# ARMs and architecture - Pt. 1

**Nicholas Cutler**

Having spent a series of articles looking at the implementation of software algorithms, this series will move on to examine the hardware in more detail. If the software could seem mysterious at times, the hardware must be even more so. After all, we are occasionally in a position of being able to examine a program ourselves (if the source code is made available), but we rarely have that luxury with the physical electronics.

The fundamental building blocks of all modern computers are transistors which act as electronic ‘switches’, controlling one circuit, depending on whether another is active or not. To explain how a computer works at this level would of course be impossible as the circuits are much too large; even the relatively simple StrongARM processor, typically found in the RISC PC, contains 100,000 transistors etched on a tiny wafer of silicon. Instead we must abstract away the details and focus on the broader picture, which is what is generally meant by the term computer architecture. Throughout the series I will illustrate the concepts with examples from ARM processors, hence my title.

Virtually all modern computers operate on the principle of the Von Neumann architecture, where a single memory stores both instructions and data. The processor reads instructions from memory and executes them sequentially. In the course of this, data may be read from memory, or the results written back. Linking the processor and memory together are two sets of electrical connections, known as the address and data busses. When reading data from memory, the processor sends the address of the desired location on the address bus, along with a read data signal. The memory responds by extracting the contents of the desired location and placing this on the data bus. These same basic ideas hold across all computers regardless of which processor they use. Inevitably there are some refinements and complications as a result of modern multi-core or parallel architectures, but the basic concept still holds.

The core of the computer is the Central Processing Unit (CPU), simply known as the processor. Despite the plethora of processor architectures which exist today, many of the various tricks which are employed in the quest for speed are hidden away internally so that a consistent model is presented to the programmer. The obvious difference between processors is the instruction set, but there are many other decisions to be made in the design of a processor: such as the number of operands taken by those instructions, where they come from, and the number of variables which can be stored internally.

All of these factors will result in various compromises between speed, power consumption, the size of the processor and the size of the software. Taken together, these various factors make up the programmer’s model of the processor, and are often grouped under the heading of Instruction Set Architecture (ISA).

Processors which share the same ISA are usually binary compatible, meaning that they can run the same programs. For example, both the ARM 2 and ARM 3 (used in the original Archimedes, and the A5000 respectively) are both instances of the same basic ISA, and software for the former would usually run on the latter. Although the ARM 3 did add a few extra instructions, the key differences were the addition of the cache, and shrinking the size of transistors on the chip; neither of which were immediately visible to the programmer, although they did have an impact on the processing speed.

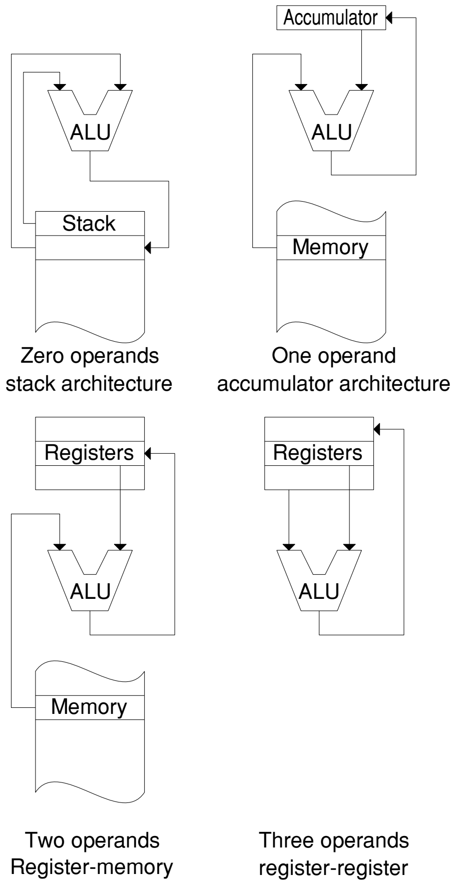
It is therefore sensible to begin with this model of the processor which can be seen as an interface between the software and hardware. All modern processors are esentially classified as having either complex or reduced instruction sets: the ARM along with the majority of other processor designs fall into the latter category, while the Intel x86 chips appear to buck the trend. In effect, all processor architectures designed since 1985 have followed the reduced instruction set model.

At first it seems counter-intuitive that the Reduced Instruction Set Computer (RISC) should have become so dominant. However, many of the additional instructions offered by the more complex instruction sets are, by their very nature, used so infrequently as to make little difference in practice. Furthermore, in order to accomodate the complex logic of these infrequently used instructions, the entire processor must run more slowly.

Equally, the terms ‘reduced’ and ‘complex’ need to be understood relatively. The 6502 processor used in the BBC Micro would seem to follow Complex Instruction Set Computer (CISC) principles, yet by reason of its age and being an 8-bit machine it is clearly a much simpler design. Similarly, the reduced instruction set ARM processor has some instructions which the 6502 does not: the multiply instruction being an obvious example.

Of course, while the comparison between RISC and CISC instruction sets is justifiable if we compare any modern RISC design (the ARM included) with, say, the Intel x86 series, the differences go beyond the instruction set.

For example, one further means of differentiating between processor architectures is in the number of operands accepted by each instruction. RISC processors, including the ARM, usually use a three operand format, while current CISC designs use a two operand format. Some earlier designs, notably the 6502 processor, used a one operand accumulator architecture. These options are summarised in the diagram below. In each case the Arithmetic and Logic Unit (ALU) is the part of the processor which computes the result of the specified operation:



Zero operands
stack architecture

One operand
accumulator architecture

Two operands
Register-memory

Three operands
register-register

In total there are four basic differences between RISC and CISC designs. The remainder of this article will explain these with examples from the ARM architecture, contrasted with the CISC Motorola 68k series.

To begin with, virtually all microprocessor designs have a limited internal memory capable of storing frequently used variables, constants and intermediate results. Such storage is known as the register file. Although this has a very limited capacity, it can be accessed quickly. This is important because processor speed has increased at a greater rate than memory, and accessing data from a register will be correspondingly faster than using the computer's memory. Having a large number of registers will speed up programs by reducing the number of memory accesses, while it is also easier to program when all important variables can be kept in registers.

The ARM architecture has 16 registers, and since these can be used as a data source or destination in any instruction, they can be described as general purpose registers. Admittedly one of these is used as the program counter (which holds the memory location of the next instruction), but, with some restrictions, it can still be used as an operand register by most instructions. Other RISC processors are similar, although most have even more registers, with 32 being a common number. In contrast, depending on what you count, the Intel x86 architecture has just eight registers, although not all of these are free to be used for general purpose data storage. Likewise the Motorola 68k series, another CISC design, but more straightforward than Intel's, has eight truly general purpose registers.

While it is often better to have more registers, there are trade-offs to be made: there are often limits to the number of variables which are needed at any one time before the final result of a computation will need to be written back to memory for long-term storage. Secondly, the contents of the processor's registers often need to be saved to memory before calling a subroutine, or when the operating system switches between tasks. In these cases a larger number of registers will actually slow things down. The ideal number of registers depends very much on the nature of the program, although modern compilers are often able to make good use of a large number of registers if they are available.

Since accessing operands from registers is much faster than from memory, typical RISC instructions are designed to take the two operands from registers and write the result back to a third register. Thus the instruction:

```
ADD R0,R1,R2
```

takes the numbers in registers 1 and 2, adds them together and writes the result back to register 0. In contrast, a single instruction on a CISC architecture can take either one, or sometimes both, operands from memory. So a similar instruction on the 68k series would be:

```
ADD D0,(A0)
```

which reads one operand from memory at the address contained in A0, adds it to the contents of D0 and writes the result back to the same register. This illustrates the use of indirect addressing (where one of the operands is kept in memory), the distinction between data and address registers, and the two operand format where the destination is the same as the first source operand. The 8-bit 6502 processor took this even further as it used a one operand format where the accumulator was the implicit destination.

Of course, both architectures need to have load instructions to fetch the initial data from memory, and store instructions to save the final results. The main difference is that on a RISC architecture these are the only instructions which can access memory. Following on from the simple ADD instruction above, we consider a more complete example to add two numbers from a data structure stored in memory and write the results back. Firstly the ARM version:

```
LDR R1,[R3]
LDR R2,[R3,#4]
ADD R1,R1,R2
STR R1,[R3,#8]
```

and the CISC version:

```
MOVE (A3),D1
ADD D1,(4,A3)
MOVE D1,(8,A3)
```

Note that the ARM version requires one extra instruction and a temporary register to load the second operand from memory before performing the addition. These simple examples additionally illustrate the use indirect addressing with an offset, which would, for example, correspond to using a particular field from a data structure.

Many of the available addressing modes are designed to support the needs of higher level programming languages. Hence direct addressing corresponds to using a variable; indirect addressing is equivalent to a memory pointer; offset addressing corresponds to a field from a data structure. Another important operation in high-level languages is to read a single element of an array. Thus, both RISC and CISC designs offer indexed addressing modes to support this, using another register as the offset. On the ARM this is written:

```
LDR R0,[R1,R2]
```

and on the CISC:

```
MOVE (0,A1,A2),D0
```

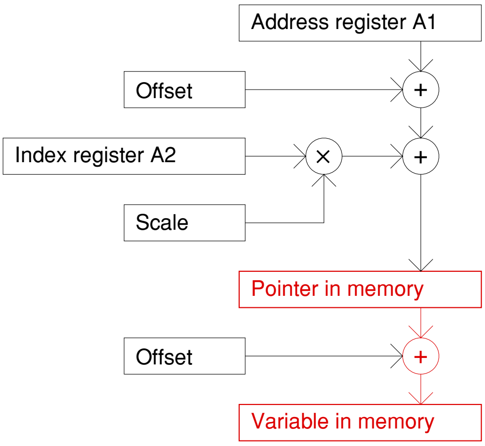
which means load register 0 from the memory location specified by register 1 plus the index in register 2. However, CISC processors usually go further and support double indirection where the address of the final operand is itself the content of another pointer plus an index. This is like using an array of pointers, or a specific field from a structure specified by a pointer. The most complicated addressing modes add a second offset to access a single element from a two dimensional array. This is written as:

```
MOVE ([bd,A0,A1*scale],od),D2
```

There is, of course, no single equivalent on a RISC processor. Instead, the closest equivalent on the ARM is the following sequence of instructions:

```
MOV    R2,#bd
ADD    R2,R2,R1,LSL #scale
LDR    R2,[R0,R2]
LDR    R2,[R2,#od]
```

This uses register 2 as a temporary before finally overwriting it with the data from memory. It also uses the unique feature on the ARM which combines a shift to scale the index with the add. Other RISC designs would instead require one further instruction and temporary register. The process of calculating the address of the memory operand can be shown more clearly in the diagram below:



The next major difference between the two classes of design is the obvious one suggested by the name: the actual set of instructions offered. In contrast to high-level languages which are able to represent relatively complex constructions in a natural and succinct notation, assembly language is usually verbose by comparison. The RISC processor takes this further as it only offers the bare essentials: the data transfer, control, logical and simple arithmetic operations which are the core of any program. Each instruction typically performs a single operation which can be easily implemented in the digital logic circuits of the microprocessor. The integer division operation might seem basic enough, but it is difficult to implement efficiently in hardware, and the ARM isn't the only RISC processor to omit it.

In contrast, the CISC designs include extra instructions to facilitate the implementation of high-level languages and operating systems. Although these extra instructions can be synthesised from simpler ones, their inclusion can make programs shorter and the compiler's task easier.

One such operation is the "compare and swap" instruction on the 68k series. This compares the destination with the first source operand. If they are equal then the destination is set to the second source operand, otherwise the first source is updated with the destination. So:

```
CAS D0,D1,D2
```

is equivalent to the following ARM instructions:

```
CMP R2,R0
MOVEQ R2,R1
MOVNE R0,R2
```

Another example is the "conditional decrement and branch" instruction. If the specified condition is false then the operand register is decremented. If this is not equal to minus one then processor follows the specified branch, otherwise the program continues with the next instruction. Thus the single instruction:

```
DBGT D0,loop
```

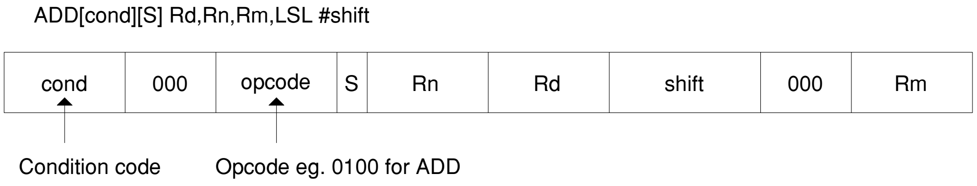
is equivalent to the following two ARM instructions:

```
SUBSGT R0,R0,#1
BPL loop
```

In effect this repeatedly executes a block of instructions while a given condition is true up to a given limit on the number of iterations. In both cases the condition 'GT' (for greater than) can be changed for another as required. In this case, the ability of the ARM to conditionally execute most instructions saves the extra comparison and short branch which most other RISC processors would need. In this case, it is worth mentioning that it is often more natural to count down to zero, and in ARM assembly language you will often see the instruction `BNE` instead.

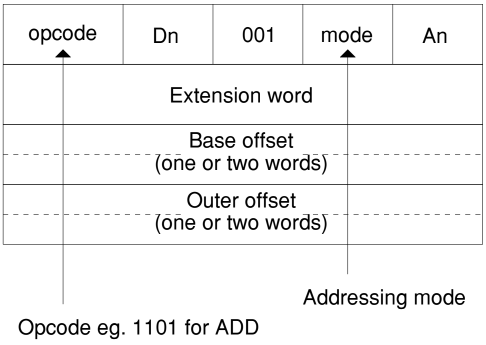
The final difference between RISC and CISC processors is largely invisible to the programmer, and instead affects the binary representation of the instructions in memory. For the RISC processor each instruction has a fixed length representation, which, like the ARM, is usually a single 32 bit word. Packed into this field are the type of operation, the registers affected and any constant data. A fixed length instruction format is convenient and enables the entire instruction to be fetched in advance in a single memory cycle. The format of a typical ARM instruction is shown below:

Conversely, with only 32 bits available, it restricts the range of instructions, addressing



modes and constants which can be represented. In particular the arbitary constants and addresses required by the indirect or indexed addressing modes on a CISC processor mean that instructions often need to be longer than 32 bits. In fact, they are of a variable length. This enables the simpler instructions to be represented concisely, while still allowing for flexibility in the more complex ones. This technique can make programs shorter, but fetching an instruction from memory can take several stages which slows the processor down. Instructions on the Intel x86 architecture are up to 18 bytes long, and on the 68k series can occupy up to 22 bytes. A typical instruction, again an add, with one operand being used from memory, on the latter processor are shown below:

ADD <effective address>,Dn



To summarise, this article has examined the main differences between RISC and CISC processors. The design philosophy behind the former is to concentrate on the basic operations which make up the majority of programs, and to execute them quickly. In contrast, the CISC processors often offer extra instructions for more complex but infrequently used operations. This, together with the increased flexibility of being able to use operands directly from memory, often simplifies the task of compiling a high-level language, and reduces the size of the final program. However, the RISC instructions will be much faster as the processor does not need to wait for data to be fetched from memory. Although the trade-offs vary with the application, in most real situations, RISC processors are usually faster as the speed gained from is greater than the cost of any extra instructions.

Further parts to this series will look in greater depth at some of the unique features of the ARM processor; some of the tricks employed by modern processors in the quest for greater speed; and the architecture of the memory system.