the most useful ones.

Unary operators: +, -, ~

Binary operators, highest precedence first:

Logical shifts:	{ ⟨⟨ LSL SHL} { ⟩⟩ LSR SHR}
Logical AND:	AND
Logical ORs:	{| OR} {^ EOR}
Multiplication:	* {/ DIV} {\ MOD}
Addition:	+ -

Numbers

Numeric constants default to decimal (base 10).

Hexadecimal numbers should be prefixed with "&", "$", or "0x".

Binary numbers should be prefixed with ":" or "0b".

Octal numbers — if you really want these — should be prefixed with "@"

Signed and Unsigned integers

The ARM handles 32-bit quantities in its registers. These may be interpreted as signed (two's complement) or unsigned integers (or other things too). Arithmetic operations treat these identically; the processor *doesn't care* what you think they are. If you care then they can be distinguished using different condition codes as part of the programme.

It's easy to regard everything as signed integers but this is often poor practice. What about the offset into that table/array? Can it be negative? Probably more integer variables are unsigned than signed in a typical programme! Try to use the appropriate 'type'.

Java integers are all signed but some languages (such as C) allow unsigned types.

27 | ASCII Character Set

00	^@	NUL	Null		10	^P	DLE	Data link escape
01	^A	SOH	Start of header		11	^Q	DC1	Device control 1
02	^B	STX	Start of text		12	^R	DC2	Device control 2
03	^C	ETX	End of text		13	^S	DC3	Device control 3
04	^D	EOT	End of transmission		14	^T	DC4	Device control 4
05	^E	ENQ	Enquire		15	^U	NAK	Negative Acknowledge
06	^F	ACK	Acknowledge Idle		16	^V	SYN	Synchronous Idle
07	^G	BEL	Bell block		17	^W	ETB	End of transmitted block
08	^H	BS	Back space		18	^X	CAN	Cancel
09	^I	HT	Horizontal Tabulate		19	^Y	EM	End of medium
0A	^J	LF	Line feed		1A	^Z	SUB	Substitute
0B	^K	VT	Vertical Tabulate		1B	^[	ESC	Escape
0C	^L	FF	Form feed		1C	^	FS	File separator
0D	^M	CR	Carriage return		1D	^]	GS	Group separator
0E	^N	SO	Shift out		1E	^^	RS	Record separator
0F	^O	SI	Shift in		1F	^_	US	Unit separator

20		SP	Space	40	@	60	`
21	!			41	A	61	a
22	"			42	B	62	b
23	#			43	C	63	c
24	$			44	D	64	d
25	%			45	E	65	e
26	&			46	F	66	f
27	´			47	G	67	g
28	(			48	H	68	h
29	)			49	I	69	i
2A	*			4A	J	6A	j
2B	+			4B	K	6B	k
2C	,			4C	L	6C	l
2D	-			4D	M	6D	m
2E	.			4E	N	6E	n
2F	/			4F	O	6F	o
30	0			50	P	70	p
31	1			51	Q	71	q
32	2			52	R	72	r
33	3			53	S	73	s
34	4			54	T	74	t
35	5			55	U	75	u
36	6			56	V	76	v
37	7			57	W	77	w
38	8			58	X	78	x
39	9			59	Y	79	y
3A	:			5A	Z	7A	z
3B	;			5B	[	7B	{
3C	<			5C	\	7C	|
3D	=			5D	]	7D	}
3E	>			5E	^	7E	~
3F	?			5F	_	7F	DEL, RUBOUT