2. Inside the ARM

In the previous chapter, we started by considering instructions executed by a mythical processor with mnemonics like **ON** and **OFF**. Then we went on to describe some of the features of an actual processor - the ARM. This chapter looks in much more detail at the ARM, including the programmer's model and its instruction types. We'll start by listing some important attributes of the CPU:

Word size

The ARM's word length is 4 bytes. That is, it's a 32-bit micro and is most at home when dealing with units of data of that length. However, the ability to process individual bytes efficiently is important - as character information is byte oriented - so the ARM has provision for dealing with these smaller units too.

Memory

When addressing memory, ARM uses a 26-bit address value. This allows for 2^{26} or 64M bytes of memory to be accessed. Although individual bytes may be transferred between the processor and memory, ARM is really word-based. All word-sized transfers must have the operands in memory residing on word-boundaries. This means the instruction addresses have to be multiples of four.

I/O

Input and output devices are memory mapped. There is no concept of a separate I/O address space. Peripheral chips are read and written as if they were areas of memory. This means that in practical ARM systems, the memory map is divided into three areas: RAM, ROM, and input/output devices (probably in decreasing order of size).

Registers

The register set, or programmer's model, of the ARM could not really be any simpler. Many popular processors have a host of dedicated (or special-purpose) registers of varying sizes which may only be used with certain instructions or in particular circumstances. ARM has sixteen 32-bit registers which may be used without restriction in any instruction. There is very little dedication - only one of the registers being permanently tied up by the processor.

Instructions

As the whole philosophy of the ARM is based on 'fast and simple', we would expect the instruction set to reflect this, and indeed it does. A small, easily remembered set of instruction types is available. This does not imply a lack of power, though. Firstly, instructions execute very quickly, and secondly, most have useful extras which add to their utility without detracting from the ease of use.

2.1 Memory and addressing

The lowest address that ARM can use is that obtained by placing 0s on all of the 26 address lines - address &0000000. The highest possible address is obtained by placing 1s on the 26 address signals, giving address &3FFFFFF. All possible combinations between these two extremes are available, allowing a total of 64M bytes to be addressed. Of course, it is very unlikely that this much memory will actually be fitted in current machines, even with the ever-increasing capacities of RAM and ROM chips. One or four megabytes of RAM is a reasonable amount to expect using today's technology.

Why allow such a large address range then? There are several good reasons. Firstly, throughout the history of computers, designers have under-estimated how much memory programmers (or rather their programs) can actually use. A good maxim is 'programs will always grow to fill the space available. And then some.' In the brief history of microprocessors, the addressing range of CPUs has grown from 256 single bytes to 4 billion bytes (i.e. 4,000,000,000 bytes) for some 32-bit micros. As the price of memory continues to fall, we can expect 16M and even 32M byte RAM capacities to become available fairly cheaply.

Another reason for providing a large address space is to allow the possibility of using virtual memory. Virtual memory is a technique whereby the fast but relatively expensive semiconductor RAM is supplemented by slower but larger capacity magnetic storage, e.g. a Winchester disc. For example, we might allocate 16M bytes of a Winchester disc to act as memory for the computer. The available RAM is used to 'buffer' as much of this as possible, say 512K bytes, making it rapidly accessible. When the need arises to access data which is not currently in RAM, we load it in from the Winchester.

Virtual memory is an important topic, but a detailed discussion of it is outside the scope of this book. We do mention some basic virtual memory techniques when talking about the memory controller chip in Chapter Seven.

The diagram below illustrates how the ARM addresses memory words and bytes.

bit 31........................bit 0

Address 0	byte 3	byte 2	byte 1	byte 0
Address 4	byte 7	byte 6	byte 5	byte 4
Address 8	byte 11	byte 10	byte 9	byte 8
Address 12	byte 15	byte 14	byte 13	byte 12

The addresses shown down the left hand side are word addresses, and increase in steps of four. Word addresses always have their least two significant bits set to zero and the other 24 bits determine which word is required. Whenever the ARM fetches an instruction from memory, a word address is used. Additionally, when a word-size operand is transferred from the ARM to memory, or vice versa, a word address is used.

When byte-sized operands are accessed, all 26 address lines are used, the least significant two bits specifying which byte within the word is required. There is a signal from the ARM chip which indicates whether the current transfer is a word or byte-sized one. This signal is used by the memory system to enable the appropriate memory chips. We will have more to say about addressing in the section on data transfer instructions.

The first few words of ARM memory have special significance. When certain events occur, e.g. the ARM is reset or an illegal instruction is encountered, the processor automatically jumps to one of these first few locations. The instructions there perform the necessary actions to deal with the event. Other than this, all ARM memory was created equal and its use is determined solely by the designer of the system.

For the rest of this section, we give brief details of the use of another chip in the ARM family called the MEMC. This information is not vital to most programmers, and may be skipped on the first reading.

A topic which is related to virtual memory mentioned above, and which unlike that, is within the scope of this book, is the relationship between 'physical' and 'logical' memory in ARM systems. Many ARM-based machines use a device called the Memorv Controller - MEMC - which is the Video Controller and I/O Controller, called VIDC and IOC respectively.)

When an ARM-based system uses MEMC, its memory map is divided into three main areas. The bottom half - 32M bytes - is called logical RAM, and is the memory that most programs 'see' when they are executing. The next 16M bytes is allocated to physical RAM. This area is only visible to system programs which use the CPU in a special mode called *supervisor mode*. Finally, the top 16M bytes is occupied by ROM and I/O devices.

The logical and physical RAM is actually the same thing, and the data is stored in the same RAM chips. However, whereas physical RAM occupies a contiguous area from address 32M to 32M+(memory size)-1, logical RAM may be scattered anywhere in the bottom 32M bytes. The physical RAM is divided into 128 'pages'. The size of a page depends on how much RAM the machine has. For example, in a 1M byte machine, a page is 8K bytes; in a 4M byte machine (the maximum that the current MEMC chip can handle) it is 32K bytes.

A table in MEMC is programmed to control where each physical page appears in the logical memory map. For example, in a particular system it might be convenient to have the screen memory at the very top of the 32M byte logical memory area. Say the page size is 8K bytes and 32K is required for the screen. The MEMC would be programmed so that four pages of physical RAM appear at the top 32K bytes of the logical address space. These four pages would be accessible to supervisor mode programs at both this location and in the appropriate place in the physical memory map, and to non-supervisor programs at just the logical memory map position.

When a program accesses the logical memory, the MEMC looks up where corresponding physical RAM is and passes that address on to the RAM chips. You could imagine the address bus passing through the MEMC on its way to the memory, and being translated on the way. This translation is totally transparent to the programmer. If a program tries to access a logical memory address for which there is no corresponding physical RAM (remember only at most 4M bytes of the possible 32M can be occupied), a signal called 'data abort' is activated on the CPU. This enables attempts to access 'illegal' locations to be dealt with.

As the 4M byte limit only applies to the current MEMC chip, there is no reason why a later device shouldn't be able to access a much larger area of physical memory.

Because of the translation performed by MEMC, the logical addresses used way, this means that a 1M byte machine will not necessarily appear to have all of this RAM at the bottom of the memory map; it might be scattered into different areas. For example, one 'chunk' of memory might be used for the screen and mapped onto a high address, whereas another region, used for application programs say, might start at a low address such as &8000.

Usually, the presence of MEMC in a system is if no consequence to a program, but it helps to explain how the memory map of an ARM-based computer appears as it does.

2.2 Programmer's model

This section describes the way in which the ARM presents itself to the programmer. The term 'model' is employed because although it describes what the programmer sees when programming the ARM, the internal representation may be very different. So long as programs behave as expected from the description given, these internal details are unimportant.

Occasionally however, a particular feature of the processor's operation may be better understood if you know what the ARM is getting up to internally. These situations are explained as they arise in the descriptions presented below.

As mentioned above, ARM has a particularly simple register organisation, which benefits both human programmers and compilers, which also need to generate ARM programs. Humans are well served because our feeble brains don't have to cope with such questions as 'can I use the X register as an operand with the ADD instruction?' These crop up quite frequently when programming in assembler on certain micros, making coding a tiresome task.

There are sixteen user registers. They are all 32-bits wide. Only two are dedicate; the others are general purpose and are used to store operands, results and pointers to memory. Of the two dedicated registers, only one of these is permanently used for a special purpose (it is the PC). Sixteen is quite a large number of registers to provide, some micros managing with only one general purpose register. These are called accumulator-based processors, and the 6502 is an example of such a chip.

All of the ARM's registers are general purpose. This means that wherever an instruction needs a register to be specified as an operand, any one of them may be used. This gives the programmer great freedom in deciding which registers to use for which purpose.

The motivation for providing a generous register set stems from the way in which the ARM performs most of its operations. All data manipulation instructions use registers. That is, if you want to add two 32-bit numbers, both of the numbers must be in registers, and the result is stored in a third register. It is not possible to add a number in memory to a register, or vice versa. In fact, the only time the ARM accesses memory is to fetch instructions and when executing one of the few register-to-memory transfer instructions.

So, given that most processing is restricted to using the fast internal registers, it is only fair that a reasonable number of them is provided. Studies by computer scientists have shown that eight general-purpose registers is sufficient for most types of program, so 16 should be plenty.

When designing the ARM, Acorn may well have been tempted to include even more registers, say 32, using the 'too much is never enough' maxim mentioned above. However, it is important to remember that if an instruction is to allow any register as an operand, the register number has to be encoded in the instruction. 16 registers need four bits of encoding; 32 registers would need five. Thus by increasing the number of registers, they would have decreased the number of bits available to encode other information in the instructions.

Such trade-offs are common in processor design, and the utility of the design depends on whether the decisions have been made wisely. On the whole, Acorn seems to have hit the right balance with the ARM.

There is an illustration of the programmer's model overleaf.

In the diagram, 'undedicated' means that the hardware imposes no particular use for the register. 'Dedicated' means that the ARM uses the register for a particular function - R15 is the PC. 'Semi-dedicated' implies that occasionally the hardware might use the register for some function (for storing addresses), but at other times it is undedicated. 'General purpose' indicates that if an instruction requires a register as an operand, any register may be specified.

As R0-R13 are undedicated, general purpose registers, nothing more needs to be said about them at this stage.

R0	Undedicated, general purpose
R1	Undedicated, general purpose
R2	Undedicated, general purpose
R3	Undedicated, general purpose
R4	Undedicated, general purpose
R5	Undedicated, general purpose
R6	Undedicated, general purpose
R7	Undedicated, general purpose
R8	Undedicated, general purpose
R9	Undedicated, general purpose
R10	Undedicated, general purpose
R11	Undedicated, general purpose
R12	Undedicated, general purpose
R13	Undedicated, general purpose
R14	Semi-dedicated, general purpose (link)
R15	Dedicated, general purpose (PC)

Special registers

Being slightly different from the rest, R14 and R15 are more interesting, especially R15. This is the only register which you cannot use in the same way as the rest to hold operands and results. The reason is that the ARM uses it to store the program counter and status register. These two components of R15 are explained below.

Register 14 is usually free to hold any value the user wishes. However, one instruction, 'branch with link', uses R14 to keep a copy of the PC. The next chapter describes branch with link, along with the rest of the instruction set, and this use of R14 is explained in more detail there.

The program counter

31	30	29	28	27	26	25 ... 2	1	0
N	Z	C	V	I	F	Program counter	S1	S0

R15 is split into two parts. This is illustrated below:
Bits 2 to 25 are the program counter (PC). That is, they hold the word address of the next instruction to be fetched. There are only 24 bits (as opposed to the full 26) because instructions are defined to reside on word boundaries. Thus the two lowest bits of an instruction's address are always zero, and there is no need to store them. When R15 is used to place the

When the ARM is reset, the program counter is set to zero, and instructions are fetched starting from that location. Normally, the program counter is incremented after every instruction is fetched, so that a program is executed in sequence. However, some instructions alter the value of the PC, causing non-consecutive instructions to be fetched. This is how **`IF…THEN…ELSE`** and **`REPEAT…UNTIL`** type constructs are programmed in machine code.

Some signals connected to the ARM chip also affect the PC when they are activated. Reset is one such signal, and as mentioned above it causes the PC to jump to location zero. Others are IRQ and FIQ, which are mentioned below, and memory abort.

Status register

The remaining bits of R15, bits 0, 1 and 26-31, form an eight-bit status register. This contains information about the state of the processor. There are two types of status information: result status and system status. The former refers to the outcome of previous operations, for example, whether a carry was generated by an addition operation. The latter refers to the four operating modes in which the ARM may be set, and whether certain events are allowed to interrupt its processing.

Here is the layout of the status register portion of R15:

	bit 31	N	Negative result flag
Result	bit 30	Z	Zero result flag
Status	bit 29	C	Carry flag
	bit 28	V	Overflowed result flag
	bit 27	IRQ	Interrupt disable flag
System	bit 26	FIQ	Fast interrupt disable flag
Status	bit 0	S0	Processor mode 0
	bit 1	S1	Processor mode 1

The result status flags are affected by the register-to-register data operations. The exact way in which these instructions change the flags is described along with the instructions. No other instructions affect the flags, unless they are loaded explicitly (along with the rest of R15) from memory.

As each flag is stored in one bit, it has two possible states. If a flag bit has the value 1, it is said to be true, or set. If it has the value 0, the flag is false or cleared. For example, if bits 31 to 28 of R15 were 1100, the N and Z flags would be set, and V and C would be cleared.

All instructions may execute conditionally on the result flags. That is to say, exists, otherwise the instruction is ignored. Additionally, an instruction may be unconditional, so that it executes regardless of the state of the flags.

The processor mode flags hold a two-bit number. The state of these two bits determine the 'mode' in which the ARM executes, as follows:

s1 s0	**Mode**
0 0	User
0 1	FIQ or fast interrupt
1 0	IRQ or interrupt
1 1	SVC or supervisor

The greater part of this book is concerned only with user mode. The other modes are 'system' modes which are only required by programs which will have generally already been written on the machine you are using. Briefly, supervisor mode is entered when the ARM is reset or certain types of error occur. IRQ and FIQ modes are entered under the interrupt conditions described below.

In non-user modes, the ARM looks and behaves in a very similar way to user mode (which we have been describing). The main difference is that certain registers (e.g. R13 and R14 in supervisor mode) are replaced by 'private copies' available only in that mode. These are called R13_SVC and R14_SVC. In user mode, the supervisor mode's versions of R13 and R14 are not visible, and vice versa. In addition, S0 and S1 may not be altered in user mode, but may be in other modes. In IRQ mode, the extra registers are R13_IRQ and R14_IRQ; in FIQ mode there are seven of them - R8_FIQ to R14_FIQ.

Non-user modes are used by 'privileged' programs which may have access to hardware which the user is not allowed to touch. This is possible because a signal from the ARM reflects the state of S0 and S1 so external hardware may determine if the processor is in a user mode or not.

Finally, the status bits FIQ and IRQ are used to enable or disable the two interrupts provided by the processor. An interrupt is a signal to the chip which, when activated, causes the ARM to suspend its current action (having finished the current instruction) and set the program counter to a pre-determined value. Hardware such as disc drives use interrupts to ask for attention when they require servicing.

The ARM provides two interrupts. The IRQ (which stands for interrupt request) signal will cause the program to be suspended only if the IRQ bit in the status register is cleared. If that bit is set, the interrupt will be ignored by except that the FIQ bit enables/disables it. If a FIQ interrupt is activated, the IRQ bit is set automatically, disabling any IRQ signal. The reverse is not true however, and a FIQ interrupt may be processed while an IRQ is active.

As mentioned above, the supervisor, FIQ and IRQ modes are rarely of interest to programmers other than those writing 'systems' software, and the system status bits of R15 may generally be ignored. Chapter Seven covers the differences in programming the ARM in the non-user modes.

2.3 The instructions set

To complement the regular architecture of the programmer's model, the ARM has a well-organised, uniform instruction set. In this section we give an overview of the instruction types, and defer detailed descriptions until the next chapter.

General properties

There are certain attributes that all instructions have in common. All instructions are 32-bits long (i.e. they occupy one word) and must lie on word boundaries. We have already seen that the address held in the program counter is a word address, and the two lowest bits of the address are set to zero when an instruction is fetched from memory.

The main reason for imposing the word-boundary restriction is one of efficiency. If an instruction were allowed to straddle two words, two accesses to memory would be required to load a single instruction. As it is, the ARM only ever has to access memory once per instruction fetch. A secondary reason is that by making the two lowest bits of the address implicit, the program address range of the ARM is increased from the 24 bits available in R15 to 26 bits - effectively quadrupling the addressing range.

A 32-bit instruction enables 2^{32} or about 4 billion possible instructions. Obviously the ARM would not be much of a reduced instruction set computer if it used all of these for wildly differing instructions. However, it does use a surprisingly large amount of this theoretical instruction space.

The instruction word may be split into different 'fields'. A field is a set of (perhaps just one) contiguous bits. For example, bits 28 to 31 of R15 could be called the result status field. Each instruction word field controls a particular aspect of the interpretation of the instruction. It is not necessary to know where these fields occur within the word and what they mean, as the assembler does that for you using the textual representation of instruction.

One field which is worth mentioning now is the condition part. Every ARM instruction has a condition code encoded into four bits of the word. Four bits enable up to 16 conditions to be specified, and all of these are used. Most instructions will use the 'unconditional' condition, i.e. they will execute regardless of the state of the flags. Other conditions are 'if zero', 'if carry set', 'if less than' and so on.

Instruction classes

There are five types of instruction. Each class is described in detail in its own section of the next chapter. In summary, they are:

Data operations

This group does most of the work. There are sixteen instructions, and they have very similar formats. Examples of instructions from this group are **ADD** and **CMP**, which add and compare two numbers respectively. As mentioned above, the operands of these instructions are always in registers (or an *immediate* number stored in the instruction itself), never in memory.

Load and save

This is a smaller group of two instructions: load a register and save a register. Variations include whether bytes or words are transferred, and how the memory location to be used is obtained.

Multiple load and save

Whereas the instructions in the previous group only transfer a single register, this group allows between one and 16 registers to be moved between the processor and memory. Only word transfers are performed by this group.

Branching

Although the PC may be altered using the data operations to cause a change in the program counter, the branch instruction provides a convenient way of reaching any part of the 64M byte address space in a single instruction. It causes a *displacement* to be added to the current value of the PC. The displacement is stored in the instruction itself.

SWI

This one-instruction group is very important. The abbreviation stands for 'SoftWare Interrupt'. It provides the way for user's programs to access the facilities provided by the operating system. All ARM-based computers provide a certain amount of pre-written software to perform such tasks as instructions, the user's program may utilise this operating system software, obviating the need to write the routines for each application.

Floating point

The first ARM chips do not provide any built-in support for dealing with floating point, or real, numbers. Instead, they have a facility for adding co-processors. A co-processor is a separate chip which executes special-purpose instructions which the ARM CPU alone cannot handle. The first such processor will be one to implement floating point instructions. These instructions have already been defined, and are currently implemented by software. The machine codes which are allocated to them are illegal instructions on the ARM-I so system software can be used to 'trap' them and perform the required action, albeit a lot slower than the co-processor would.

Because the floating point instructions are not part of the basic ARM instruction set, they are not discussed in the main part of this book, but are described in Appendix B.