1.6 The Reduced Instruction Set Computer  
RISC  
architecture  
In 1980 Patterson and Ditzel published a paper entitled 'The Case for the Reduced Instruction Set Computer' (a full reference is given in the bibliography on page 410). In this seminal work they expounded the view that the optimal architecture for a single-chip processor need not be the same as the optimal architecture for a multi-chip processor. Their argument was subsequently supported by the results of a processor design project undertaken by a postgraduate class at Berkeley which incorporated a Reduced Instruction Set Computer (RISC) architecture. This design, the Berkeley RISC I, was much simpler than the commercial CISC processors of the day and had taken an order of magnitude less design effort to develop, but nevertheless delivered a very similar performance.  
The RISC I instruction set differed from the minicomputer-like CISC instruction sets used on commercial microprocessors in a number of ways. It had the following key features:  
• A fixed (32-bit) instruction size with few formats; CISC processors typically had variable length instruction sets with many formats.  
• A load-store architecture where instructions that process data operate only on registers and are separate from instructions that access memory; CISC processors typically allowed values in memory to be used as operands in data processing instructions.

• A large register bank of thirty-two 32-bit registers, all of which could be used for any purpose, to allow the load-store architecture to operate efficiently; CISC register sets were getting larger, but none was this large and most had different registers for different purposes (for example, the *data* and *address* registers on the Motorola MC68000).  
These differences greatly simplified the design of the processor and allowed the designers to implement the architecture using organizational features that contributed to the performance of the prototype devices:  
• Hard-wired instruction decode logic; CISC processors used large microcode ROMs to decode their instructions.  
• Pipelined execution; CISC processors allowed little, if any, overlap between consecutive instructions (though they do now).  
• Single-cycle execution; CISC processors typically took many clock cycles to complete a single instruction.  
By incorporating all these architectural and organizational changes at once, the Berkeley RISC microprocessor effectively escaped from the problem that haunts progress by incremental improvement, which is the risk of getting stuck in a local maximum of the performance function.  
Patterson and Ditzel argued that RISC offered three principal advantages:  
• A smaller die size.  
A simple processor should require fewer transistors and less silicon area. Therefore a whole CPU will fit on a chip at an earlier stage in process technology development, and once the technology has developed beyond the point where either CPU will fit on a chip, a RISC CPU leaves more die area free for performance-enhancing features such as cache memory, memory management functions, floating-point hardware, and so on.  
• A shorter development time.  
A simple processor should take less design effort and therefore have a lower design cost and be better matched to the process technology when it is launched (since process technology developments need be predicted over a shorter development period).  
• A higher performance.  
This is the tricky one! The previous two advantages are easy to accept, but in a world where higher performance had been sought through ever-increasing complexity, this was a bit hard to swallow.  
The argument goes something like this: smaller things have higher natural frequencies (insects flap their wings faster than small birds, small birds faster than large birds, and so on) so a simple processor ought to allow a high clock rate. So let's design our complex processor by starting with a simple one, then add complex instructions one at a time. When we add a complex instruction it will make some high-level function more efficient, but it will also slow the clock down a bit for all instructions. We can measure the overall benefit on typical programs, and when we do, all complex instructions make the program run slower. Hence we stick to the simple processor we started with.  
These arguments were backed up by experimental results and the prototype processors (the Berkeley RISC II came shortly after RISC I). The commercial processor companies were sceptical at first, but most new companies designing processors for their own purposes saw an opportunity to reduce development costs and get ahead of the game. These commercial RISC designs, of which the ARM was the first, showed that the idea worked, and since 1980 all new general-purpose processor architectures have embraced the concepts of the RISC to a greater or lesser degree.

Since the RISC is now well established in commercial use it is possible to look back and see more clearly what its contribution to the evolution of the microprocessor really was.  
Early RISCs achieved their performance through:  
• Pipelining.  
Pipelining is the simplest form of concurrency to implement in a processor and delivers around two to three times speed-up. A simple instruction set greatly simplifies the design of the pipeline.  
• A high clock rate with single-cycle execution.  
In 1980 standard semiconductor memories (DRAMs - Dynamic Random Access Memories) could operate at around 3 MHz for random accesses and at 6 MHz for sequential (page mode) accesses. The CISC microprocessors of the time could access memory at most at 2 MHz, so memory bandwidth was not being exploited to the full. RISC processors, being rather simpler, could be designed to operate at  
clock rates that would use all the available memory bandwidth.  
Neither of these properties is a feature of the architecture, but both depend on the architecture being simple enough to allow the implementation to incorporate it. RISC architectures succeeded because they were simple enough to enable the designers to exploit these organizational techniques. It was entirely feasible to implement a fixed-length instruction load-store architecture using microcode, multi-cycle execution and no pipeline, but such an implementation would exhibit no advantage  
over an off-the-shelf CISC. It was *not* possible, at that time, to implement a hard-wired, single-cycle execution pipelined CISC. But it is now!  
As footnotes to the above analysis, there are two aspects of the clock rate discussion  
that require further explanation:

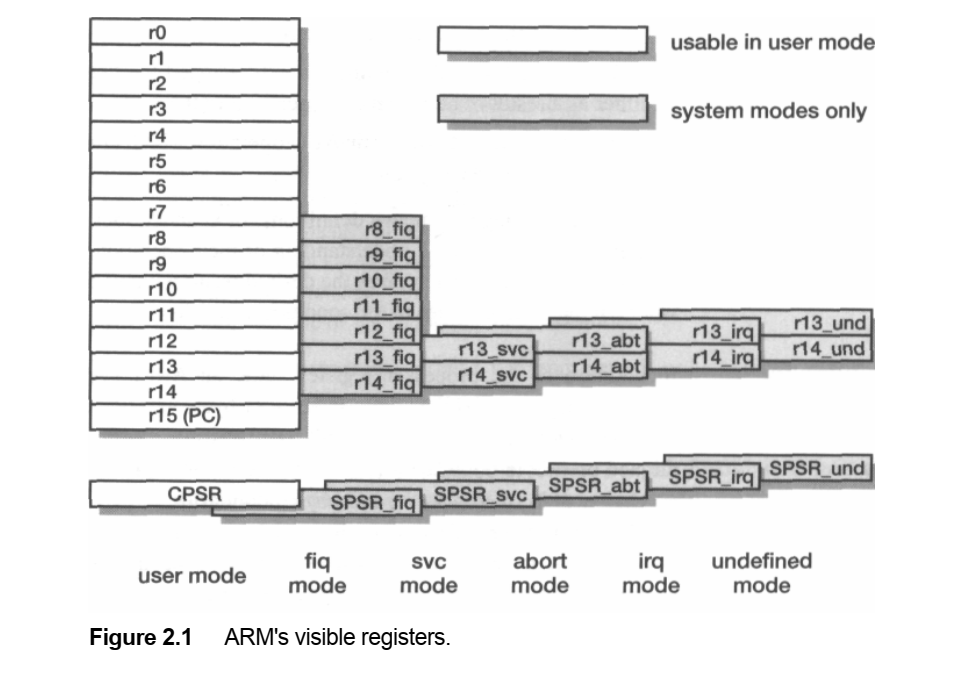
• 1980s CISC processors often had higher clock rates than the early RISCs, but they took several clock cycles to perform a single memory access, so they had a lower memory access rate. Beware of evaluating processors on their clock rate alone!  
• The mismatch between the CISC memory access rate and the available bandwidth appears to conflict with the comments in 'Complex Instruction Set Computers' on page 20 where microcode is justified in an early 1970s minicomputer on the grounds of the slow main memory speed relative to the processor speed. The resolution of the conflict lies in observing that in the intervening decade memory technology had become significantly faster while early CISC microprocessors were slower than typical minicomputer processors. This loss of processor speed was due to the necessity to switch from fast bipolar technologies to much slower NMOS technologies to achieve the logic density required to fit the complete processor onto a single chip.  
RISC processors have clearly won the performance battle and should cost less to design, so is a RISC all good news? With the passage of time, two drawbacks have come to light:  
• RISCs generally have poor code density compared with CISCs.  
• RISCs don't execute x86 code.  
The second of these is hard to fix, though PC emulation software is available for many RISC platforms. It is only a problem, however, if you want to build an IBM PC compatible; for other applications it can safely be ignored.  
The poor code density is a consequence of the fixed-length instruction set and is rather more serious for a wide range of applications. In the absence of a cache, poor code density leads to more main memory bandwidth being used for instruction fetching, resulting in a higher memory power consumption. When the processor incorporates an on-chip cache of a particular size, poor code density results in a smaller proportion of the working set being held in the cache at any time, increasing the cache miss rate, resulting in an even greater increase in the main memory bandwidth  
requirement and consequent power consumption.  
The ARM processor design is based on RISC principles, but for various reasons suffers less from poor code density than most other RISCs. Its code density is still, however, not as good as some CISC processors. Where code density is of prime importance, ARM Limited has incorporated a novel mechanism, called the Thumb architecture, into some versions of the ARM processor. The Thumb instruction set is a 16-bit compressed form of the original 32-bit ARM instruction set, and employs  
dynamic decompression hardware in the instruction pipeline. Thumb code density is better than that achieved by most CISC processors.

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

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.  
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.  
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.

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. 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.

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 r!4), 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)



A close-up of a computer screen

Description automatically generated

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.  
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 232-l.  
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.

A close-up of a computer

Description automatically generated

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 addi tional 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).

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.  
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 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 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.

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 *rl4\_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 0016 and 1C16, 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 rl3\_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 *rl4\_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.

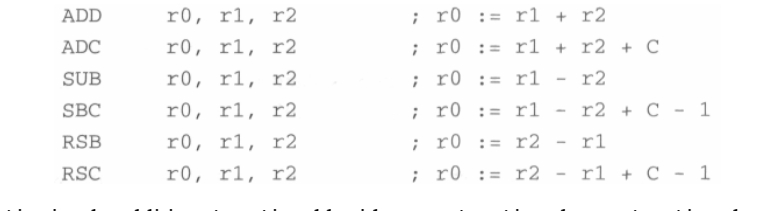
Data processing instructions  
ARM data processing instructions enable the programmer to perform arithmetic and logical operations on data values in registers. All other instructions just move data around and control the sequence of program execution, so the data processing instructions are the only instructions which modify data values. These instructions typically require two operands and produce a single result, though there are exceptions to both of these rules. A characteristic operation is to add two values together to produce a single result which is the sum.  
Here are some rules which apply to ARM data processing instructions:  
• All operands are 32 bits wide and come from registers or are specified as literals in the instruction itself.  
• The result, if there is one, is 32 bits wide and is placed in a register.  
(There is an exception here: long multiply instructions produce a 64-bit result;  
they are discussed in Section 5.8 on page 122.)  
• Each of the operand registers and the result register are independently specified in the instruction. That is, the ARM uses a '3-address' format for these instructions.

Simple register  
operands  
A typical ARM data processing instruction is written in assembly language as  
shown below: A black text on a white background

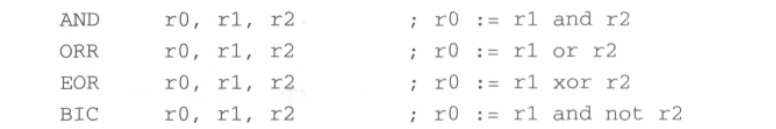
Description automatically generated

The semicolon in this line indicates that everything to the right of it is a comment and should be ignored by the assembler. Comments are put into the assembly source code to make reading and understanding it easier.  
This example simply takes the values in two registers (r1 and r2), adds them together, and places the result in a third register (r0). The values in the source registers are 32 bits wide and may be considered to be either unsigned integers or signed 2's-complement integers. The addition may produce a carry-out or, in the case of signed 2's-complement values, an internal overflow into the sign bit, but in  
either case this is ignored.  
Note that in writing the assembly language source code, care must be taken to write the operands in the correct order, which is result register first, then the first operand and lastly the second operand (though for commutative operations the order of the first and second operands is not significant when they are both registers). When this instruction is executed the only change to the system state is the  
value of the destination register r0 (and, optionally, the N, Z, C and V flags in the CPSR, as we shall see later)

The different instructions available in this form are listed below in their classes:  
• Arithmetic operations.  
These instructions perform binary arithmetic (addition, subtraction and reverse subtraction, which is subtraction with the operand order reversed) on two 32-bit operands. The operands may be unsigned or 2's-complement signed integers; the carry-in, when used, is the current value of the C bit in the CPSR



'ADD' is simple addition, 'ADC' is add with carry, 'SUB' is subtract, 'SBC' is subtract with carry, 'RSB' is reverse subtraction and 'RSC' reverse subtract with carry.  
• Bit-wise logical operations.  
These instructions perform the specified Boolean logic operation on each bit pair  
of the input operands, so in the first case *r0[i]:= r1[i]* AND *r2[i]* for each value  
of *i* from 0 to 31 inclusive, where *r0[i]* is the *i*th bit of r0.

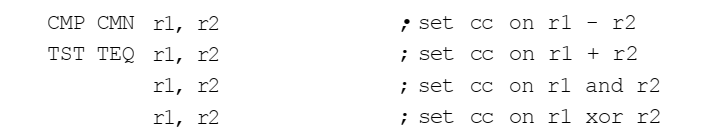


We have met AND, OR and XOR (here called EOR) logical operations at the  
hardware gate level in Section 1.2 on page 3; the final mnemonic, BIC, stands for  
'bit clear' where every ' 1' in the second operand clears the corresponding bit in  
the first. (The 'not' operation in the assembly language comment inverts each bit  
of the following operand.)  
• Register movement operations.  
These instructions ignore the first operand, which is omitted from the assembly  
language format, and simply move the second operand (possibly bit-wise  
inverted) to the destination



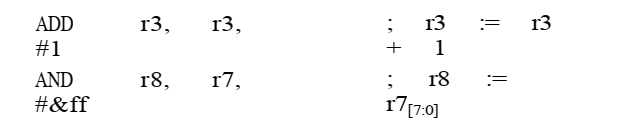
The 'MVN' mnemonic stands for 'move negated'; it leaves the result register set to  
the value obtained by inverting every bit in the source operand.

Comparison operations.  
These instructions do not produce a result (which is therefore omitted from the  
assembly language format) but just set the condition code bits (N, Z, C and V) in  
the CPSR according to the selected operation.



The mnemonics stand for 'compare' (CMP), 'compare negated' (CMN), '(bit) test'  
(TST) and 'test equal' (TEQ).

If, instead of adding two registers, we simply wish to add a constant to a register we  
can replace the second source operand with an immediate value, which is a literal  
constant, preceded by '#':



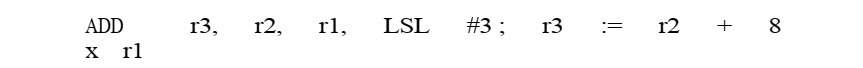
The first example also illustrates that although the 3-address format allows source  
and destination operands to be specified separately, they are not required to be distinct  
registers. The second example shows that the immediate value may be specified in  
hexadecimal (base 16) notation by putting '&' after the '#'.  
Since the immediate value is coded within the 32 bits of the instruction, it is not  
possible to enter every possible 32-bit value as an immediate. The values which can be  
entered correspond to any 32-bit binary number where all the binary ones fall within a  
group of eight adjacent bit positions on a 2-bit boundary. Most valid immediate values  
are given by:

A number with a arrow pointing to the right

Description automatically generated with medium confidence

where 0 < *n <* 12 . The assembler will also replace MOV with MVN, ADD with SUB, and  
so on, where this can bring the immediate within range.  
This may appear a complex constraint on the immediate values, but it does, in  
practice, cover all the most common cases such as a byte value at any of the four  
byte positions within a 32-bit word, any power of 2, and so on. In any case the  
assembler will report any value which is requested that it cannot encode.

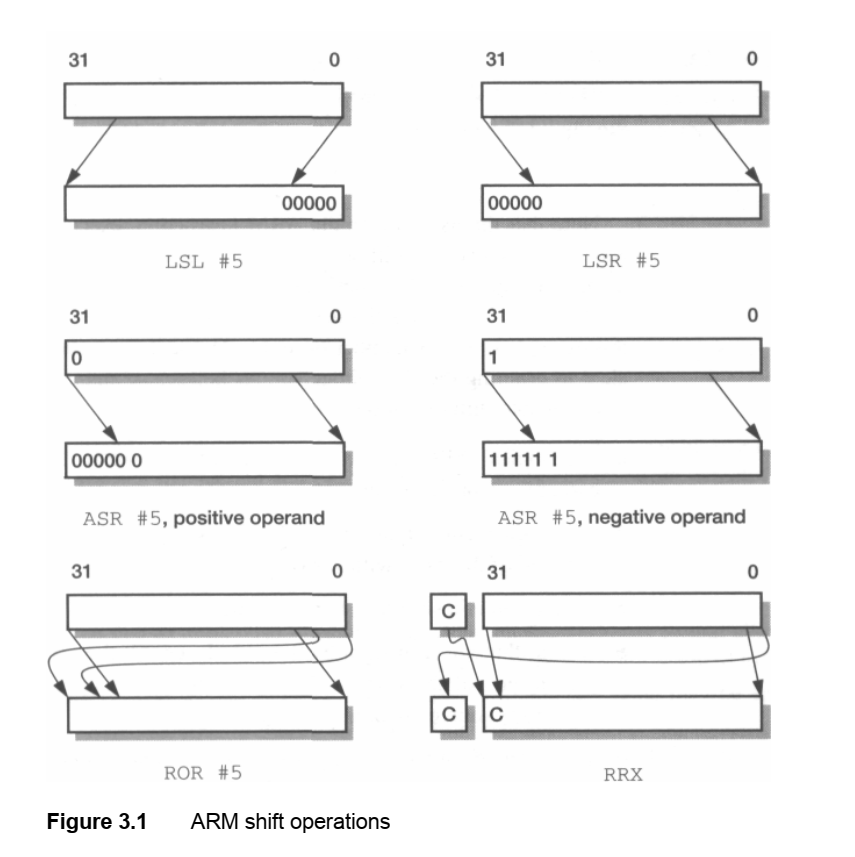
A third way to specify a data operation is similar to the first, but allows the second  
register operand to be subject to a shift operation before it is combined with the first  
operand. For example:



Note that this is still a single ARM instruction, executed in a single clock cycle.  
Most processors offer shift operations as separate instructions, but the ARM combines  
them with a general ALU operation in a single instruction.  
Here 'LSL' indicates 'logical shift left by the specified number of bits', which in  
this example is 3. Any number from 0 to 31 may be specified, though using 0 is equivalent to omitting the shift altogether. As before, '#' indicates an immediate quantity.  
The available shift operations are:  
• LSL: logical shift left by 0 to 31 places; fill the vacated bits at the least significant  
end of the word with zeros.  
• LSR: logical shift right by 0 to 32 places; fill the vacated bits at the most significant end of the word with zeros.  
• ASL: arithmetic shift left; this is a synonym for LSL.  
• ASR: arithmetic shift right by 0 to 32 places; fill the vacated bits at the most significant end of the word with zeros if the source operand was positive, or with ones if the source operand was negative.  
• ROR: rotate right by 0 to 32 places; the bits which fall off the least significant end of the word are used, in order, to fill the vacated bits at the most significant end of the word.  
• RRX: rotate right extended by 1 place; the vacated bit (bit 31) is filled with the old value of the C flag and the operand is shifted one place to the right. With appropriate use of the condition codes (see below) a 33-bit rotate of the operand and the C flag is performed.  
These shift operations are illustrated in Figure 3.1 on page 54. It is also possible to use a register value to specify the number of bits the second operand should be shifted by:



This is a 4-address instruction. Only the bottom eight bits of r2 are significant, but since shifts by more than 32 bits are not very useful this limitation is not important for most purposes.  
Any data processing instruction can set the condition codes (N, Z, C and V) if the  
programmer wishes it to. The comparison operations only set the condition codes,  
so there is no option with them, but for all other data processing instructions a



specific request must be made. At the assembly language level this request is indicated by adding an 's' to the opcode, standing for 'Set condition codes'. As an  
example, the following code performs a 64-bit addition of two numbers held in  
r0-r1 and r2-r3, using the C condition code flag to store the intermediate carry:

A close up of a text

Description automatically generated

Since the s opcode extension gives the programmer control over whether or not an  
instruction modifies the condition codes, the codes can be preserved over long instruction sequences when it is appropriate to do so.  
An arithmetic operation (which here includes CMP and CMN) sets all the flags  
according to the arithmetic result. A logical or move operation does not produce a  
meaningful value for C or V, so these operations set N and Z according to the result  
but preserve V, and either preserve C when there is no shift operation, or set C to the  
value of the last bit to fall off the end of the shift. This detail is not often significant.

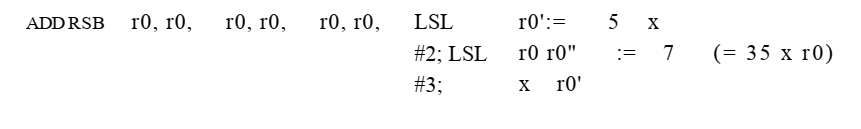
Use Of the We have already seen the C flag used as an input to an arithmetic data processing  
Condition codes instruction. However we have not yet seen the most important use of the condition  
codes, which is to control the program flow through the conditional branch instructions. These will be described in Section 3.3 on page 63.  
Multiplies A special form of the data processing instruction supports multiplication:



There are some important differences from the other arithmetic instructions:  
• Immediate second operands are not supported.  
• The result register must not be the same as the first source register.  
• If the ' s' bit is set the V flag is preserved (as for a logical instruction) and the C flag is rendered meaningless.  
Multiplying two 32-bit integers gives a 64-bit result, the least significant 32 bits of which are placed in the result register and the rest are ignored. This can be viewed as multiplication in modulo 232 arithmetic and gives the correct result whether the operands are viewed as signed or unsigned integers. (ARMs also support long multiply instructions which place the most significant 32 bits into a second result register; these are described in Section 5.8 on page 122.)  
An alternative form, subject to the same restrictions, adds the product to a running total. This is the multiply-accumulate instruction:



Multiplication by a constant can be implemented by loading the constant into a register and then using one of these instructions, but it is usually more efficient to use a short series of data processing instructions using shifts and adds or subtracts. For example, to multiply r0 by 35:



Data transfer instructions  
Data transfer instructions move data between ARM registers and memory. There are three basic forms of data transfer instruction in the ARM instruction set:  
• Single register load and store instructions.  
These instructions provide the most flexible way to transfer single data items between an ARM register and memory. The data item may be a byte, a 32-bit word, or a 16-bit half-word. (Older ARM chips may not support half-words.)

Multiple register load and store instructions.  
These instructions are less flexible than single register transfer instructions, but enable large quantities of data to be transferred more efficiently. They are used for procedure entry and exit, to save and restore workspace registers, and to copy blocks of data around memory.  
• Single register swap instructions.  
These instructions allow a value in a register to be exchanged with a value in memory, effectively doing both a load and a store operation in one instruction. They are little used in user-level programs, so they will not be discussed further in this section. Their principal use is to implement semaphores to ensure mutual exclusion on accesses to shared data structures in multi-processor systems, but don't worry if this explanation has little meaning for you at the moment.  
It is quite possible to write any program for the ARM using only the single register load and store instructions, but there are situations where the multiple register transfers are much more efficient, so the programmer should be familiar with them.  
Towards the end of Section 1.4 on page 14 there was a discussion of memory addressing mechanisms that are available to the processor instruction set designer.  
The ARM data transfer instructions are all based around register-indirect addressing, with modes that include base-plus-offset and base-plus-index addressing.  
Register-indirect addressing uses a value in one register (the **base** register) as a memory address and either **loads** the value from that address into another register or **stores** the value from another register into that memory address.  
These instructions are written in assembly language as follows:

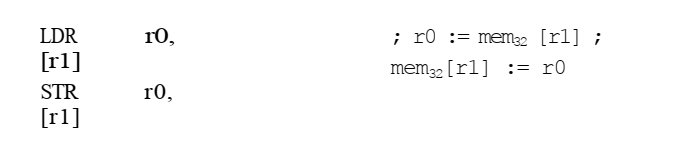


Other forms of addressing all build on this form, adding immediate or register offsets to the base address. In all cases it is necessary to have an ARM register loaded with an address which is near to the desired transfer address, so we will begin by looking at ways of getting memory addresses into a register.  
To load or store from or to a particular memory location, an ARM register must be initialized to contain the address of that location, or, in the case of single register transfer instructions, an address within 4 Kbytes of that location (the 4 Kbyte range will be explained later).  
If the location is close to the code being executed it is often possible to exploit the fact that the program counter, r15, is close to the desired address. A data processing instruction can be employed to add a small offset to r15, but calculating the appropriate offset may not be that straightforward. However, this is the sort of tricky calculation that assemblers are good at, and ARM assemblers have an inbuilt 'pseudo instruction', ADR, which makes this easy. A pseudo instruction looks like a normal instruction in the assembly source code but does not correspond directly to a particular ARM instruction. Instead, the assembler has a set of rules which enable it to select  
the most appropriate ARM instruction or short instruction sequence for the situation  
in which the pseudo instruction is used. (In fact, ADR is always assembled into a single  
ADD or SUB instruction.)  
As an example, consider a program which must copy data from TABLE1 to  
TABLE2, both of which are near to the code:

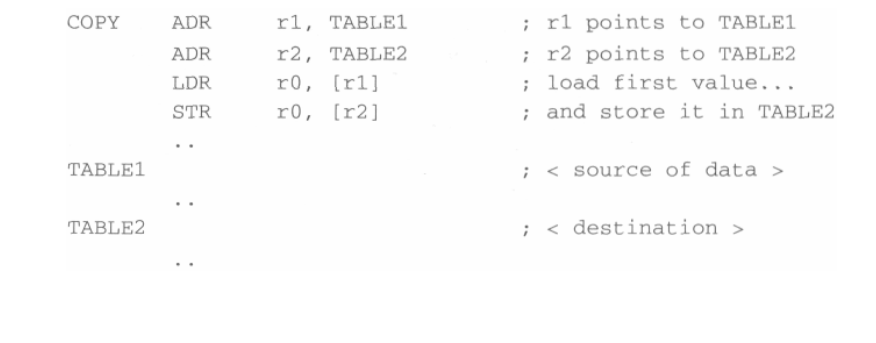
A close-up of a computer code

Description automatically generated

Here we have introduced **labels** (COPY, TABLE1 and TABLE2) which are simply  
names given to particular points in the assembly code. The first ADR pseudo instruction causes r1 to contain the address of the data that follows TABLE1; the second ADR  
likewise causes r2 to hold the address of the memory starting at TABLE2.  
Of course any ARM instruction can be used to compute the address of a data item  
in memory, but for the purposes of small programs the ADR pseudo instruction will do  
what we require.  
These instructions compute an address for the transfer using a base register, which  
should contain an address near to the target address, and an offset which may be  
another register or an immediate value.  
We have just seen the simplest form of these instructions, which does not use  
an offset:



The notation used here indicates that the data quantity is the 32-bit memory word  
addressed by r1. The word address in r1 should be aligned on a 4-byte boundary, so  
the two least significant bits of r1 should be zero. We can now copy the first word from  
one table to the other:



The load and store instructions are repeated until the required number of values has  
been copied into TABLE2, then the loop is exited. Control flow instructions are  
required to determine the loop exit; they will be introduced shortly.  
In the above examples the address offset from the base register was always an  
immediate value. It can equally be another register, optionally subject to a shift operation before being added to the base, but such forms of the instruction are less useful  
than the immediate offset form. They are described fully in Section 5.10 on page 125.  
As a final variation, the size of the data item which is transferred may be a single  
unsigned 8-bit byte instead of a 32-bit word. This option is selected by adding a letter  
B onto the opcode:



In this case the transfer address can have any alignment and is not restricted to a  
4-byte boundary, since bytes may be stored at any byte address. The loaded byte is  
placed in the bottom byte of r0 and the remaining bytes in r0 are filled with zeros.  
(All but the oldest ARM processors also support **signed** bytes, where the top bit of  
the byte indicates whether the value should be treated as positive or negative, and  
signed and unsigned 16-bit half-words; these variants will be described when we  
return to look at the instruction set in more detail in Section 5.11 on page 128.)

Where considerable quantities of data are to be transferred, it is preferable to move  
several registers at a time. These instructions allow any subset (or all) of the 16 registers to be transferred with a single instruction. The trade-off is that the available  
addressing modes are more restricted than with a single register transfer instruction.  
A simple example of this instruction class is:

A close-up of a test

Description automatically generated

Since the transferred data items are always 32-bit words, the base address (r1)  
should be word-aligned.  
The transfer list, within the curly brackets, may contain any or all of r0 to r15. The  
order of the registers within the list is insignificant and does not affect the order of  
transfer or the values in the registers after the instruction has executed. It is normal  
practice, however, to specify the registers in increasing order within the list.  
Note that including r15 in the list will cause a change in the control flow, since  
r15 is the PC. We will return to this case when we discuss control flow instructions  
and will not consider it further until then.  
The above example illustrates a common feature of all forms of these instructions: the lowest register is transferred to or from the lowest address, and then the  
other registers are transferred in order of register number to or from consecutive  
word addresses above the first. However there are several variations on how the  
first address is formed, and auto-indexing is also available (again by adding a '!'  
after the base register).  
The addressing variations stem from the fact that one use of these instructions is to  
implement stacks within memory. A stack is a form of last-in-first-out store which  
supports simple dynamic memory allocation, that is, memory allocation where the  
address to be used to store a data value is not known at the time the program is  
compiled or assembled. An example would be a recursive function, where the depth  
of recursion depends on the value of the argument. A stack is usually implemented  
as a linear data structure which grows up (an **ascending** stack) or down (a **descending** stack) memory as data is added to it and shrinks back as data is removed. A  
**stack pointer** holds the address of the current top of the stack, either by pointing to  
the last valid data item pushed onto the stack (a **full** stack), or by pointing to the  
vacant slot where the next data item will be placed (an **empty** stack).  
The above description suggests that there are four variations on a stack, representing all the combinations of ascending and descending full and empty stacks. The  
ARM multiple register transfer instructions support all four forms of stack:  
• Full ascending: the stack grows up through increasing memory addresses and the  
base register points to the highest address containing a valid item.

• Empty ascending: the stack grows up through increasing memory addresses and  
the base register points to the first empty location above the stack.  
• Full descending: the stack grows down through decreasing memory addresses  
and the base register points to the lowest address containing a valid item.  
• Empty descending: the stack grows down through decreasing memory addresses  
and the base register points to the first empty location below the stack.  
Although the stack view of multiple register transfer instructions is useful, there are  
occasions when a different view is easier to understand. For example, when these  
instructions are used to copy a block of data from one place in memory to another a  
mechanistic view of the addressing process is more useful. Therefore the ARM  
assembler supports two different views of the addressing mechanism, both of which  
map onto the same basic instructions, and which can be used interchangeably. The  
block copy view is based on whether the data is to be stored above or below the  
address held in the base register and whether the address incrementing or decrementing begins before or after storing the first value. The mapping between the two  
views depends on whether the operation is a load or a store, and is detailed in  
Table 3.1 on page 62.  
The block copy views are illustrated in Figure 3.2 on page 62, which shows how  
each variant stores three registers into memory and how the base register is modified if  
auto-indexing is enabled. The base register value before the instruction is r9, and after  
the auto-indexing it is r9'.  
To illustrate the use of these instructions, here are two instructions which copy  
eight words from the location r0 points to to the location r1 points to:  
LDMIA r0!, {r2-r9}  
STMIA r1, {r2-r9}  
After executing these instructions r0 has increased by 32 since the '!' causes it to  
auto-index across eight words, whereas r1 is unchanged. If r2 to r9 contained useful  
values, we could preserve them across this operation by pushing them onto a stack:

|  |  |  |  |
| --- | --- | --- | --- |
| STMFD  LDMIA  STMIA  LDMFD | r13!,  r0!,  r1,  r13!, | {r2-r9} {r2-r9} {r2-r9} {r2-r9} | save regs onto stack |
| ; restore from stack |  |  |  |

Here the 'FD' postfix on the first and last instructions signifies the full descending stack address mode as described earlier. Note that auto-indexing is almost  
always specified for stack operations in order to ensure that the stack pointer has a  
consistent behaviour.  
The load and store multiple register instructions are an efficient way to save and  
restore processor state and to move blocks of data around in memory. They save code  
space and operate up to four times faster than the equivalent sequence of single

A screenshot of a computer program

Description automatically generated

register load or store instructions (a factor of two due to improved sequential behaviour and another factor of nearly two due to the reduced instruction count). This significant advantage suggests that it is worth thinking carefully about how data is  
organized in memory in order to maximize the potential for using multiple register  
data transfer instructions to access it.  
These instructions are, perhaps, not pure 'RISC' since they cannot be executed in a  
single clock cycle even with separate instruction and data caches, but other RISC  
architectures are beginning to adopt multiple register transfer instructions in order to  
increase the data bandwidth between the processor's registers and the memory.  
On the other side of the equation, load and store multiple instructions are complex  
to implement, as we shall see later.  
The ARM multiple register transfer instructions are uniquely flexible in being able  
to transfer any subset of the 16 currently visible registers, and this feature is powerfully exploited by the ARM procedure call mechanism which is described in  
Section 6.8 on page 175.  
3.3 Control flow instructions  
This third category of instructions neither processes data nor moves it around; it  
simply determines which instructions get executed next.  
Branch  
instructions  
The most common way to switch program execution from one place to another is  
use the branch instruction:

A close-up of a white background

Description automatically generated  
  
The processor normally executes instructions sequentially, but when it reaches the  
branch instruction it proceeds directly to the instruction at LABEL instead of executing  
the instruction immediately after the branch. In this example LABEL comes after the  
branch instruction in the program, so the instructions in between are skipped. However, LABEL could equally well come before the branch, in which case the processor  
goes back to it and possibly repeats some instructions it has already executed.  
Sometimes you will want the processor to take a decision whether or not to branch.  
For example, to implement a loop a branch back to the start of the loop is required,  
but this branch should only be taken until the loop has been executed the required  
number of times, then the branch should be skipped.  
The mechanism used to control loop exit is conditional branching. Here the branch  
has a condition associated with it and it is only executed if the condition codes have  
the correct value. A typical loop control sequence might be:

A white background with black text

Description automatically generated

This example shows one sort of conditional branch, BNE, or 'branch if not equal'.  
There are many forms of the condition. All the forms are listed in Table 3.2, along  
with their normal interpretations. The pairs of conditions which are listed in the same  
row of the table (for instance BCC and BLO) are synonyms which result in identical  
binary code, but both are available because each makes the interpretation of the  
assembly source code easier in particular circumstances. Where the table refers to  
signed or unsigned comparisons this does not reflect a choice in the comparison  
instruction itself but supports alternative interpretations of the operands.

A table with text on it

Description automatically generated

A page of a paper

Description automatically generated

A close-up of a document

Description automatically generated

A close-up of a document

Description automatically generated A document with text and a few words

Description automatically generated with medium confidence

