ARMs and architecture - Pt. 2

Nicholas Cutler

The previous part of this series introduced the programmer's model of the processor at the instruction set level. This concentrated on the differences between the Reduced Instruction Set ARM, and the Complex Instruction Set designs like the Intel x86 series.

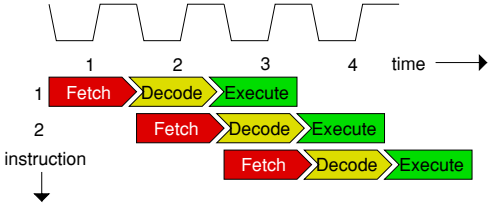
Similarly, the first article also introduced the Von Neumann architecture with a single memory for programs and data, together with the sequential cycle of fetching an instruction from memory and executing it. Despite its origins in the earliest computers, this remains a remarkably good description of how a microprocessor operates. However, modern computers employ a number of tricks in order to run faster, and a lot of effort has been made to present a consistent model to the programmer.

While it is not essential to understand computers at this lower level, a knowledge of principles involved is still useful. For the assembly language programmer, and the compiler writer, it will help them to optimise their programs; while for the end user it will help them to appreciate the differences between otherwise identical machines.

Recall from the previous part that it can often take several cycles of the processor's clock to complete one instruction. Even the relatively simple RISC instructions cannot complete in a single cycle because of the overhead incurred in fetching the instruction from memory. However, most modern processors are able to give the illusion of executing one or more instructions in each single clock cycle. One obvious way to achieve this is to slow the clock down sufficiently to give each instruction long enough to complete. This is an unattractive option as it makes the whole device operate at the speed of the slowest instruction.

Instead, issuing one instruction per cycle at high frequencies, effectively means that more work must be done on each cycle. Executing a single instruction can be split into several stages: firstly it is fetched from memory, secondly it is decoded, before finally the result is computed. Each of these steps uses a different part of the processor, and it is therefore possible to overlap several instructions: while one is being fetched the previous one can be decoded, and so on.

To explain the idea using familiar concepts, consider the analogy of a cafeteria service. You enter the cafeteria and collect a plate, then give your order, wait for the food to be served on to your plate and, finally, pay for the meal. Although the whole process may take a minute, one customer enters and another leaves every 15 seconds. A broadly similar idea applies to instructions in the computer's processor. Although each individual operation takes just as long, the entire program runs faster. This concept, known as pipelining, helps to achieve the goal of issuing one instruction in each cycle. Beyond that, however, it also makes more efficient use of the resources on the processor, by allowing the different phases to overlap. The idea is shown in the diagram below:



Although this concept will be developed further, it is worth pausing to consider one main problem.

Hitherto, it has been assumed that the instructions are arranged sequentially in the computer's memory. While this is true for most of the time, it takes no account of branch instructions, either to enter a subroutine, in loops, or conditional statements. In each case, there is no way of knowing at the point of fetching an instruction whether the branch will be taken, and therefore where the next instruction will be found. The easiest solution is to wait until the branch reaches the final stage, and then, if necessary, discard the pending operations. This effectively stalls the processor while the instructions are fetched from the new location and decoded. Such delays are known as control hazards.

Another possible solution to this problem is for programmers and compiler designers to avoid branch instructions wherever possible. This isn't always easy, but the ARM architecture has one unique facility to help, namely conditional execution. Instead of jumping to a different part of the program if a certain condition is true, individual instructions can instead be ignored if the condition is false. Although one cycle has been wasted in deciding to ignore the instruction, it has the advantage that there is no disruption to the pipeline. To illustrate this concept, consider a simple conditional statement in a high level language:

```
if a=8 then a:=0 else a:=a+1
```

This would normally compile to something like:

```
CMP R0,#8
BNE if_false
MOV R0,#0
B done
if_false:
ADD R0,R0,#1
done:
```

However, with conditional execution the branches can be eliminated altogether giving something like:

```
CMP R0,#8
MOVEQ R0,#0
ADDNE R0,R0,#1
```

Notice that the condition codes are the opposite way round: instead of branching if the condition is false, the instruction is executed only if it is true. This has allowed us to eliminate the two branches entirely, and shortened the program in the process. Similar branches are often used when repeating a sequence of instructions. For example consider the loop:

```
for i:=1 to 4 do a[i]:=i
```

Which might compile to:

```
MOV R0,#4
loop:
STR R0,[R1,R0,LSL #2]
SUBS R0,R0,#1
BNE loop
```

In this case, we can't use conditional execution to eliminate the loop, but by counting down to zero and testing this condition it is possible to eliminate an explicit comparison. However, by repeating the body of the loop (the store and subtract instructions) sufficient times, the branch can, once again, be eliminated. This technique is known as unrolling the loop. While it does speed up the program, it also makes it longer, so it may not be acceptable in all cases.

For those cases where it is not possible to eliminate a branch (which account for about 10% of all instructions), it is natural to ask if it is possible to reduce the speed penalty? In fact there are a number of possible solutions. Firstly, notice that, like the above examples, branches are often quite short, that is they jump forwards or backwards by only a few instructions. Also, recall that some of the delay is caused by reading the new instructions from the relatively slow memory.

One possibility is to introduce a small, fast, memory as a buffer between the processor and the large, slow, main memory. This buffer is known as a cache, and the idea is quite common in computing wherever a fast process has to interface with a much slower one. Similar examples are the operating system cacheing frequently used files in memory to speed up reading data from disc, or a web browser keeping local copies of recent pages to save having to fetch them from the network.

The ARM 3 processor was the first of the series to employ a cache, and because it only needed to store recently used parts of the program, it didn't need to be very large. Just 4kb was sufficient in the ARM 3, and even now caches of only 16 to 64kb are common. The small size means that the cache can be located on the same chip as the processor, while also enabling the use of faster memory to reduce the speed penalty. Thus, although branches will still disrupt the pipeline, there is a good chance that the next instruction will be in the cache, and can be fetched quickly.

In conjunction with the cache memory, it is also possible to mitigate effect of branches by predicting and anticipating them. One simple way of doing this is to examine the branch target, which can be done during the decode stage. If the branch leads backwards to an earlier part of the program then, like the example above, this is usually a loop. In most cases, such a branch would succeed. Conversely, a jump forwards to a later part of the program is likely to be a conditional statement. It is assumed that, in general, such branches are not taken. With some knowledge of the control flow in the program, it is possible to start fetching the instructions speculatively in advance of their being needed.

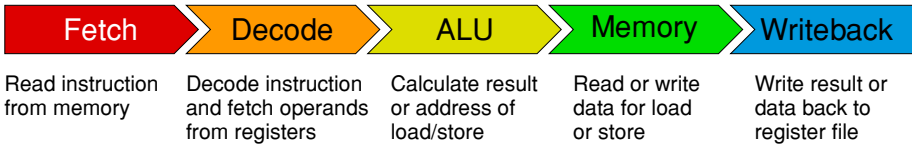
In reality, branch prediction schemes are often more sophisticated than this as the processor tries to take the previous behavior of the program into account. Of course, if the prediction is incorrect then the pipeline will stall as before, but this penalty is more than cancelled out by the gain from correctly predicted branches.

Caches also help to speed up other memory accesses, whether they are for instructions or data. Thus, load and store instructions don't have to wait for the main memory and can complete faster.

This is important, because while the three-stage pipeline was simple to implement and enabled the efficiency for which the ARM architecture is reknown, it isn't perfect. Processors aimed at high-performance applications often used a longer five-stage pipeline, which scheme was also adopted in the StrongARM. However, this means that the potential for delays caused by memory accesses become more significant.

With five stages, it is possible to subdivide the final execution of an instruction, allowing load and store operations an extra cycle to access memory, and a final stage to write the result back to the register file. The full five stages in

The five stage RISC pipeline:



this scheme are:

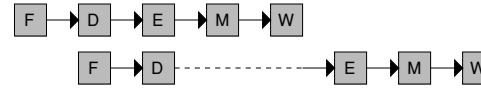
1. Fetch the instruction from memory.
2. Decode the instruction and get operands from registers.
3. Calculate the result or compute the address of a load or store.
4. Fetch data from memory if necessary.
5. Write the data or result back to the register file.

Implementing such a pipeline introduces a conflict between the first (fetch) and fourth (memory) stages: both need to access the memory at the same time. With cache memory on the processor, it is possible to resolve this by effectively separating instruction and data memory. This allows two accesses to different parts of memory (one to fetch an instruction, the other for data) to happen at the same time.

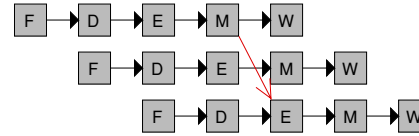
In the case of the five-stage pipeline, there is another problem when one instruction needs data from a register before the previous instruction has been able to write its result back. For example, consider the following sequence:

```
LDR R0,[R1,R2]
ADD R0,R0,R3
ADD R2,R2,#4
```

Because the memory access happens at a late stage in the pipeline, the first add instruction needs the data from R0 before it has been fetched from memory and written to the register file. In the worst case the pipeline is stalled for two cycles until the load operation completes. This situation is known as a data hazard. One partial solution is to change the order of instructions in the program to allow one operation to complete before another needs to access the result. In the above example, we can re-order the two add instructions (good compilers will try to do this automatically) but this still leaves a delay of one cycle before R0 can be used. A more satisfactory solution is to pass the data directly from the output of the memory stage to the input of the execute stage of the following instruction. This facility known as forwarding saves one further cycle, which, together with reordering instructions, can eliminate the pipeline stall.



The add instruction stalls waiting for the load to complete.



With data forwarding (indicated by the red arrow) there is no delay.

Although it has been possible to reduce the impact of control and data hazards with caches and data forwarding, a problem remains with intrinsically slow operations like multiplication. This is a sufficiently common operation to merit being implemented in hardware, even in RISC processors, yet it takes several cycles to execute. While it is possible to speed up such instructions at the expense of large amounts of logic and power, it is unrealistic to complete a multiplication in a single cycle.

Of course, a good programmer will avoid unnecessary work, and the ability of the ARM to combine a shift with other data processing operations helps. Multiplication by small constants can often be achieved by a few simple instructions. For example:

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

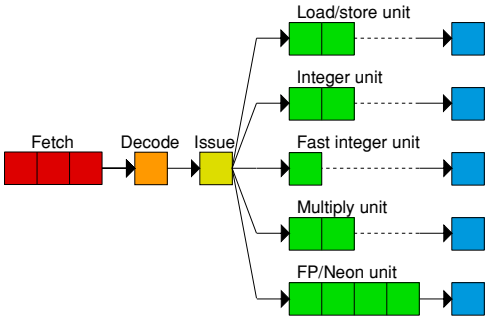
will multiply by five (x+x*4). With an extra addition, multiplication by ten is possible:

```
ADD R1,R0,R0,LSL #2
ADD R0,R1,R1
```

As elegant as such tricks may be, the need to

accommodate a general purpose multiplication instruction remains. The simple answer is to allow such instructions to be cycled through the execute stage as many times as required. This effectively means that the following instructions are held up until the slow operation has finished. Another possible solution is to add extra pipeline stages only for slow instructions. This doesn't help much either, because the following instructions cannot reach the final stage until the slow one has finished. This is an example of a structural hazard; a delay caused by the internal design of the pipeline.

The ultimate solution to such problems is to provide an extra execution unit for any slow instructions, allowing the regular ALU to be used for the numerous fast instructions (very often multiplications account for only about 1% of instructions). In effect this allows two instructions to execute in parallel. Of course, the other stages of the pipeline, like the fetch and decode, are still shared, but it potentially allows the processor to keep working even while waiting for slow running instructions to complete. While this is not the same as a true multi-processor, which can fetch and execute instructions from several streams at once, it is nevertheless a huge improvement over the Von Neumann model.



The pipeline of the Cortex-A9: the extra fetch stages accommodate branch prediction. The issue stage allocates instructions to the appropriate unit.

This concept is not limited to multiply instructions, and modern processors often include extra functional units for loads and stores and floating point instructions too. For example, the ARM Cortex-A9 used in the Pandaboard, offers: a load/store unit, two integer units, a dedicated multiplier and a floating-point unit. Superscalar processors are, therefore, able to exploit implicit parallelism at the level of individual instructions, although there are limits to what they can achieve. In the example above, it is possible to execute two instructions together: one multiply and one simpler instruction. However, it is not always possible to execute any two consecutive instructions together, as this relies on instructions being independent of each other. If one operand is the result of a preceeding instruction, then again everything must wait for that result to be computed. In the case of a multiply operation this could still mean a delay of several cycles. The only solution to this problem is for the compiler to identify data dependencies between operations and re-order them, as far as possible, to minimise delays in the pipeline. The Intel Itanium architecture relied on the compiler being able to explicitly allocate instructions into 'bundles' which can be executed together.

In the interests of completeness, it is worth mentioning that some processors take the superscalar concept further. By looking ahead in the program to identify those instructions without data dependencies, it is possible to execute them out of order. Although this can exploit more implicit parallelism without the need for support from the compiler, it does require a lot of power intensive control logic in the processor. Therefore, while this can be a useful technique in high-performance applications, there are still limits on what can be achieved.

The application of the techniques discussed above has been considerably simplified by the

widespread adoption of RISC architectures. The early pipeline stages (fetch and decode) are easier to implement with instructions of a fixed length. Equally, the simpler nature of the operations makes the execution easier to pipeline, while it also allows a good compiler more flexibility in scheduling instructions to avoid stalls. Finally, limiting memory access to the load and store instructions reduces data hazards and makes data cache management simpler. As a result, pipelined versions of CISC processors usually translate the instructions to simpler RISC-like operations and schedule those. The Intel x86 architecture is the best known example of this.

In conclusion, the microprocessor has come a long way since the 8-bit devices in the first microcomputers. Until recently, the number of transistors on a single chip has increased exponentially. This growth has made possible 32 and even 64-bit devices with long pipelines operating at high frequencies, while still being able to issue an instruction in each cycle. Separate caches for instructions and data reduce conflicts and provide an interface to the relatively slow main memory. Techniques like branch prediction and data forwarding reduce pipeline hazards, while superscalar processors take advantage of implicit parallelism.

This part of the series has concluded the discussion of processor architecture, looking at the internal features found on nearly all modern devices. There remains one further component of any practical computer, namely the memory. The third and final part of the series will look at memory hierarchies in more detail.