1. Introduction

The Intel XScale processor used in Castle's IYONIX pc does not have the old 26-bit addressing modes that are used by earlier versions of RISC OS, and the IYONIX pc is therefore unable to run 26-bit applications directly. In order to use these applications they must either be converted to run in the XScale's 32-bit addressing modes, or a program must be used to provide an emulation of the 26-bit environment that these applications expect.

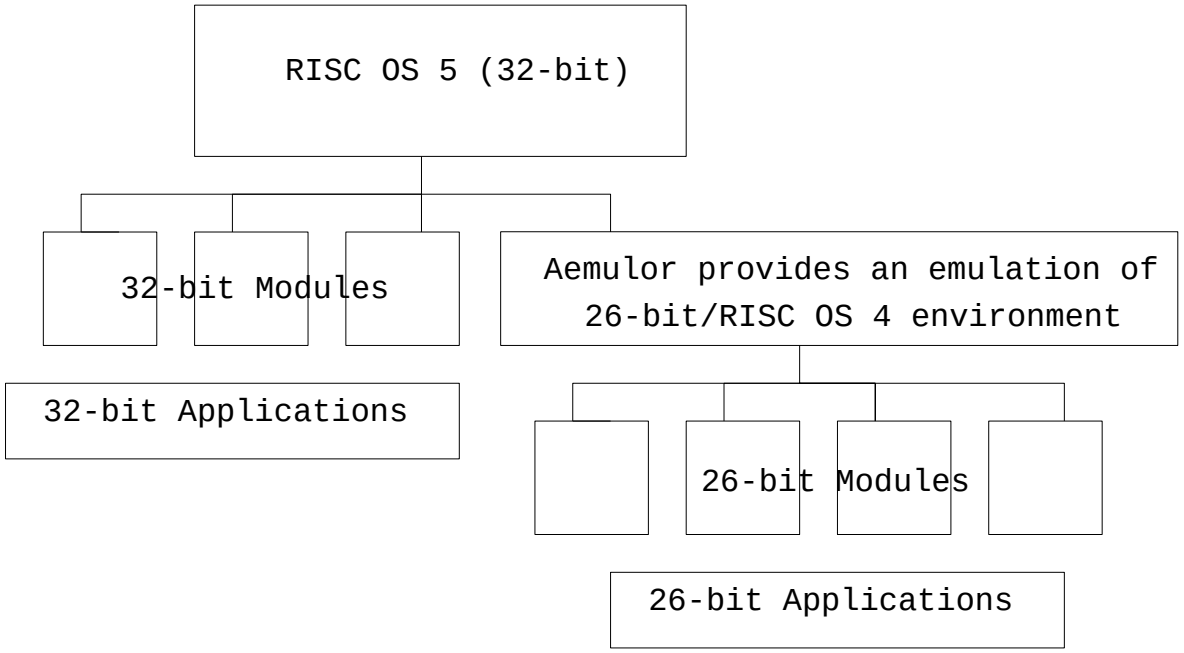
Aemulor is a program that allows many 26-bit RISC OS applications, including important applications like Impression and PhotoDesk, to be used on the IYONIX pc with RISC OS 5. This article will explain how Aemulor achieves this, and then suggests some ways in which Aemulor may be developed in the future.

The first release of Aemulor is intended to run mainly 26-bit RISC OS 4 desktop applications, and particularly those applications such as Impression that are unlikely to ever be converted to run natively on the IYONIX pc. Some other applications, such as games that do not directly access hardware, will also run under Aemulor but it should be noted that Aemulor does not provide a full emulation of the RiscPC hardware; only a 26-bit CPU emulation and RISC OS 4 API.

In general, the more low-level a program is, the less likely it is to work correctly under Aemulor. So, for example, device drivers that directly access hardware are unlikely to work under the first release of Aemulor. Drivers that use the documented RISC OS APIs are more likely to work.

2. Overview

Aemulor can be thought of as 26-bit/RISC OS 4 compatibility layer that sits between the 26-bit applications/modules and the 32-bit XScale/RISC OS 5 of the IYONIX pc as shown below:



This means that, in general, 32-bit applications running natively on the IYONIX pc are unable to use the 26-bit modules that are running under Aemulor. Having said that, RISC OS is unaware that there is anything unusual about the 26-bit applications that are running under Aemulor; to RISC OS they appear as normal applications - they appear on the icon bar as normal, and can communicate fully with other 26- or 32-bit applications. They can also be terminated using Alt-Break if necessary. Indeed the *user* needn't know whether an application is 26- or 32-bit code because Aemulor can be configured to transparently run all 26-bit code.

There are two main components to Aemulor - the 26-bit CPU emulation, and the RISC OS 4 API emulation; the former largely determines the speed at which 26-bit programs will execute, and it is very important that Aemulor executes 26-bit code as quickly as possible. The latter component, emulation of the RISC OS 4 API, largely determines which applications will work and which won't.

Whilst I'll have more to say about the 26-bit CPU emulation, and in particular how Aemulor has been optimised to obtain good performance, the reader should understand that the development of Aemulor has been split almost 50/50 between the RO4 API emulation and the 26-bit CPU emulation. Contrary perhaps to the expectations of most people, emulating a 26-bit CPU on the 32-bit XScale has not been the most difficult part of developing Aemulor.

However, I have opted to describe in detail the performance tricks that were devised whilst developing Aemulor, because I believe that they may be useful to programmers writing other software which requires high-speed emulation of a CPU. The reader will find it helpful to have some knowledge of the ARM instruction set.

3. RISC OS 4 API emulation

In order to run applications written for RISC OS 4 or earlier on a RISC OS 5 machine, Aemulor must hide a number of changes to the OS API, so that the applications still see the old behaviour that they expect.

3.1 SWIs and CPU flags

An application requests services from the OS by calling a SoftWare Interrupt (SWI) and providing various information in the CPU registers. The majority of SWIs used to be defined as preserving the state of the N,Z and C flags of the CPU over the SWI call. With the arrival of a 32-bit RISC OS this has changed and most SWIs now corrupt these flags, which would cause some 26-bit applications to fail. Aemulor must therefore preserve the flags on behalf of the application for those SWIs which now corrupt these flags. Other SWIs have to be handled specially by Aemulor because they actually return something meaningful in these flags.

3.2 SWI API changes

A number of SWIs have been redefined slightly because they were previously defined as using top-bit set values (ie. negative values when viewed as signed numbers) for a special-purpose; these values are now valid addresses so the APIs have been redefined as accepting only -1 and/or 0 for this special purpose.

Some other OS SWIs used to accept a 26-bit address in a register, and use the remaining 6 upper bits of the register has flags to change the meaning of the SWI. These SWIs have been redefined so that they can accept a full 32-bit address and the flags have been moved into another register. Aemulor must therefore intercept calls to these SWIs and separate the flags and address before calling RISC OS.

3.3 Modules

RISC OS 5 has no support for 26-bit modules. Rather than trying to patch or wrap 26-bit modules so that they could be loaded directly by RISC OS 5, it was decided to handle 26-bit modules within Aemulor itself, so that - if necessary - 32-bit applications could use a 32-bit version of module X, whilst 26-bit applications could instead use the 26-bit version of the same module. This is necessary if the 32-bit version is not backwards compatible.

In fact this is the case for the SharedCLibrary which is used by a large number of applications. The 32-bit version supplied with the IYONIX pc does not provide support for the 26-bit procedure calling standard (APCS-R) used by 26-bit C programs. Aemulor therefore loads the 26-bit version that is supplied with the Castle C/C++ Development Suite, and hides the 32-bit one so that 26-bit applications see only the 26-bit SharedCLibrary. 32-bit applications still see the 32-bit SharedCLibrary of course.

3.4 Aemulor RMA

Additionally, because code running in an emulated 26-bit addressing mode only has a 26-bit program counter, it can only address 64MB of address space. RISC OS modules are loaded into an area of memory called the RMA (Relocatable Module Area) and on RISC OS 5 this area has been moved to a high address so that the application space can be larger (currently 512MB). So Aemulor cannot load 26-bit modules into the RMA because they would be inaccessible in 26-bit addressing modes without some form of address translation which would make emulation much slower.

Aemulor therefore also provides an emulated RMA, located at the same address as the RMA on RISC OS 4. The 'Aemulor RMA' is then used to load all 26-bit modules, and any RMA-operations performed by 26-bit code are redirected to this memory area. This area is also used for running 26-bit Utility programs for the same reasons.

From RISC OS's perspective, the Aemulor RMA is a normal dynamic area, but Aemulor remaps the memory at an address below 64MB so that it becomes addressable within the 26-bit environment. Because this emulated RMA is visible to all applications, native 32-bit applications are also restricted to a maximum size of 28MB each (as per RISC OS 4) whilst Aemulor is running. It is hoped that this limitation can be removed with a later version.

3.5 BASIC

Aemulor is capable of running a 26-bit version of the BASIC interpreter module under emulation, in the same way that it runs the 26-bit SharedCLibrary module. However, to avoid the licensing implications of distributing BASIC, and to give much better performance, Aemulor copies the 32-bit BASIC module supplied with RISC OS 5, and then patches it to run 26-bit BASIC programs under Aemulor, including those that contain inline 26-bit assembler code. Because Aemulor actually analyses the BASIC module rather than making assumptions about the positions of the code sections that must be patched, it should be robust against changes that Castle may make to the BASIC module in the future. As with the SharedCLibrary, 32-bit applications see the unmodified 32-bit BASIC module.

4. Running 26-bit ARM code on a 32-bit XScale processor

Since the 32-bit XScale processor is unable to directly execute code that has been written for 26-bit addressing modes, some form of emulation must be provided, ie. Aemulor must pretend to be a 26-bit ARM processor. It therefore contains a full software emulation of an ARM processor, actually an hybrid of an ARM610 and a StrongARM; ie. for maximum performance it will emulate a StrongARM processor, and applications that know about the separate instruction and data caches of the StrongARM (a feature it shares with the XScale) can be executed much faster under Aemulor. Some older applications rely upon having a single cache like that provided by the ARM3, ARM6 and ARM7 processors, and cannot benefit from this acceleration. For example, they may modify program code or load/generate new code, and then execute it without first calling OS_SynchroniseCodeAreas to ensure that the program code written into the data cache is also visible to the CPU via the instruction cache.

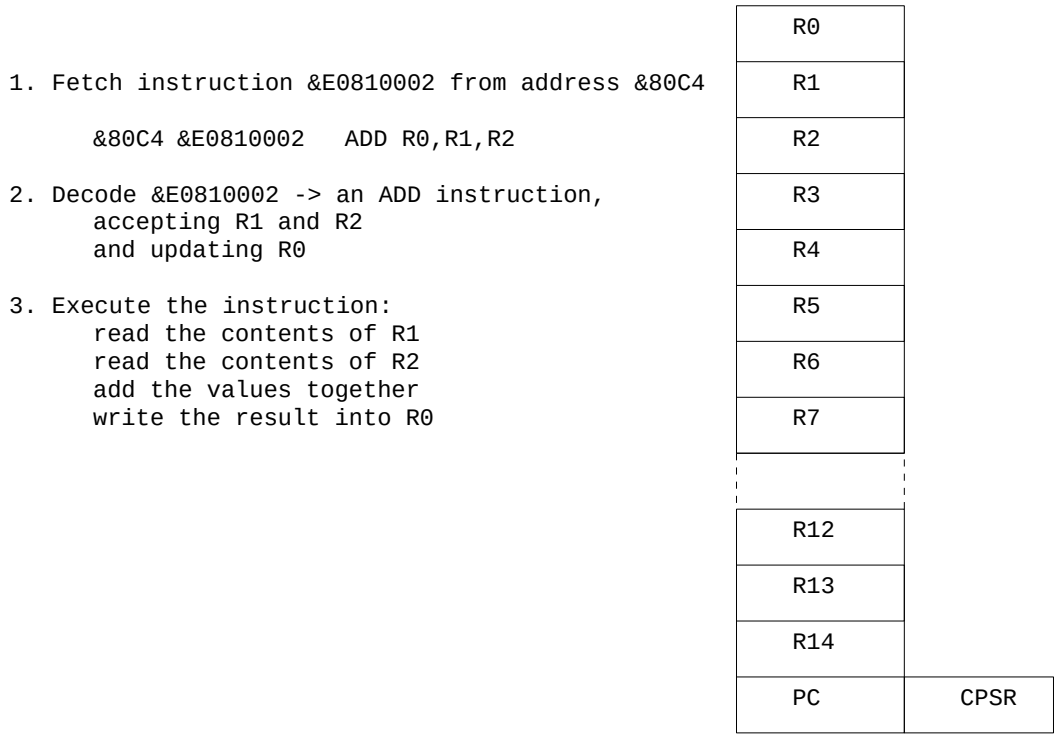
So Aemulor actually consists of two different types of emulation - the ARM610

emulation provided by Aemulor employs a traditional interpreter and will be described after a brief introduction to the basic ideas of CPU emulation. The second approach, used to greatly accelerate StrongARM-compatible code, will then be described. Whilst this is still referred to as emulation, the term is a bit misleading in this case, because Aemulor only executes those instructions whose behaviour is different in 26-bit and 32-bit addressing modes, instead leaving the XScale to run the remaining instructions. Since most ARM instructions do *not* depend upon whether 26-bit or 32-bit addressing is being used, this form of 'emulation' is a lot faster than a traditional interpreter.

4.1 Basics of Emulation

A software emulation of a CPU, or indeed an hardware implementation, traditionally consists of the following three phases:

1. Fetch the instruction from memory
2. Decode the instruction
3. Execute the instruction, changing registers and/or memory



Implementing these 3 phases directly in software and performing each of the 3 phases on every instruction that the emulated CPU executes is very slow. Even well-optimised code will take in the region of 30 to 50 instructions which, allowing for pipeline stalls etc, will be around 100 times slower than native execution. Hence, the 600MHz XScale processor used in the IYONIX pc might be expected to give around the same performance as a 6MHz ARM, ie. slower than the original Archimedes machines. In fact the very first versions of Aemulor really were that slow.

4.2 A Simple Interpreter

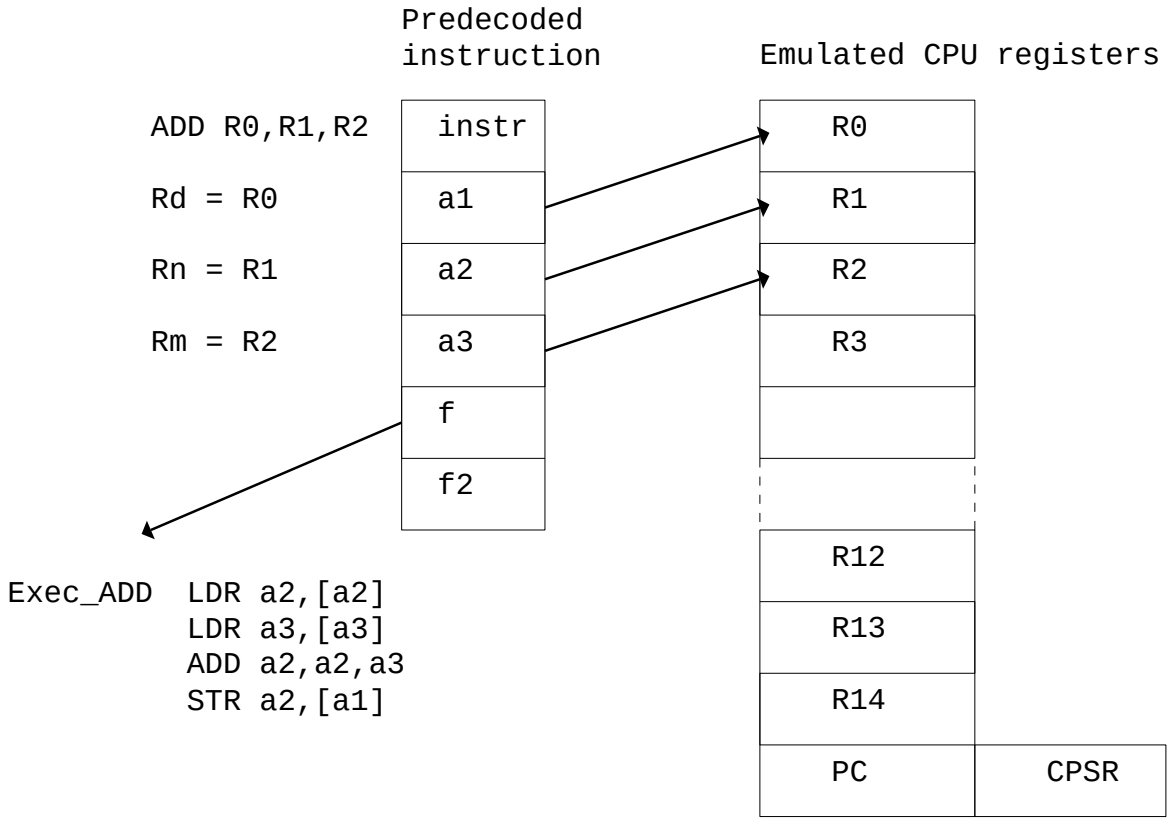
The solution to this unacceptably low performance is to reduce or eliminate the repetition that is inherent in the above scheme. Almost all programs spend most of their execution time running a few sequences of instructions over and over again - referred to as 'looping.' It can be seen that this must be true of most programs, because otherwise a 256KB program would run to completion in much less than 1 second even on a 30MHz ARM processor!

Processors invariably fetch and decode each instruction in a loop on every iteration of that loop, although modern processors greatly reduce the cost of instruction fetching by cacheing the

instructions locally within the processor chip, rather than retrieving them from the slower external memory every time. Software emulation, however, must also aim to reduce the cost of decoding the instructions every time, so that only the actual execution phase is left.

The execution phase can't really be avoided, since this is where the actual work is done, ie. it produces the output results of the program. Having said that, the execution cost should be reduced as far as possible by careful programming, and perhaps even by trying to optimise the original 26-bit code since it is very unlikely to already be expressed in optimal form, especially given the change of processor.

A suitable first step towards improving the speed of the interpreter is to keep a cache of recently-executed instructions in their already decoded form. Since the instructions within a loop will already be available in decoded form after the first iteration this will increase the execution speed of the loop. At this point it's worth describing how instructions can be stored in decoded form to speed up execution, since these 'predecoded instructions' are used throughout Aemulor and the remainder of this article.



The illustration above shows a simple ADD instruction, ADD R0,R1,R2, in 'predecoded' form as three operands a1, a2 and a3 which are direct pointers into the emulated register set (stored in memory), and the address f of the code which performs this operation, ie. Exec_ADD. This routine is very short and simply collects the values from the appropriate registers, performs the necessary operation (in this case addition), and then writes the result into the correct destination register.

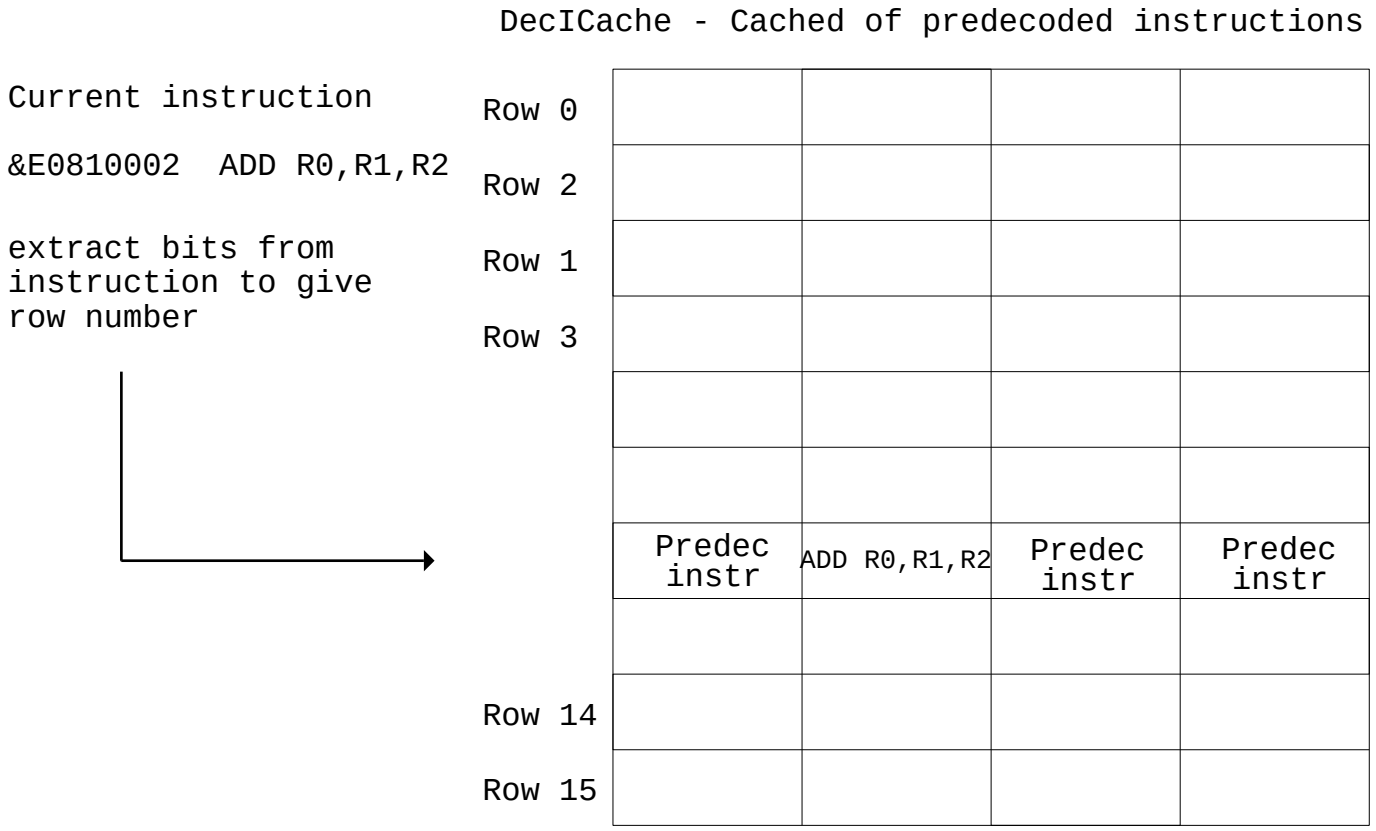
Aemulor has about 200 short routines which perform the various operations available in the ARM instruction set. Passing the source and destination registers to each routine keeps the number of routines manageable whilst also keeping each routine fairly short which is important for good performance.

The 'f2' field in each predecoded instruction is used to implement conditional exection; in this case 'f' points to a short stub routine, eg. Exec_EQ, which tests whether the emulated flags are such that the instruction *should* be executed - if they are, then it passes control to the appropriate routine via 'f2.' If not, then the next predecoded instruction is executed instead. For example, ADDEQ R0,R1,R2 in predecoded form would be:

instr	ADDEQ R0,R1,R2
a1	pointer to emulated R0
a2	pointer to emulated R1
a3	pointer to emulated R2
f	pointer to Exec_EQ stub routine
f2	pointer to Exec_ADD routine

This approach maximises the speed of unconditional instructions, since these are much more common than conditional instructions.

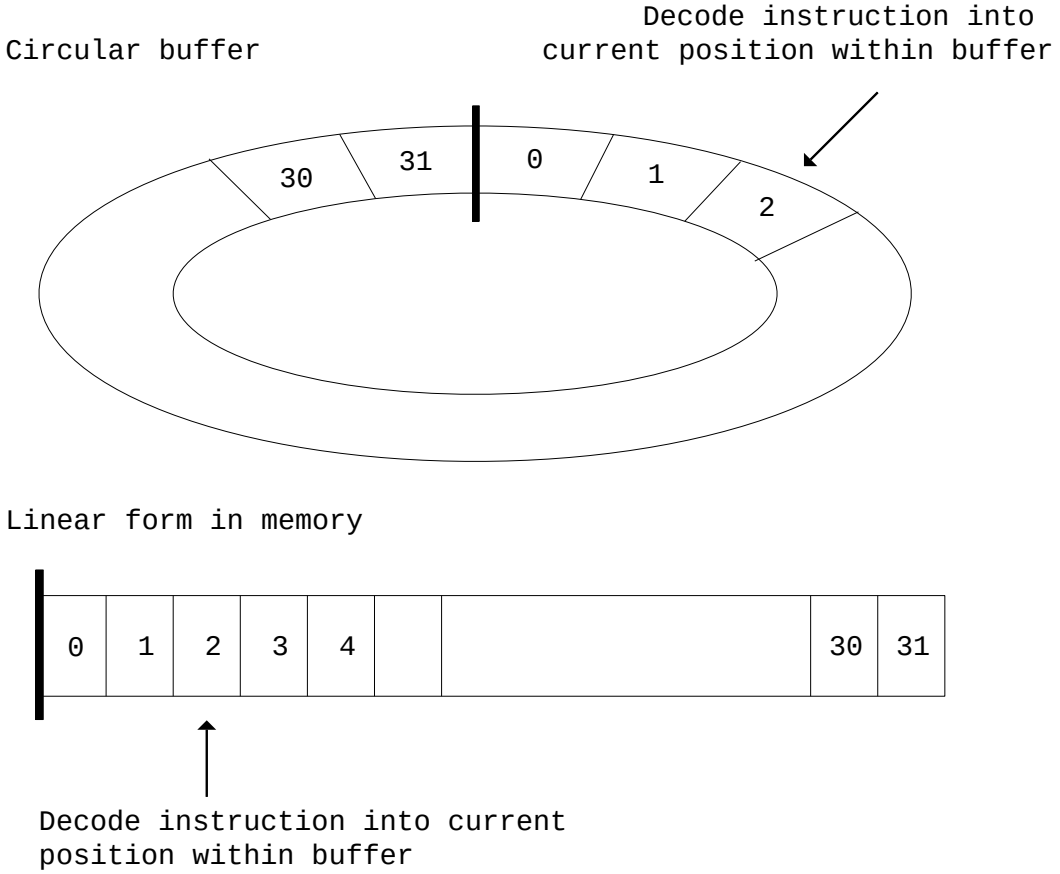
The cache of predecoded instructions, referred to as the DecICache within Aemulor, is structured and accessed very like the instruction cache of a processor but with one notable exception - to maintain compatibility with code that is not StrongARM-aware, the cache is accessed using the current *instruction* rather than its address. This is slightly slower because the interpreter must first read the instruction from memory, but it does mean that if the code is modified during its execution then the emulation will still be correct.



Before deciding to decode an instruction X, the interpreter searches a given row of the DecIcache to see whether it has already decoded this instruction. The exact bits extracted from an instruction to give the row number must be chosen to ensure that commonly-encountered instructions are distributed evenly over the cache rows, particularly since the width of each cache row must be kept fairly small because, unlike a hardware cache, a software emulator must check each entry in the row sequentially; a hardware cache checks all of the row entries simultaneously.

There is still a lot of overhead involved in fetching instructions, searching the DecICache and executing the instructions but there isn't much scope left for improving the performance of a sequence of instructions that is executed just once. We therefore need to further reduce the cost of code that is executed many times over, ie. program loops. As mentioned earlier, this is where most of the program's execution time lies, and therefore a good place to start applying optimisations. The obvious thing to do is to *remember* the predecoded instructions on the first iteration of the loop and then simple execute them on each subsequent iteration. This can be

achieved using a circular buffer:

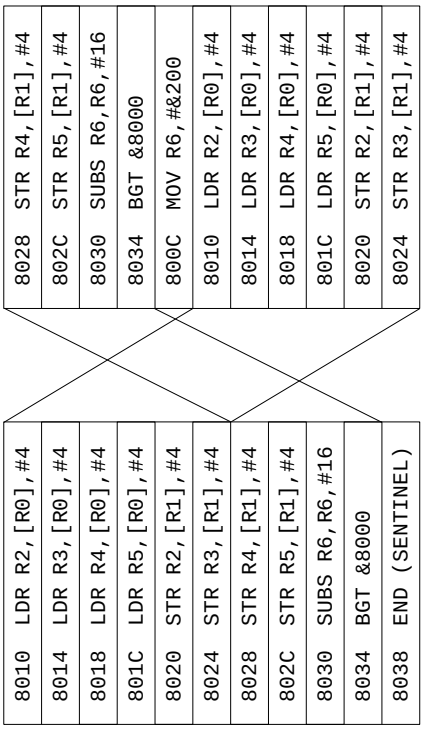


When executing sequential instructions, we decode the instruction into the circular buffer at the current position, and then advance the pointer, wrapping around at the end of the buffer. Then, when a backwards branch is encountered, if the target address of the branch is an instruction in our buffer, we can simply continue executing from the appropriate point in the buffer without needing to fetch and decode the instructions again, or even to search the DecICache.

Note that whilst this scheme is not strictly ARM610-compatible, because we are assuming that the instructions we have decoded into our buffer are *not* going to be changed by the execution of the loop, it will in fact run most code, even if the code is self-modifying because our buffer will typically be fairly small (<=64 instructions, say). Also, in practice, most self-modifying code actually modifies code that *follows* the current position (PC) rather than preceding it.

Even this simple optimisation provides quite a good speed increase because it catches the common case of short loops that are executed many times, eg. memory copying, string searching, and counting routines. It will not, however, work for more complex loops that consist of multiple code sequences, or loops that contain function calls.

However, there are a number of optimisations that can be used to improve even this simple scheme. Having now practically eliminated the cost of fetching and decoding instructions within program loops, it's worth concentrating upon reducing the execution cost. One way to do this is to store the decoded instructions sequentially in memory, by 'straightening out' the program loop when a backwards branch is encountered:



This avoids the need to check for pointers wrapping around at the end of the buffer. Without straightening out the code like this, a check would be required after each executed instruction so the cost of checking can be quite high. The code would still need to check whether the end of the decoded sequence has been reached, but this also can be removed by introducing what is known as a 'sentinel.' A sentinel is a special marker that is known to be present at the end of the sequence, and in this case would be a 'special' decoded instruction which, when 'executed', actually returns to normal, non-looping execution rather than changing the emulated CPU state in any way. Given the format of our predecoded instructions, this simply means setting the 'f' field of the sentinel to point to a special 'Exec_END' routine.

Execution of the next instruction in sequence is then simply a matter of loading its operands (a1-a3) and calling the routine, which can be done using just 4 instructions if we already have a pointer to the predecoded instruction in a register 'ib':

```
Exec_END   ; --- execute this instruction ---
           LDR a2,[a2]
           LDR a3,[a3]
           ADD ib,ib,#sz_INSTR                ;advance to next instruction
           ADD a2,a2,a3
           STR a2,[a1]

           ; ---  call the execution routine for the next instruction in the buffer ---
           LDR a1,[ib,#INSTR_a1]
           LDR a2,[ib,#INSTR_a2]
           LDR a3,[ib,#INSTR_a3]
           LDR pc,[ib,#INSTR_f]
```

Jumping directly to the execution routine for the next instruction, as opposed to returning from the Exec_ function to a dispatch loop, halves the number of pipeline stalls; unfortunately there's no way to avoid the pipeline stall that occurs when loading pc at the end of each routine.

Also note that, having eliminated the need to check our pointer (ib) after each instruction, we can now hold the emulated flags (N, Z, C and V) in the real 32-bit CPU flags which greatly simplifies the execution routines for instructions that use or alter the flags since there is no need to load the from/

store them to the emulated register set.

4.3 A Faster Interpreter

To increase the speed of non-trivial loops we need to resort to a more complex scheme. The approach used within Aemulor is to have a small cache of short circular buffers of the form described above. The cache is then indexed by some bits extracted from the PC whenever a non-sequential transfer of control occurs, eg. when a branch, procedure call or return from procedure is encountered. The interpreter then searches the cache of buffers (referred to as CodeSequences because each buffer holds a *sequence* of instructions) to see whether it has already decoded the code at that address. If so, the predecoded instructions are executed as described above. If not, the interpreter must choose a CodeSequence within the cache for replacement; this is done in a round-robin fashion, mimicking the way that a real ARM processor replaces its cache lines.

The interpreter then decodes instructions into the chosen CodeSequence until a non-sequential control transfer occurs (ie. an instruction explicitly sets the PC, in R15, rather than just advancing to the next instruction) whereupon the CodeSequence is straightened and the sentinel appended to speed up execution should this CodeSequence be executed again later.

There is a compromise required here between performance - which suggests a larger cache size - and compatibility with self-modifying code, which requires that we cache as little code as possible. Fortunately, because the StrongARM processor has now been around for more than five years, self-modifying code is now pretty rare in RISC OS applications, at least amongst those applications at which Aemulor is initially being targeted.

Readers may be interested to know that the pre-release version of Aemulor demonstrated when the IYONIX pc was released used only the techniques described up to this point and they have proved sufficient to run most applications at a perfectly usable speed. However, dramatic speed increases have since been obtained by using some hardware features that are unique to the XScale processor and not present on earlier ARM processors. They can only be used for running StrongARM-compatible applications, however, and the interpreter is kept as a fallback for applications that don't work with the hardware acceleration which I'll now describe.

4.4 Hardware Breakpoints

The Intel XScale contains some hardware debugging facilities that allow hardware breakpoints to be set on up to two instructions, ie. the address of an instruction is placed in a special register located in coprocessor 15 (there are two such registers), and when the XScale encounters the instruction at this address it will instead raise an exception, allowing an exception handler to take any appropriate action. Aemulor can therefore scan through the 26-bit application code from the current emulated position (PC) until it encounters an instruction that is *not* safe for native execution in a 32-bit mode because the behaviour would be different in the 26-bit mode that the application expects. A breakpoint is then placed on this instruction and an appropriate exception handler installed to emulate that instruction in software. This allow as much code as possible to run natively on the XScale at full speed, only dropping back to emulation where absolutely necessary.

There have been discussions on the comp.sys.acorn.* newsgroups about the possibility of dynamically patching 26-bit application code to run in a 32-bit mode, and the use of breakpoints is very similar in many respects but it has the considerable advantage of not needing to *alter* the 26-bit code itself; it is therefore - like the interpreter approach - still compatible with code that checks itself for modification or corruption, such as some copy protection schemes. Even more importantly, this scheme does not affect the XScale's caches, ie. there is no need to clean the data cache and/or flush the instruction cache to ensure that any patched instructions are visible to the XScale core.

By careful coding it is possible to greatly reduce the cost of those instructions which *do*

require emulation and to quickly set the breakpoint at the next position before returning to native execution. To this end Aemulor uses a cache that is indexed by the current emulated PC and returns the address at which the breakpoint should next be set. The exception handler is thus able to quickly emulate the current, breakpointed instruction, move the breakpoint to the next location and install the exception handler appropriate to the instruction at that location before resuming native execution.

Specialised execution handlers are used for emulating those instructions which occur most often such as BL (Branch with Link) which is used to call a procedure/function, and MOV(S) PC,R14 and LDMFD R13!,{...PC}(^) which are used to return to the caller.

I think it is unlikely that this approach can be bettered (in performance terms) in software without resorting to some form of compilation from 26-bit to 32-bit code, though it is always dangerous to make claims about as yet unknown alternative approaches!

4.5 Just In Time Compilation

The use of hardware breakpoints gives very good performance but still has some limitations, notably:

(i) Emulating instructions in software is necessarily somewhat slower than performing the equivalent operation in native code, even if to do so would require multiple instructions, as is now the case (eg. BL must merge the PSR - flags, interrupt state and mode bits - into R14 - for compatibility with the 26-bit behaviour of this instruction).

Whilst this operation could be performed in as few as 4 or 5 instructions natively, it takes much longer to enter, execute and return from the exception handler that emulates a BL instruction when using breakpoints.

(ii) The 26-bit code being executed predates the XScale and most of it predates the StrongARM. In fact much of it was written with an ARM2 or ARM3 processor in mind and these exhibit very different performance characteristics to the XScale CPU. In fact even the latest versions of the RISC OS C compiler do not optimise for the XScale CPU, and yet the XScale - much more so than earlier ARM processors - has many complex rules about how to order instructions so as to keep the pipeline filled and running smoothly to attain maximum performance.

So even running this 26-bit code natively on the XScale will not achieve the maximum peformance of which the XScale is capable. Breakpointing is unable to properly address this issue because to do so would mean changing the 26-bit code and thus introducing the need for cache flush. Plus it would probably reduce compatibility with some applications.

A compiler would take the original 26-bit code and produce a section of 32-bit code that can be run natively on the XScale to produce the same end result. This is analogous to a C compiler converting C source code into ARM machine code, though in a very much simpler form, particularly since the 26-bit and 32-bit ARM architectures are nearly identical.

The term 'Just in Time' (JIT) as used to describe such a compiler refers to the fact that the compiler only compiles a (relatively) small section of code immediately prior to it being needed, rather than trying to compile the entire program in advanced before running anything.

The output code of a JIT compiler can be appreciably faster than emulating the 26-bit code, even if the XScale's hardware breakpoints are used and only the '26-/32-bit different' instructions are emulated, *if* such instructions occur frequently in a small loop that is executed many times. In this case the overhead of emulation will be quite large, and it becomes beneficial to remove it by compilation.

A JIT compiler can also take the opportunity to optimise the code for execution on the XScale whilst it is compiling it, eg. by reordering instructions or replacing slow instructions with a sequence of short instructions which - perhaps counter-intuitively - can often execute faster on the XScale than the single instruction would. This is because the XScale core is actually more RISC-like than earlier ARM processors, and it decomposes complex instructions such as LDM/STM into sequences of simple instructions (LDR/STR).

However, compiling to native code is expensive on an ARM/XScale processor because changes made to an address via the data cache do not affect any instructions already read from these addresses and stored in the instruction cache. It is thus necessary to ensure that the newly-generated code has been written out to memory (referred to as *cleaning the data cache*) and the old code, if present, removed from the instruction cache (referred to as *flushing*). Furthermore, on ARM/XScale processors it is only possible to flush the entire instruction cache, not just specific addresses.

So an emulator must be 'certain' that the cost of compilation, writing the changes to memory and subsequent reloading of the instruction cache will be offset by the speed gains of running optimised native code. For this reason the JIT is likely to be beneficial only on sections of code that are executed many times, for example an image processing function applied to a large image. It is unlikely that a JIT would give good performance if used to execute *all* 26-bit application/module code and was not backed up by an interpreter of some sort.

For these reasons, effort was concentrated upon speeding up the interpreter and breakpointing phases as much as possible, and the first release of Aemulor will not include a JIT phase. Early experiments have shown that deciding which sections of code should be compiled for better performance is a difficult task that is likely to require some fairly sophisticated analysis.

5. Future directions

5.1 Support for < 256-colour screen modes

One of the most-requested 26-bit applications is Sibelius 7 which uses a 4-colour screen mode. Screen modes with *fewer than* 256 colours are not available on the IYONIX pc because it uses a standard PCI graphics card from NVidia that does not provide these screen modes.

A possible solution would be to provide emulation of these screen modes within Aemulor, and Aemulor would convert between the emulated screen and the real screen provided by the graphics card. This would, of course, be transparent to the user so that Sibelius would appear exactly as it does on the RiscPC. The feasibility of this approach has not yet been investigated but it is hoped that it can be introduced with Aemulor Pro which should then be capable of running Sibelius. It will *not* be present in the first release of Aemulor.

5.2 JIT compiler

An early version of Aemulor did include a simple JIT compiler, but this has since been greatly outperformed by the interpreter and use of hardware breakpoints. It has therefore been dropped pending further exploration. After more research and experimentation a JIT compilation phase may be introduced in Aemulor Pro, if it is proven to give a beneficial speed increase. It's possible that compilation would be used to target specific areas of popular 26-bit applications that are found to run slowly under Aemulor.

5.3 Application-specific enhancements

Since Aemulor is effectively in control of everything that the 26-bit application does, there is always the option to analyse exactly what a specific application is doing, and thus improve the performance

of any areas that are found to execute too slowly. Another possibility is that Aemulor could *remember* which optimisations have yielded good performance enhancements when an application was used previously, and thus quickly reuse these optimisations rather than waste time analysing the 26-bit code again. Performance would then improve on successive runs of the application as Aemulor learned which optimisations work and which do not.

All of these ideas are speculative at this stage, of course. Which, if any, are incorporated into Aemulor, or its successor Aemulor Pro which we intend to release later at a higher price, will depend upon the results of our testing. The first release version of Aemulor already offers performance equivalent to a 202MHz StrongARM RiscPC, judged both by the feel of applications such as Impression and StrongED, and by the timings of the computationally-intensive rendering code in ArtWorks. As Aemulor is developed this performance will undoubtedly increase further.

Finally, as a point of interest, this article was written using Impression Style running on an IYONIX pc with a pre-release version of Aemulor.