# 2 The ARM Architecture

## Summary of chapter contents

The ARM processor is a *Reduced Instruction Set Computer* (RISC). The RISC concept, as we saw in the previous chapter, originated in processor research programmes at Stanford and Berkeley universities around 1980.

In this chapter we see how the RISC ideas helped shape the ARM processors. The ARM was originally developed at Acorn Computers Limited of Cambridge, England, between 1983 and 1985. It was the first RISC microprocessor developed for commercial use and has some significant differences from subsequent RISC architectures. The principal features of the ARM architecture are presented here in overview form; the details are postponed to subsequent chapters.

In 1990 ARM Limited was established as a separate company specifically to widen the exploitation of ARM technology, since when the ARM has been licensed to many semiconductor manufacturers around the world. It has become established as a market-leader for low-power and cost-sensitive embedded applications.

No processor is particularly useful without the support of hardware and software development tools. The ARM is supported by a toolkit which includes an instruction set emulator for hardware modelling and software testing and benchmarking, an assembler, C and C++ compilers, a linker and a symbolic debugger.



Scanned with CamScanner

## 2.1 The Acorn RISC Machine

The first ARM processor was developed at Acorn Computers Limited, of Cambridge, England, between October 1983 and April 1985. At that time, and until the formation of Advanced RISC Machines Limited (which later was renamed simply ARM Limited) in 1990, ARM stood for **Acorn RISC Machine**.

Acorn had developed a strong position in the UK personal computer market due to the success of the BBC (British Broadcasting Corporation) microcomputer. The BBC micro was a machine powered by the 8-bit 6502 microprocessor and rapidly became established as the dominant machine in UK schools following its introduction in January 1982 in support of a series of television programmes broadcast by the BBC. It also enjoyed enthusiastic support in the hobbyist market and found its way into a number of research laboratories and higher education establishments.

Following the success of the BBC micro, Acorn's engineers looked at various microprocessors to build a successor machine around, but found all the commercial offerings lacking. The 16-bit CISC microprocessors that were available in 1983 were slower than standard memory parts. They also had instructions that took many clock cycles to complete (in some cases, many hundreds of clock cycles), giving them very long interrupt latencies. The BBC micro benefited greatly from the 6502's rapid interrupt response, so Acorn's designers were unwilling to accept a retrograde step in this aspect of the processor's performance.

As a result of these frustrations with the commercial microprocessor offerings, the design of a proprietary microprocessor was considered. The major stumbling block was that the Acorn team knew that commercial microprocessor projects had absorbed hundreds of man-years of design effort. Acorn could not contemplate an investment on that scale since it was a company of only just over 400 employees in total. It had to produce a better design with a fraction of the design effort, and with no experience in custom chip design beyond a few small gate arrays designed for the BBC micro.

Into this apparently impossible scenario, the papers on the Berkeley RISC I fell like a bolt from the blue. Here was a processor which had been designed by a few postgraduate students in under a year, yet was competitive with the leading commercial offerings. It was inherently simple, so there were no complex instructions to ruin the interrupt latency. It also came with supporting arguments that suggested it could point the way to the future, though technical merit, however well supported by academic argument, is no guarantee of commercial success.

The ARM, then, was born through a serendipitous combination of factors, and became the core component in Acorn's product line. Later, after a judicious modification of the acronym expansion to **Advanced RISC Machine**, it lent its name to the company formed to broaden its market beyond Acorn's product range. Despite the change of name, the architecture still remains close to the original Acorn design.

Scanned with CamScanner

## 2.2 Architectural inheritance

At the time the first ARM chip was designed, the only examples of RISC architectures were the Berkeley RISC I and II and the Stanford MIPS (which stands for **Microprocessor without Interlocking Pipeline Stages**), although some earlier machines such as the Digital PDP-8, the Cray-1 and the IBM 801, which predated the RISC concept, shared many of the characteristics which later came to be associated with RISCs.

Features used

The ARM architecture incorporated a number of features from the Berkeley RISC design, but a number of other features were rejected. Those that were used were:

- a load-store architecture;
- fixed-length 32-bit instructions;
- 3-address instruction formats.

Features rejected

The features that were employed on the Berkeley RISC designs which were rejected by the ARM designers were:

- Register windows.

  The register banks on the Berkeley RISC processors incorporated a large number of registers, 32 of which were visible at any time. Procedure entry and exit instructions moved the visible 'window' to give each procedure access to new registers, thereby reducing the data traffic between the processor and memory resulting from register saving and restoring.

  The principal problem with register windows is the large chip area occupied by the large number of registers. This feature was therefore rejected on cost grounds, although the shadow registers used to handle exceptions on the ARM are not too different in concept.

  In the early days of RISC the register window mechanism was strongly associated with the RISC idea due to its inclusion in the Berkeley prototypes, but subsequently only the Sun SPARC architecture has adopted it in its original form

- Delayed branches.

  Branches cause pipelines problems since they interrupt the smooth flow of instructions. Most RISC processors ameliorate the problem by using delayed branches where the branch takes effect *after* the following instruction has executed.

  The problem with delayed branches is that they remove the atomicity of individual instructions. They work well on single issue pipelined processors, but they do not scale well to super-scalar implementations and can interact badly with branch prediction mechanisms.

Scanned with CamScanner

On the original ARM delayed branches were not used because they made exception handling more complex; in the long run this has turned out to be a good decision since it simplifies re-implementing the architecture with a different pipeline.

- Single-cycle execution of all instructions.

  Although the ARM executes most data processing instructions in a single clock cycle, many other instructions take multiple clock cycles.

  The rationale here was based on the observation that with a single memory for both data and instructions, even a simple load or store instruction requires at least two memory accesses (one for the instruction and one for the data). Therefore single cycle operation of all instructions is only possible with separate data and instruction memories, which were considered too expensive for the intended ARM application areas.

  Instead of single-cycle execution of all instructions, the ARM was designed to use the minimum number of cycles required for memory accesses. Where this was greater than one, the extra cycles were used, where possible, to do something useful, such as support auto-indexing addressing modes. This reduces the total number of ARM instructions required to perform any sequence of operations, improving performance and code density.

## implicity

An overriding concern of the original ARM design team was the need to keep the design simple. Before the first ARM chips, Acorn designers had experience only of gate arrays with complexities up to around 2,000 gates, so the full-custom CMOS design medium was approached with some respect. When venturing into unknown territory it is advisable to minimize those risks which are under your control, since this still leaves significant risks from those factors which are not well understood or are fundamentally not controllable.

The simplicity of the ARM may be more apparent in the hardware organization and implementation (described in Chapter 4) than it is in the instruction set architecture. From the programmer's perspective it is perhaps more visible as a conservatism in the ARM instruction set design which, while accepting the fundamental precepts of the RISC approach, is less radical than many subsequent RISC designs.

The combination of the simple hardware with an instruction set that is grounded in RISC ideas but retains a few key CISC features, and thereby achieves a significantly better code density than a pure RISC, has given the ARM its power-efficiency and its small core size.

Scanned with CamScanner

## 2.3 The ARM programmer's model

A processor's instruction set defines the operations that the programmer can use to change the state of the system incorporating the processor. This state usually comprises the values of the data items in the processor's visible registers and the system's memory. Each instruction can be viewed as performing a defined transformation from the state before the instruction is executed to the state after it has completed. Note that although a processor will typically have many invisible registers involved in executing an instruction, the values of these registers before and after the instruction is executed are not significant; only the values in the visible registers have any significance. The visible registers in an ARM processor are shown in Figure 2.1.

When writing user-level programs, only the 15 general-purpose 32-bit registers (r0 to r14), the program counter (r15) and the current program status register (CPSR) need be considered. The remaining registers are used only for system-level programming and for handling exceptions (for example, interrupts).

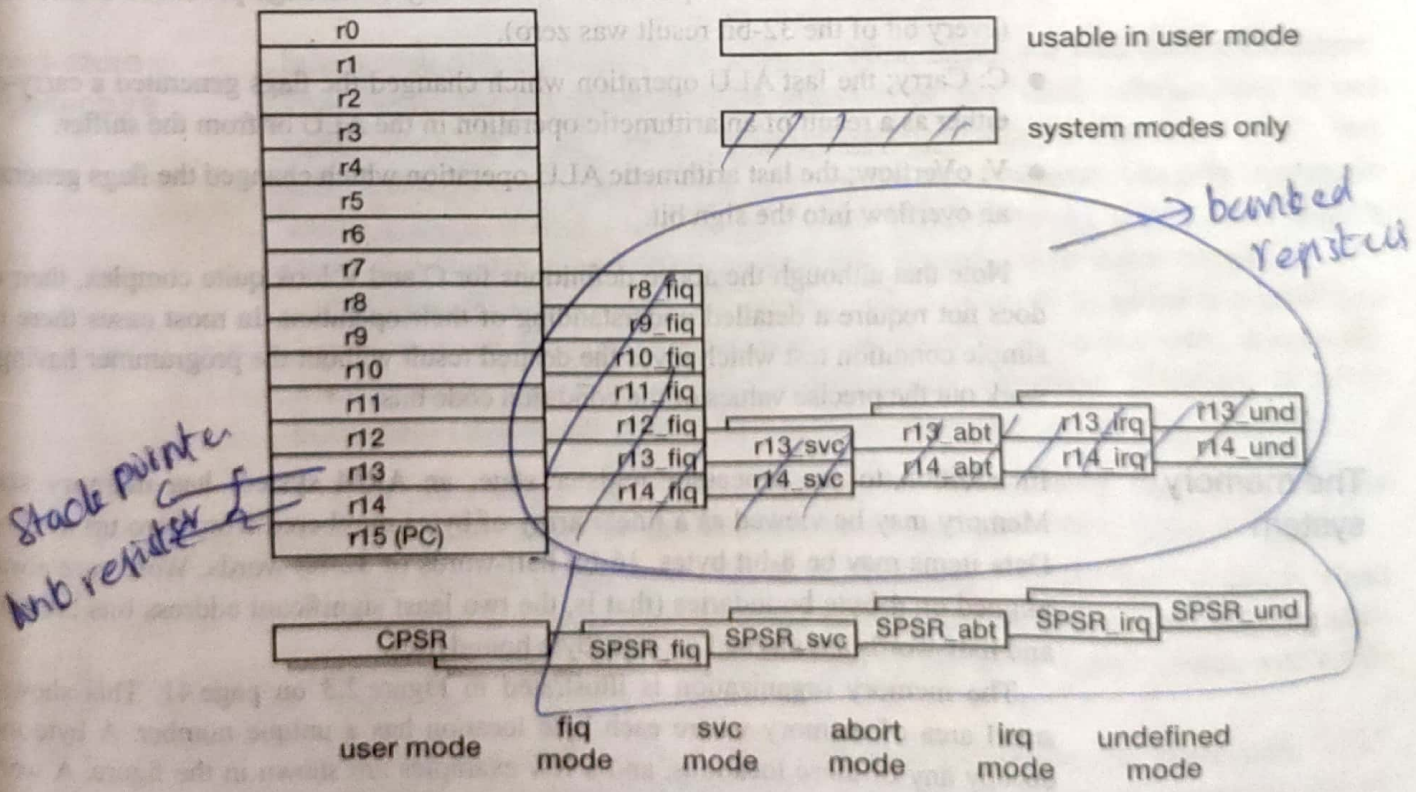


**Figure 2.1** ARM's visible registers.

Scanned with CamScanner

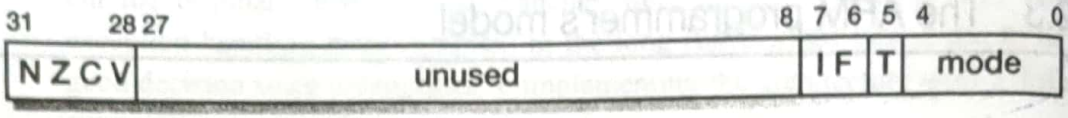


**Figure 2.2** ARM CPSR format.

## The Current Program Status Register (CPSR)

The CPSR is used in user-level programs to store the condition code bits. These bits are used, for example, to record the result of a comparison operation and to control whether or not a conditional branch is taken. The user-level programmer need not usually be concerned with how this register is configured, but for completeness the register is illustrated in Figure 2.2. The bits at the bottom of the register control the processor mode (see Section 5.1 on page 106), instruction set ('T', see Section 7.1 on page 189) and interrupt enables ('I' and 'F', see Section 5.2 on page 108) and are protected from change by the user-level program. The condition code flags are in the top four bits of the register and have the following meanings:

- N: Negative; the last ALU operation which changed the flags produced a negative result (the top bit of the 32-bit result was a one).
- Z: Zero; the last ALU operation which changed the flags produced a zero result (every bit of the 32-bit result was zero).
- C: Carry; the last ALU operation which changed the flags generated a carry-out, either as a result of an arithmetic operation in the ALU or from the shifter.
- V: oVerflow; the last arithmetic ALU operation which changed the flags generated an overflow into the sign bit.

Note that although the above definitions for C and V look quite complex, their use does not require a detailed understanding of their operation. In most cases there is a simple condition test which gives the desired result without the programmer having to work out the precise values of the condition code bits.

## The memory system

In addition to the processor register state, an ARM system has memory state. Memory may be viewed as a linear array of bytes numbered from zero up to $2^{32}-1$. Data items may be 8-bit bytes, 16-bit half-words or 32-bit words. Words are always aligned on 4-byte boundaries (that is, the two least significant address bits are zero) and half-words are aligned on even byte boundaries.

The memory organization is illustrated in Figure 2.3 on page 41. This shows a small area of memory where each byte location has a unique number. A byte may occupy any of these locations, and a few examples are shown in the figure. A word-sized data item must occupy a group of four byte locations starting at a byte address which is a multiple of four, and again the figure contains a couple of examples. Half-words occupy two byte locations starting at an even byte address.

Scanned with CamScanner

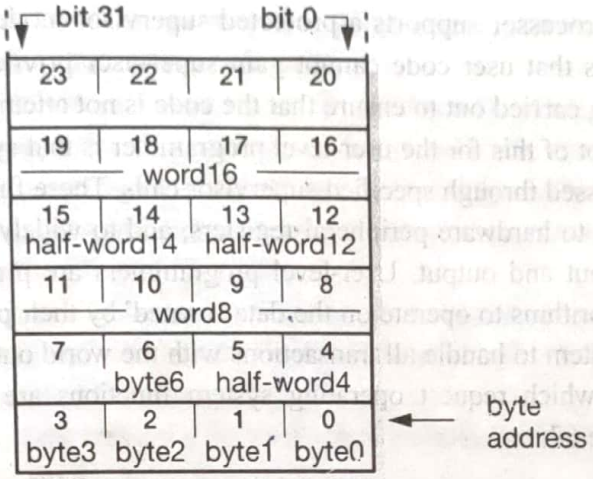


**Figure 2.3** ARM memory organization.

(This is the standard, 'little-endian', memory organization used by the ARM. ARM can also be configured to work with a 'big-endian' memory organization; we will return to this issue in Chapter 5.)

Load-store architecture

In common with most RISC processors, ARM employs a load-store architecture. This means that the instruction set will only process (add, subtract, and so on) values which are in registers (or specified directly within the instruction itself), and will always place the results of such processing into a register. The only operations which apply to memory state are ones which copy memory values into registers (load instructions) or copy register values into memory (store instructions).

CISC processors typically allow a value from memory to be added to a value in a register, and sometimes allow a value in a register to be added to a value in memory. ARM does not support such 'memory-to-memory' operations. Therefore all ARM instructions fall into one of the following three categories:

1. Data processing instructions. These use and change only register values. For example, an instruction can add two registers and place the result in a register.
2. Data transfer instructions. These copy memory values into registers (load instructions) or copy register values into memory (store instructions). An additional form, useful only in systems code, exchanges a memory value with a register value.
3. Control flow instructions. Normal instruction execution uses instructions stored at consecutive memory addresses. Control flow instructions cause execution to switch to a different address, either permanently (branch instructions) or saving a return address to resume the original sequence (branch and link instructions) or trapping into system code (supervisor calls).

Scanned with CamScanner

### Supervisor mode

The ARM processor supports a protected supervisor mode. The protection mechanism ensures that user code cannot gain supervisor privileges without appropriate checks being carried out to ensure that the code is not attempting illegal operations.

The upshot of this for the user-level programmer is that system-level functions can only be accessed through specified supervisor calls. These functions generally include any accesses to hardware peripheral registers, and to widely used operations such as character input and output. User-level programmers are principally concerned with devising algorithms to operate on the data 'owned' by their programs, and rely on the operating system to handle all transactions with the world outside their programs. The instructions which request operating system functions are covered in 'Supervisor calls' on page 67.

### The ARM instruction set

All ARM instructions are 32 bits wide (except the compressed 16-bit Thumb instructions which are described in Chapter 7) and are aligned on 4-byte boundaries in memory. Basic use of the instruction set is described in Chapter 3 and full details, including the binary instruction formats, are given in Chapter 5. The most notable features of the ARM instruction set are:

- The load-store architecture;
- 3-address data processing instructions (that is, the two source operand registers and the result register are all independently specified);
- conditional execution of every instruction;
- the inclusion of very powerful load and store multiple register instructions;
- the ability to perform a general shift operation and a general ALU operation in a single instruction that executes in a single clock cycle;
- open instruction set extension through the coprocessor instruction set, including adding new registers and data types to the programmer's model;
- a very dense 16-bit compressed representation of the instruction set in the Thumb architecture.

To those readers familiar with modern RISC instruction sets, the ARM instruction set may appear to have rather more formats than other commercial RISC processors. While this is certainly the case and it does lead to more complex instruction decoding, it also leads to higher code density. For the small embedded systems that most ARM processors are used in, this code density advantage outweighs the small performance penalty incurred by the decode complexity. Thumb code extends this advantage to give ARM better code density than most CISC processors.

### The I/O system

The ARM handles **I/O** (input/output) peripherals (such as disk controllers, network interfaces, and so on) as memory-mapped devices with interrupt support. The internal registers in these devices appear as addressable locations within the ARM's

Scanned with CamScanner

memory map and may be read and written using the same (load-store) instructions as any other memory locations.

Peripherals may attract the processor's attention by making an interrupt request using either the normal interrupt (*IRQ*) or the fast interrupt (*FIQ*) input. Both interrupt inputs are level-sensitive and maskable. Normally most interrupt sources share the IRQ input, with just one or two time-critical sources connected to the higher-priority FIQ input.

Some systems may include direct memory access (**DMA**) hardware external to the processor to handle high-bandwidth I/O traffic. This is discussed further in Section 11.9 on page 312.

Interrupts are a form of *exception* and are handled as outlined below.

### ARM exceptions

The ARM architecture supports a range of interrupts, traps and supervisor calls, all grouped under the general heading of exceptions. The general way these are handled is the same in all cases:

1. The current state is saved by copying the PC into r14_*exc* and the CPSR into SPSR_*exc* (where *exc* stands for the exception type).
2. The processor operating mode is changed to the appropriate exception mode.
3. The PC is forced to a value between $00_{16}$ and $1C_{16}$, the particular value depending on the type of exception.

The instruction at the location the PC is forced to (the *vector address*) will usually contain a branch to the exception handler. The exception handler will use r13_*exc*, which will normally have been initialized to point to a dedicated stack in memory, to save some user registers for use as work registers.

The return to the user program is achieved by restoring the user registers and then using an instruction to restore the PC and the CPSR atomically. This may involve some adjustment of the PC value saved in r14_*exc* to compensate for the state of the pipeline when the exception arose. This is described in more detail in Section 5.2 on page 108.

## 2.4 ARM development tools

Software development for the ARM is supported by a coherent range of tools developed by ARM Limited, and there are also many third party and public domain tools available, such as an ARM back-end for the *gcc* C compiler.

Since the ARM is widely used as an embedded controller where the target hardware will not make a good environment for software development, the tools are intended for **cross-development** (that is, they run on a different architecture from the

Scanned with CamScanner