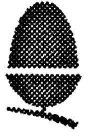
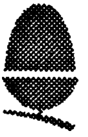
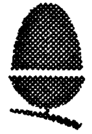


RISC Technology Module

Contents



This module gives an introduction to the RISC processor technology which lies behind the phenomenal speed and power of the Acorn Archimedes/A3000. It charts the development of the conventional microprocessor over the years, points out where this development has caused problems, and shows how RISC technology solves them. It also provides an introduction to the other Acorn custom chips which are used in Archimedes/A3000, and carry out almost all of the tasks needed to transform a processor into a usable computer.



Over the last twenty years, we have seen a massive increase in the size, complexity and power of microprocessors. The earliest microprocessors, such as the Intel 4004, appeared in the early 1970s. They were very limited, compared with modern designs, by two things: the width of their data busses and the complexity of the instructions which they could execute.

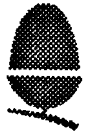
Data Bus Width

The 'width' of a processor's data bus determines the amount of data which it can process in a single instruction. The earliest processors had 4-bit data busses, which meant that each operation could work on numbers in the range 0-15, or a very limited range of memory addresses. Of course, this isn't enough for real-world problems: very rarely will a bank balance be within 0.015 pence of zero, or will a book be less than 15 characters long.

In order to deal with problems like this, our 4-bit micro had to perform multiple-precision arithmetic. In order to add up a couple of numbers, it would perform long addition in very much the same way as we do, adding pairs of digits and keeping track of carries. Each addition takes a fixed length of time (around 2 microseconds), and in order to cope with real-life numbers it would have to perform several of them.

Instructions

Another problem with these early processors was that they could only understand simple instructions. If you wanted to multiply a pair of numbers together, you had to write a little program to do it, again in a very similar way to how we perform long multiplication. As you may remember, long multiplication takes a human longer than long addition, and the same holds true for micros - it may take several hundred instructions to multiply two numbers together on a simple processor.



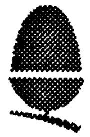
There are a number of approaches to microprocessor development which can result in more powerful processors. Over the last ten years, these have been combined to increase typical processing speeds from 0.05 MIPS (millions of instructions per second) to around 10 MIPS: a 200-fold improvement.

Increase instruction speed. If the processor can execute instructions faster, then it can process data faster. There is a practical limit on the speed at which instructions can be executed, which is the speed at which instructions can be fetched from memory. Memory operations typically take (at the moment) 100-200ns, down from 400-500ns ten years ago. However, the use of caches has improved apparent memory access times to around 20ns, giving maximum instruction execution rates of 30MHz now as opposed to 1 MHz 10 years ago. Note that instruction execution rate is not equal to clock speed - processors may take several clock cycles to execute a single instruction.

Increase processor data bus width. If the processor can process more data per instruction, then it will need to perform fewer instructions to process a given amount of data, and will hence take fewer instructions. Processor bus widths have increased from a typical 8 bits ten years ago to 32 bits now, giving an apparent fourfold increase in throughput. In practice, the improvement gained may be more than this, as operations such as multiply become very much simpler with larger words.

Add more instructions. Extra instructions may also help to make the processor more powerful. If it can multiply two numbers together itself, then, at the very least, it can lose the overhead associated with fetching the instructions needed before from memory. It is difficult to quantify the effect of adding more instructions to a processor's repertoire.

Rethink the instruction set. In the light of experience, it may appear that programmers want different instructions from those provided by the processor's designers. This knowledge may be profitably used in future design.



Undoubtedly, the most successful processor (in terms of volume) of the 1980s has been Intel's 8086 family, as used in IBM PCs. The original 8086 has been extensively developed, and the latest version, the 80486, offers a 25-50 times speed improvement whilst being entirely code-compatible with the original.

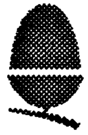
The original 8086 is an example of a CISC (Complex Instruction Set Computer) processor. It can execute a wide variety of powerful instructions, and has a 16-bit data bus. Its little brother, the 8088, which appeared in the original IBM PC, had only got an 8-bit data bus, but its internal data path was 16-bit. This is a compromise between the processing power afforded by having a 16-bit data path in the processor, and the cost advantages of only having to provide an 8-bit memory system.

A couple of generations further on lies the 80386. The 80386 is a 32-bit processor at heart, with a 32-bit data bus. It can perform full 32 bit arithmetic, including multiplication and division. It provides the memory management facilities that are needed to allow proper multi-tasking.

This inherently powerful processor is severely handicapped by the requirement that it be able to run the same software as the 8086. For this reason, it has to have, for example, a set of 16-bit arithmetic instructions as well as its 32-bit ones. All of its extra facilities have had to be added around what was already there.

One of the outcomes of the above is that the chip is somewhat tricky to program directly (in assembler). It has a mind-boggling number of instructions to remember, which makes finding the best way of performing a given action rather tricky. It has little quirks which make life for confused programmers difficult. For instance, the original 8-bit arithmetic instructions in the 8086 had one bit in the instruction code which was not used; the 80386 uses this to determine whether the instruction is 8-bit or 32-bit. This would be fine, except that the 80386 also has a flag in its status register which, if set, turns 8-bit instructions into 32-bit ones and vice versa. This is the sort of thing which produces obscure and hard-tofind bugs.

More on the 80386



Microprocessors make use of registers, which are held on the processor and used to hold operands for arithmetic operations, results from arithmetic operations, loop counters, memory addresses, etc. The 80386 is quite well-endowed with these. However, they all have specific functions - a register which can be used as a memory address cannot be used as an arithmetic register, etc. This reduces the chip's flexibility - for some applications, a number of pointers into different areas of memory may all be needed at once, and for others, a number of operands for an arithmetic operation may be wanted. The 80386 doesn't, despite its broad repertoire of instructions, have the flexibility required.

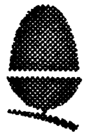
The complex design of the 80386 brings a number (over 1000, at the last count) of bugs in the chip itself. The complexity of the chip makes it extremely difficult to test satisfactorily, especially when operations may interact and bring to light obscure problems. You may remember that early 80386's had a bug in the 32-bit multiply instruction. This wasn't found for a number of months, mostly because people weren't using it. There are a couple of reasons for this: firstly, the chip was (and still is) used almost exclusively to run 8086 code, and secondly that programmers had found that the 80386 would multiply numbers together faster if they wrote a little program to do the multiplication rather than using the instruction provided.

The observation was made back in the late 1970's that these problems might occur if chips continued to get more complex. The major problems are:-

Adding more instructions does not necessarily make the chip faster or easier to program

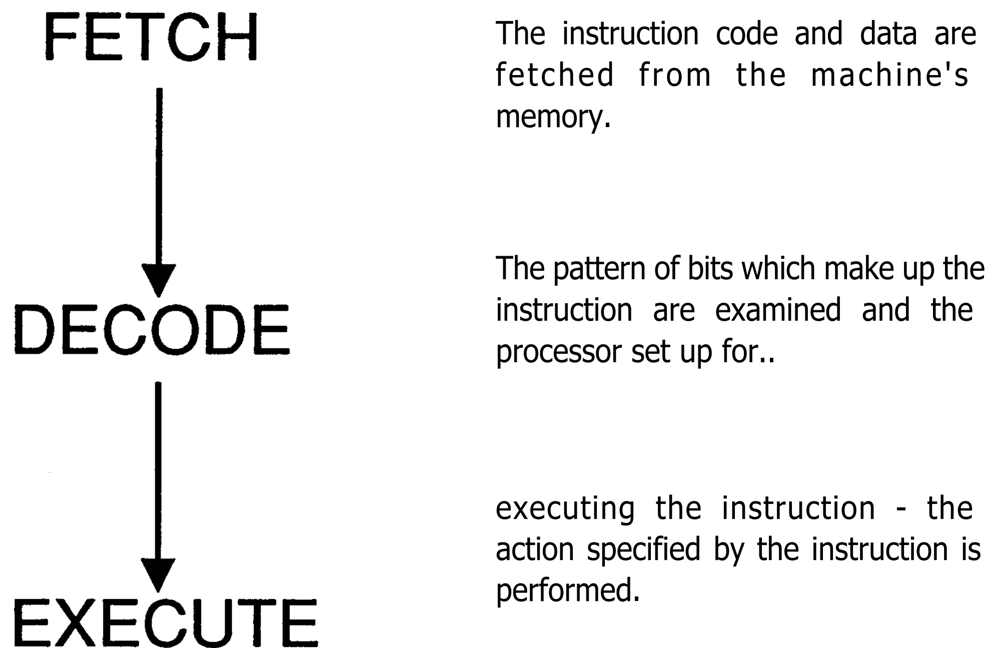
Adding more instructions definitely does make the chip harder to design and to test, and more complex, which results in an unreliable, expensive chip for the consumer.

For these reasons, an alternative philosophy to CISC was proposed: RISC (Reduced Instruction Set Computers), where the microprocessor would understand a small set of carefully chosen instructions, and execute them extremely fast.

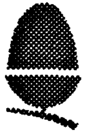


How can a simple processor be faster than a complex one? It would seem that, given a simple processor, all that needs to be done to make it more powerful is to add extra instructions which do their jobs faster than the equivalent sequence of simple instructions on the basic processor. Extra instructions can't slow the processor down, can they?

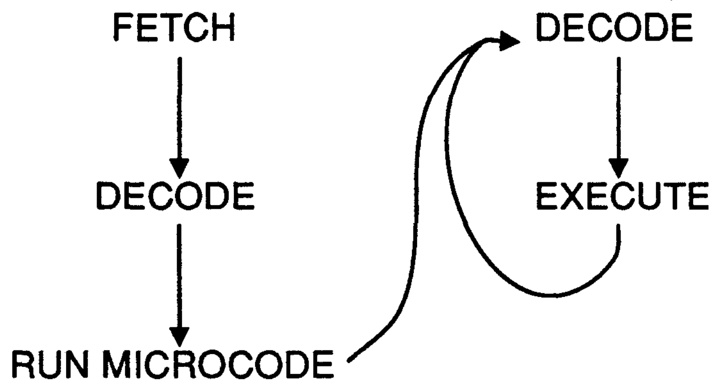
Well the answer is that, in fact, adding extra instructions to a processor can slow it down. A basic architecture for instruction execution is shown in the diagram below:



Adding extra instructions increases the complexity of the 'Decode' phase of instruction execution. The more options it has, the longer it will take. But the real point at which adding instructions to the processor slows it down is when the instructions are sufficiently complex that they cannot be executed directly in hardware, but instead have to be interpreted by programs running on the processor hardware. This 'indirect' form of instruction execution produces what is known as a 'microcoded' processor.



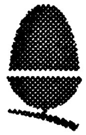
Microcoded processors are slower than their simpler alternatives in the same way as an interpreted computer language will be slower than a compiled one. The instruction execution looks something like this:



The simple instruction execution which we saw earlier has grown another phase - the microcode execution phase. Microcode is the name given to 'sub-machine code' which is actually implemented by the processor hardware, with the instructions which the processor appears to execute implemented as microcode programs.

The fetch and decode of the instruction from the outside world are essentially 'wasted' time in this design. Executing the microcode for the instruction may take several tens of cycles (it's not unusual for an 80386 instruction to take longer than 20 cycles to execute).

There is a tradeoff between hardware complexity and execution speed. For instance, complex hardware can multiply a pair of numbers together in a single cycle, whereas a simpler hardware design will take several. The microcode to interpret a processor's instructions is stored on the chip, in ROM. The space thus occupied, which tends to be considerable, could possibly be put to better use making the existing instructions faster.



Many aspects of life show 80/20 (or 90/10, etc.) splits - 10% of the country's population hold 90% of its wealth, and so on. The same tends to hold true for processors - 80% of the processor's time is spent executing 20% of its instructions. There are a few reasons for this:-

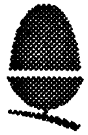
> The simple instructions (load a word from memory, call a subroutine) tend to be heavily used as they are the most useful instructions provided. Others, such as division instructions, tend to be used much less frequently.
>
> Most code which processors execute wasn't written by humans, but by compilers. A compiler is a computer program which takes another program in a high-level language and translates it into the machine code of the processor on which it is to be run. Compilers aren't very clever - they tend to only use the simple instructions provided by the machine.
>
> For the common CISC processors, the 80x86 and 68x00, most of their time is spent running code written for the lowest member of the family (the 8086 or 68000) which, of course, doesn't use the more complex instructions provided by the later models.

Again, would not the space used on the chip by the microcode for these unused instructions be better spent speeding up the ones which are used all of the time?

The concept of a reduced instruction set computer was first evolved at Stamford University in 1976. They produced a theoretical design on paper for a RISC processor which had roughly the power of a 68000, but this was at the time of 6502's and 280's. More recently, RISC has caught on with most leading chip companies: Intel has produced the i860 and Motorola the 88000. Neither of these have really found their way into mainstream computing, although the i860 has seen some use in graphics accelerator cards and is now shipped on the colour card for the NeXT machine.

Some companies have released machines based around RISC processors, notably IBM with their RT/6150, recently re-vamped as the RS/6000 in an attempt to sell some, Sun with their successful range of SPARC-based workstations, and Acorn.

The ARM



The ARM (Acorn RISC Machine) is Acorn's own RISC processor, and is used in the Archimedes/A3000 machines. Acorn scored several notable firsts with the ARM and Archimedes; in particular, they produced the world's first 32-bit RISC micro, and it cost less than £1000, which, in 1987, was an achievement.

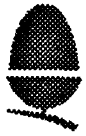
The ARM is a 32-bit RISC processor. It was designed by Acorn, and mass-produced by VLSI Technology. It's now the second-best selling RISC processor in the world - at the time of writing (December 1990), Inmos's Transputer holds the lead (350,000 units), Acorn's ARM comes next (140,000 units) and Sun's SPARC comes third.

How does the ARM compare with other processors in terms of processing power, chip complexity and cost? Well...

Processor	Cost	Transistors	Clock	MIPS
6502	£2.50	20,000	2MHz	0.1
80386	£300.00	300,000	33MHz	3.5
80486	£650.00	1,200,000	25MHz	10
ARM2	£20.00	25,000	8MHz	4
ARM3	£100.00	100,000	30MHz	13.5

System	Cost	Graphics	MIPS
'386 PC	£1,400	VGA	3.5
'486 PC	£4,000	VGA	10
RS/6000	£15,000	None	22.5
A3000	£800	VGA+	4
A540	£3,000	SuperVGA+	13.5

So, in terms of price per MIP, the ARM chip is comfortably ahead of the competition. It's also the leader in terms of MIPS per watt, which makes it useful for powerful battery-operated systems, such as the forthcoming Active Book.



The ARM is a 32-bit pipelined RISC processor. You should already be familiar with the concepts of 32-bit and RISC, but what is pipelining?

Remember that executing an instruction on a processor uses a number of 'phases' - the instruction has first to be fetched, then decoded and finally executed. These three phases use different bits of the chip's circuitry. Using the processor to process just one instruction at a time is actually rather wasteful, as only 1/3rd of the chip is being used at any given instant.

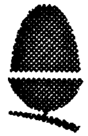
A pipelined chip takes advantage of the fact that the three instruction execution phases are essentially separate to increase the effective speed of the chip by processing three instructions at once. At any given moment, it will be executing one instruction, decoding the next and fetching the one which it is going to execute in two instruction's time. This reduces the number of cycles taken to execute each instruction from three to one - tripling the speed of the processor.

There's only one problem - the processor needs to know where the next-but-one instruction is coming from, and that is not always possible. Most of the time, however, the processor will be executing instructions from memory in order, and hence it's a good guess that the next-but-one instruction to execute will be the next-but-one in memory.

If this proves not to be the case, because the processor has executed a 'branch' instruction, then the contents of the pipeline are invalid, and need to be cleared. This process is known as 'flushing' the pipeline, and, after a branch, it will obviously take a couple of cycles to refill the pipeline before the processor can start executing instructions again. Note that this is only as bad as having no pipeline, not worse!

Branch instructions are therefore more expensive to execute than other types of instruction. This might not seem to be a serious problem on the face of it, but a study has shown that every third instruction executed by a compiled program on a VAX (CISC minicomputer) is a branch. It would therefore seem to be a good idea to attempt to find ways by which the number of branches in the code could be reduced.

Conditional Execution



Consider the following code:

```
CMP       R1,#1
BNE       sub1
ADD       R2,R2,#1
B         continue .
sub1
SUB       R2,R2,#1 .
continue
...
```

This code compares a register (a variable) with 1, and either increments or decrements another depending on the outcome of the comparison. In the case where the R1 <> 1, it will take 5 cycles to run, or if R1 = 1, it will take 6 cycles. A significant amount of the time is spent refilling the pipeline after each branch instruction.

The ARM allows every instruction to be executed conditionally. This allows bits of code such as the one above to be written much more concisely:

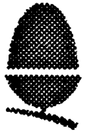
```
CMP       R1,#1
ADDEQ     R2,R2,#1    ;Only executed if R1 =1
SUBNE     R2,R2,#1    ;Only executed if R1 <> 1
```

This code will always execute in three cycles. The speedup is entirely due to losing the branch instructions, allowing the pipeline to remain full all the way through the code. Incidentally, code to perform the same operation on an 8086:

```
cmp       ax, 1
bne       sub1
add       dx,1
jmp       continue
sub1:
sub       dx,1
continue:
```

Well designed RISC processors may allow more compact code to be written than CISC processors!

ARM's Addressing Modes



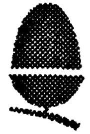
The ARM supports a number of powerful addressing modes (ways of getting data from memory.) Examples of some of these, with their 8086 equivalents, follow:

```
LDR     R1, [R2],#4 ;Loads R1 from the address given by R2, and then adds 4 to R2 In
8086:
mov     ax, [bx]
add     bx, 4

LDR     R1, [R2,#4] ;Loads R1 from the address given by (R2+4) In
8086:
mov     ax, [bx+4]

LDR     R1. [R2,#4]! ;Adds 4 to R2, and then loads R1 from the resultant address
add     bx, 4
mov     ax, [bx]

LDR     R1, [R2,R3,LSL#4]        ;Loads R1 from (R2+16" R3)
In 8086:
mov     cl, 4
shl     dx, cl
mov     di, dx
mov     ax, [bx+di]
```

There are, of course, cases where the 8086's addressing modes result in more compact code than the ARM's: in particular, the 8086 can add together two registers and a constant offset to generate a memory address. The ARM can either add together two registers, or a register and a constant.

You may feel that the comparison between the 8086 and the ARM is a little unfair, as the 8086 is an older processor than the ARM and only 16-bit, as opposed to the ARM's 32. This is true, but the 8086 is a more complex processor (at the circuit level) than that ARM, and it can also be argued that comparing a 300,000 transistor processor (the 80386) with a 25,000 transistor ARM is slightly unfair, too!

Subroutine Calls



Most processors provide some sort of subroutine call/return mechanism. Traditionally, this has been achieved by the use of a stack, which is handled by the processor, and provides a pair of instructions to call and return from a subroutine. There will usually be instructions to transfer the contents of registers to (push) and from (pop) the stack, as it is a convenient place to store the contents of registers which the subroutine would like to use but the calling program would like to see preserved.

The ARM doesn't provide a stack. Instead, it provides instructions to allow a stack to be implemented easily (see above), and a subroutine call mechanism which makes use of one of the ARM's registers, R14. When a subroutine call is executed, the return address for the subroutine is placed in R14, which is known as the link register. This method has a couple of advantages over the sack-based approach: it's simpler, as the processor doesn't have to know about stacks, and faster, as no memory access has to take place to store the return address on the stack, or retrieve it on returning from the subroutine. It does have disadvantages, though - a subroutine may not call any other subroutines before it has saved the link register, as otherwise its own return address would be corrupted.

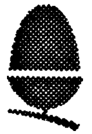
The. ARM provides multiple register load/store instructions, which can be used to implement a stack and for saving temporary registers and the link register during subroutine execution, e. g.

```
.Subroutine
STMFD       R13!, {R0-R3,R14}        ;Save registers and return address

LDMFD       R13!, {R0-R3,PC}         ;Restore registers and return.
```

In 8086:

```
Subroutine:
push ax
push bx
push cx
push dx

pop dx
pop cx
pop bx
pop ax
ret
```

ARM's Arithmetic Instructions



ARM's arithmetic instructions operate on a `register-to-register' basis - i.e. they take their operands from a pair of registers and store the result in a third. Example:

```
ADD         R1, R2, R3 ; R1 = R2+R3
```

This contrasts with the 8086, which only allows the specification of two operands, that which is to be operated on and what to operate on it with. The 8086 can, however, perform arithmetic directly on operands in memory. This takes longer to do - two memory accesses are needed, one to fetch the operand and one to write it back.

```
add         ax,10           ; ax = ax+10
add         [dx], 100       ; add 100 to the memory location pointed to by dx.
```

ARM code for the second of these would be:

```
LDR '       R0,[R1]
ADD         R0,R0,#100
STR         R0,[R1]
```

The ARM does provide enough registers that this type of operation is not often needed.

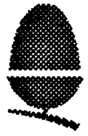
The ARM provides a full set of arithmetic instructions, including addition, subtraction, AND, OR, EOR, Bit Clear (AND NOT) and moves. One instruction type which appears to be missing is the binary shift type (e.g. shift left or right), which is particularly useful for multiplying and dividing numbers by powers of two.

The last operand in any arithmetic instruction may be shifted right or left by either a given amount or by the contents of a register. This makes the arithmetic instructions particularly powerful, as they can often incorporate what would be two separate operations into a single instruction. The provision of the shifted often removes the need for a `multiply' instruction, e.g.

```
ADD         R1,R2,R3,LSL#4          ;R1 = R2+16*R3
```

And one which you might like to verify by hand:

```
RSB        'R1,R2,R2,LSL#5
RSB         R1,R1,R1,LSL#6
ADD         R1,R2,R1,LSL#3
MOV         R1,R1,LSL#6 ;R1 = R2*1000000
```



The processing speed of an ARM2-based system is limited by the bandwidth of the memory system, which determines the rate at which instructions can be fed to the processor. In order to overcome this bottleneck, the ARM3 processor has been developed.

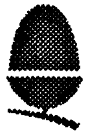
The ARM3 processor adds a 4k on-chip cache to the ARM2. The cache is an area of highspeed (around 15nS) memory, in which the last few instructions exectued by the processor are stored. The hope is that the next instruction required will be found in the cache, from where it can be retrieved very quickly, rather than from main memory. The presence of the cache allows the ARM3 to run at speeds in excess of 30MHz, with an instruction execution rate in practice of 10-15 MIPS.

Like pipelining, the cache requires to predict the future - the next instruction to be executed has to be found in the cache for any speedup to occur. Pipelining relies on the fact that coderuns in straight lines most of the time; cacheing takes a slightly broader view of a program and relies on code running in loops.

Most code runs in loops. For instance, to look for the string 'Dave' in another, a program might compare 'Dave' with thefirst four characters of the string, and the the second four, until it either reaches the end of the string (fail) or finds a match. This operation would typically be performed as a loop.

The first time through the loop, the code has all to be fetched from main memory, and so the cache provides no benefit at all. After the initial pass, and assuming that the loop is less than 4k long (most are), all of the instructions will be fetched from the cache. In real programs, 98% or more of instructions are fetched from the cache on an ARM3.

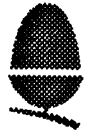
The cache is '64-way set associative,' which means that 64 distinct blocks of 16 bytes may be cached. The 80486 only has a 4-way cache, making it less effective if a number of different areas of memory are being accessed in quick succession (e.g. if a loop calls more than a couple of subroutines.)



A complete computer system cannot just be built around a processor chip. It needs things like a screen display, a keyboard and some sort of mass storage (floppy or hard disk.) Some system memory (RAM/ROM) is also desirable. The ARM doesn't provide all of these facilities itself, and so Acorn have designed another three chips which, together with ARM, make up the 'ARM chip set.'

> MEMC is a memory controller chip, which handles memory map decoding, ROM, RAM and I/O access each at various speeds, page mode (fast) RAM access, virtual-to-real address translation for protected multi-tasking and processor clocking.
>
> VIDC is a custom, programmable video display generator, which can produce displays of resolutions of up to and beyond 800x600 in up to 256 colours from a palette of 4096 and replay 8 channels of sampled sound in stereo at up to 33KHz through internal dual 8-bit log DACs.
>
> IOC is a custom, programmable I/O controller which handles peripheral decoding and timing, controls interrupts, drives the keyboard, provides an IIC interface and produces a few uncommitted I/O lines.

These three chips allow a complete ARM-based system to be built with just the addition of RAM, ROM and the desired set of peripheral circuitry. Some other items need to be provided, such as a system clock and some buffering on the video output lines, but these are not at all complex.



MEMC (short for MEMory Controller) provides the hardware needed to access memory on an ARM-based system.

Address decoding. The area of memory which the processor can access is split into various areas, including a RAM area, a ROM area and an I/O area. MEMC is responsible for looking at the address generated by the processor and working out which area it's for.

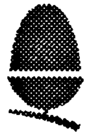
ROM access timing. ROM chips come in different speeds, with access times ranging from about 450nS to 100nS. In order to accommodate this variation, MEMC can generate different cycle timings when ROM is being accessed.

RAM refresh. The RAM on the system is 'dynamic' - it forgets what it's been told to remember! This might seem catastrophic, but it only forgets after a little while (10mS or so) and has circuitry to 'refresh' its memory. This is driven by MEMC.

RAM address generation. Dynamic RAM chips need to be given an address in two halves (this reduces the number of address pins needed on the package and helps keep the chip small and cost low.) This 'address multiplexing' is performed by MEMC, which also generates the necessary timing strobes.

Fast RAM access. Dynamic RAM chips are arranged in such a way that it is generally quicker for them to access successive memory locations than random ones. Most of the time, processor and video accesses to RAM are sequential, and MEMC makes use of this to almost double the effective speed of the RAM.

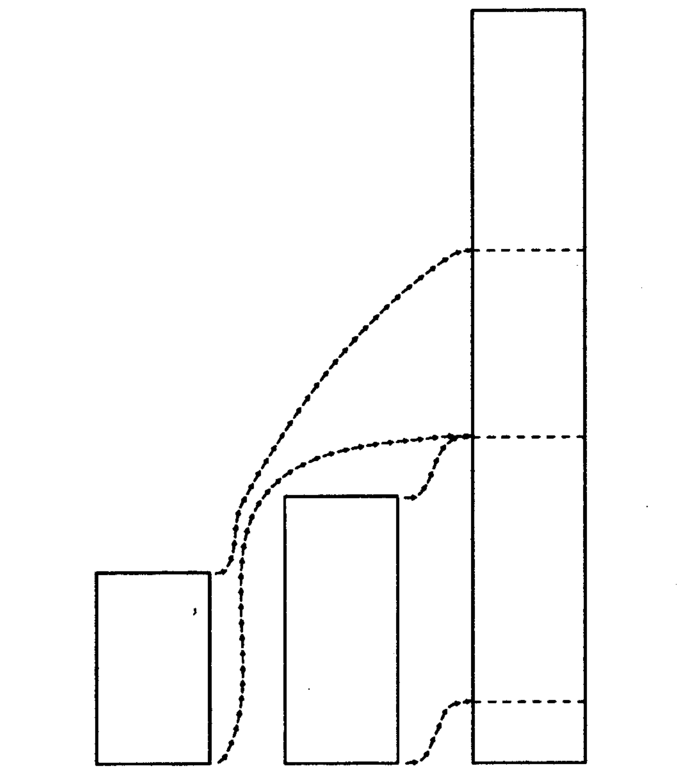
DMA. Handling the system RAM, MEMC is ideally placed to control DMA (direct access to system RAM by peripherals,) This feature is used to allow the video display and sound samples to be held in system RAM, rather than in a separate area of memory.



When attempting to run more than one program on a computer at once ('multi-tasking'), it is desirable to protect them from each other. In the normal run of things, with software which works perfectly, this protection isn't needed, but in the real world, perfect software is not often come across.

Mainframes in the 1970's used (and still do) 'virtual machines' to run multiple tasks on a single computer. Each task which is running appears to have the whole computer (give or take some memory) to itself. This eases the programmer's job - he doesn't have to worry about any other tasks - and gives protection. This system is implemented on the 80386, and also on ARM systems by MEMC.

So how do we make a computer look like several smaller ones? MEMC has internally a 'page translation table' which is used to map addresses produced by ARM ('virtual' addresses) onto addresses in RAM ('real' addresses.) This needs to be reprogrammed during task switches, in order to 'map out' the task which has just finished running and 'map in' the new one.



In this diagram, the system's memory is represented by the column on the right. It contains two virtual machines, each running a single task. When a task is being run, the addresses from the processor are translates by the 'page translation table' in MEMC, which allows the tasks to think that they occupy the same area in memory.

MEMC's page translation has 128 entries. Each entry specifies where 1/128th of the system memory (a page) should appear in the processors memory map. In order to translate addresses, MEMC attempts to find a table entry which matches the processors address.

VIDC (VIDeo Controller) is the chip responsible for producing video displays and sound. It is highly programmable, in terms of both screen resolution, colours and scanning rates. In particular, it can produce screen displays of up to 1056x256 on a standard monitor, in **2, 4,** 16 or 256 colours; or 800x600 plus on a multisync monitor, or, with a small amount of extra circuitry, 1152x984 monochrome on a 'workstation' monitor.

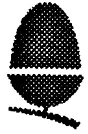
VIDC also provides support for a single hardware 'sprite', which is used to display the desktop pointer, amongst other things. MS Windows apparently spends 25% of its time just moving its pointer around its desktop, as it has to do it in software.

Sound is produced by replaying samples (like a CD) through a pair of DACs. VIDC supports up to 8 channels of sound, and can set the stereo position for each channel independently. The usual playback rate on an Archimedes/A3000 is 20kHz, giving a bandwidth for the sound system of around 8kHz. This isn't hi-fi, but it's significantly better than a telephone or MW/LW radio broadcasts.

The video frame and sound samples are stored in main memory. This contrasts with the PC world, where the video system sits on a separate card and the video controller has local memory. There are pros and cons for each system, but the V!DC system scores on speed of picture update and releasing screen memory for programs when it's not being used, but loses because the processor has to share its RAM with the screen, which may slow it down.

Screen addressing is cunningly done not by V!DC, but by MEMC. MEMC is then responsible for arbitration between processor, screen and sound accesses to main memory, and already has all of the hardware needed to generate RAM addresses onboard.

VIDC also provides support (but not complete circuitry) for genlocking and video overlay. Genlocking is the name given to a process of synchronising two video sources, so that they scan frames together. It is a prerequisite for video overlay, which involves switching between video sources to cause one to appear to be overlaid on the other. Uses include the addition of scrolling credits to video programmes, and the addition of teletext-style subtitles.



The ARM provides no special signals for interfacing to peripherals. Hence they are 'memory-mapped' - each peripheral occupies a few memory locations which the processor can access. Peripherals may also need to be accessed at a slower speed than memory: where RAM may have a 60nS cycle time, a peripheral chip may not leave much change out of a microsecond.

MEMC (see above) decodes out peripheral accesses by the processor. But it doesn't work out which peripheral is to be accessed; neither does it know how long it should take. These tasks are carried out by IOC.

IOC is also responsible for handling interrupt requests from peripherals to ARM. Peripherals signal to the processor that they want attention by sending an interrupt signal; this is much more efficient than having the processor poll peripherals every so often. The processor has only got two interrupt inputs, though, and hence cannot distinguish which one of the (several) peripherals in the system has generated the interrupt. Therefore, peripherals signal interrupt requests to IOC, which checks to see if interrupts from that particular peripheral are allowed and passes the request on to the processor if they are. The processor can then interrogate IOC to see which peripheral requested the interrupt, rather than having to poll all of the peripherals to find out.

The keyboard interfaces directly to IOC down a bidirectional serial link. IOC also controls the RTC (real-time clock) chip via a pair of its I/O lines, which are used to implement an IIC bus. This bus, originally designed for use in teletext TVs and video recorders, now has a wide variety of possible customers - as well as RTCs, programmable frequency generators and latches, IIC may allow you to program your video to run in slow reverse record mode...try doing that from the front panel!

IOC provides a few more uncommitted I/O lines, which may be used directly for controlling simple hardware, and a number of timers, which can be used to interrupt the processor after a given time or at a given rate. One of these is used to update the machine's internal clock - that's why it tends to be about 30 minutes out after the machine has been on for 24 hours. The RTC is, however, accurate to a few seconds a year.