Development diary

This diary was kept mainly for personal reference and for the perusal of helpful people around the world in the hope that their insights to my way of thinking might save me time and trouble. It also provides some entertainment to look back on how little I knew at the beginning. This diary comes with a health warning, it has been left as it was written, often late at night and over many months. It should provide interesting reading to anyone else approaching a similar project to see how this one progressed. Note, the entries are in reverse-chronological order, the most recent first.

13/3/01
Project presentation day.

12/3/01
Completed code generation of all instructions necessary to demonstrate the GCD example. Worked on register allocation problem so that allocations can be exchanged between basic blocks if needed. Work still necessary on combining spilling of constant pool to code at the end of basic blocks and spilling allocated regs to memory since their may be some redundancy there and by the code co-operating this can be reduced. Final preperations for the presentation.

11/3/01
Incredible progress with x86 code generation. Instruction selection done using multiple if..then statements for ultimate flexibility. Implemented the add armlet completely and it worked. Followed this up by implementing most of the features of the constant pool to allow constant evaluation at compile time, this involved evaluating the movc and mov armlets. Added support for the cmp armlet including the awkward inverted carry problem by suffixing it with cmc to invert the carry flag back to ARM format (if SBC were used then it would be silly to have two cmc's next to each other so work will be needed here). Added support for all goto armlets and backpatching which works correctly after a small issue where I had to find out that the x86 PC points to the instruction after the one being executed (according to objdump's disassembly) so all jmp instructions were off by 6 bytes (the number of bytes taken up by the jmp instruction). I've initially just used far jmps (32 bit offsets) for simplicity though most of the time only 8 bit jumps are required (since chunks are so small), some adjustment here would probably be fairly simple though near jmps take fewer bytes than far jmps which may cause complications for backpatching. Implemented part of the sub armlet's generation needed for the GCD program. Linked the objdump execution into the program using system command to call the DOS program.

10/3/01
Restructured the object construction so that it was as clean as my design said it should be and so that errors that appeared yesterday are gone forever. Spent the rest of the day preparing presentation slides and handouts. The presentation preperation is mostly complete now, with just the final finishing touches left to be done.

9/3/01
Progressed with the generation and execution of machine code. Generated several instructions and was able to manipulate data in a memory block and return to C

without error. Then progressed to call a C function from the generated machine code and to be able to return a value from the machine code to C at the end of the execution of the generated code. Constructed a simple CodeCache structure, and other small parts of the framework. I had hoped to make good progress with the real armlet to x86 translation but was dogged with small but incredibly frustrating idiosyncratic errors. Finally worked round them all by the end of the day.

8/3/01
Tested and validated the basic block detection and the redundant flag calculation elimination. Fixed issues with the conditional block optimisation that had been created by the adjustments made introducing INTCHECK. Continued to develop the framework, specifiying native chunks, and amalgamating Julian brown's x86 asm generator (which is basically a list of macros defining instruction encodings) into my code though there are some initial difficulties.

7/3/01
Tidied up updating of PC before leaving chunk. Tidied up generation of IOCLOCK and SIGNALS so that they are now just one armlet, INTCHECK. INTCHECK takes an argument which is the number of IO clocks to perform and checks for interrupts returning a non-zero value if an interrupt occurred. Verified that the recompiler in the profiler works if given an address to recompile from and exits correctly. Implemented the framework for the optimiser and generator and am pressing on with the generator since optimisations can be easily added later and I'd ideally like to have the generator mostly working in time for the presentation. Tested the objdump x86 disassembler that I extracted from GCC and verified that it can disassemble an object file passed to it. Experimented with RS on a friend's laptop in preperation for the presentation, as a result of the limited speed, poor display refresh and awkward touch-pad mouse I have opted to use my desktop PC. Started coding the rule decision making for the generator. Converted the linked list of armlets into a buffer (to give indexed access) then added basic block determination to the optimiser to identify the leaders in the chunk. Wrote first draft of redundant flag removal which simply removes any outflags settings which are overwritten by a later armlets outflags settings without any inflags requirements inbetween. This needs testing.

6/3/01
Solved the rather nasty problem with coalesced branches – very complex but now works perfectly, good optimisation as well. Enquired after the Dhrystone benchmarking that Julian did (he will be able to send it to me in a few weeks) and got in contact with Mohsen about ARMIndex from ArmSI4 (unreleased). Met with Graham, discussed report contents and presentation style.

5/3/01
Added flag information generation to IR generator. Demonstrated to myself the it translated code ok and fixed some bugs in generation. Dealth with the problem of TEQP instructions which had been omited before, these directly write to the ARM PSR and are awkward in amongst the otherwise well behaved data processing instructions. Hit problem with coalescing the conditional execution that was not anticipated in that a branch instruction could go back to part way through the coalesced block and would need conditional execution code there to handle it. If several such branches existed then it would be nasty patching the code to cope with

them, this will be attempted tomorrow. Spoke to Sara concerning register allocation techniques, she confirmed my concerns about the standard compiler techniques and recommended that I keep it simple, avoid static register allocation (because of the limited number of registers available) and either use the linear scan method if it seems useful or got for a simple spill a register with single instruction lookahead. Wrote draft report contents to discuss with Graham.

4/3/01
Spent most of today adding the block data transfer instruction translation from ARM to armlets. This was very, very, complex – particular issues involve what to do with exceptions (drop back to dispatcher), what to do with instructions that are forced to operate on the user mode register bank (drop back to dispatcher) not to mention all the possible varieties of code to be generated. Added the coprocessor instruction translations but they all drop back to the dispatcher since embedding coprocessor emulation is not really sensible because a) it's complex b) it wouldn't be easily extendible c) it would lead to code bloat. This completes the actual translation from ARM to armlets, the next phase is optimisation and x86 generation (which should be a laugh).

3/3/01
Wrote code to translate single data transfer instructions, the SWP instruction, and branch instructions. Also rewrote the SWI instruction so that rather than attempting to embed mode-switching into the recompiled code it sets the PC to point to the instruction and leaves, signalling the reason for leaving as being a SWI. On first inspection this seems a little silly, why recompile the SWI code at all, it would surely be better if the instruction before the SWI performed a lookahead and saw that the SWI was not to be recompiled, so ended the chunk there. However, the SWI could be conditionally executed which would mean the chunk was ended unnecessarily. Also the STM/LDM block data transfer instructions can temporarily force a mode change when they access the user mode register bank from other modes, this is not really sensible to determine from lookahead (as it would slow down lookahead which would need to be done for every instruction) and so would need to leave partway through the instruction. Hence the reason that the SWI instruction generates the code to leave.

2/3/01
Discovered that the armlets being mistranslated were actually being translated fine (if inappropriately) but the armlet disassembler wasn't handling them properly. Developed armlet disassembler to handle all opcode and operand types. Tested the profiler with sequential instructions with the same conditional execution code and with a slight adjustment that worked fine as implemented previously. Backpatching of gotos for conditional execution works fine. Implemented translation of the ARM branch instruction to armlets. The GCD program is now converted perfectly into armlets.

1/3/01
Back on the project having completed the current uses essays this afternoon. Spent the intervening time reading up on compiler optimisations, instruction selection and register allocation issues from compiler literature.

Started testing framework for IR generation. Using the TestMemory class loaded the GCD calculating program and translated the first instruction. Commenced work on the framework of the armlet disassembler and evidenced that the first instruction of the GCD program had been successfully translated though there seems to be a lot of other armlets that have been mistranslated, some investigation will be needed into what is happening.

28/2/01
Have been reading up on register allocation algorithms and instruction selection (maximal munch). Decided to talk to Sara concerning register allocation, graph colouring seems excessive but the linear scan method looks possible if I go down the dynamic route.

21/2/01
Spent some time considering optimisations on the IR.
Profiling usage of ARM regs, r0, r1 used a lot, r14 r15 used most.
Profiling interpreter to see what's used most – generic speed & hints for dynarec. Unable to get working.

20/2/01
Spent some time considering my options as regards exceptions, in light of the complexity of the SWI instruction and the impending problems from adddress exceptions and data aborts. Following much discussion with Graeme Barnes about the issues arising from my armlet design, it seems sensible to handle all mode changes (and therefore exceptions) from C rather than trying to put massive amounts of tedious and seldom used processing into generated code. Still pondering explicitly fetching ARM regs into armlet variables, this is as yet undecided but doesn't affect development much - decisions about x86 generation are bound to influence this.
Met with Graham – quantity not quality. Important to descibe what my project does that hasn't been done before.

19/2/01
Implemented the armlet generation for MUL/MLA and SWI, started looking at SWP but paused to consider the complexities of memory access.
When I realised the complexity of recompiling the SWI exception to change modes, swapping multiple registers etc. there was a temptation to drop back to the dispatcher to handle this. However, it seems necessary to leave the IR with the ARM details in a stable state and not completing an instruction after recompilation has been started (e.g. IOCLOCK emitted) could cause plenty of potential problems (not to mention being a cop out).
Started the shell of the armlet disassembler though there is little point in continuing with it until more of the IR generator is complete so that the IR is concrete. Otherwise I could find the armlet disassembler goalposts being moved by variations in the IR.
The next thing to do is to evaluate all the optimisations I have in mind to decide exactly which are to be implemented given the limited time available and then translate the linked list of armlets to the final storage state. Once this is done and more armlet generation is implemented a start can be made on the armlet interpreter for debugging the armlet generation.

18/2/01

Spent a lot of time writing the generation of code for data processing instructions and inparticular the barrel shifter which I had anticipated being a problem but was a lot worse than I ever imagined. The code has been written and now needs to be tested in detail.
Refined parts of the design of the IR, particularly in relation to adjustments to the PC and modes, largely shifting the responsibility onto the dispatching algorithms. Now that the barrel shifting is dealt with most of the rest of the translation should be fairly straightforward if timeconsuming, with the exception of LDM^.
Performed some simple dynamic profiling to discover the frequency of the different types of exceptions. It appears that though there are a cluster of data aborts early on in execution (I believe in the POST), following that nearly all exceptions are either IRQs (presumably from mouse pointer movement) or SWIs (about 5 times as many SWIs as IRQs). These two exceptions occur fairly regularly, around every 1000 instruction executions, and certainly not more than about 10000 executions apart. There is the occasional undefined instruction which I suspect is probably the floating point emulator being used (when drawing the bezier curves again in Draw).

17/2/01
Progress made with IR generation, many more problems have come to light such as the complexity of the LDM^ operation (particular regarding exceptions) in that it doesn't fit in nicely with the IR design and I will need to add some kind of meta control statements, or allow armlet instructions to be used to not affect ARM regs/flags. Also the memory access problem from exceptions needs handling so as to spill regs/flags and throw the exception, in the event of memory failure (which will occur once in a blue moon).
Performed some brief dynamic profiling of real code today and found to my amazement that SVC mode is used for drastically more time than any other mode. See stats in IR design doc.

16/2/01
Developed the framework to deal with coalescing blocks of sequential instructions with the same conditional execution code and the backpatching to deal with conditional execution in general. Started the development of the decoding and the emitting of armlets to the linked list. Developed the TestMemory class, something that would have been useful for testing the interpreter and disassembler which just gives access through a common interface to flat model memory rather than memory through MEMC.

15/2/01
Wasted large amounts of the day looking into the problems of hash tables and linked lists for IR generation. Developed my own hash but have finally decided to plump for STL maps for hash tables for the moment returning to my own implementation if they prove inefficient. STL linked lists appear to be more challenging and possibly more expensive, I will therefore probably work with my own implementation which seems to operate fine.

14/2/01
Began work on the implementation of the IR generation. Understood basic algorithm and investigated different ways of backpatching the internal jumps in the IR. Devised an algorithm so that sequential instructions with the same conditional execution code

(that don't affect the condition flags) can all use just one check of the condition on the first instruction rather than unnecessarily repeatig it (as happens in ARMphetamine's IR). This is added into the code generation rather than as a post-generation optimisation. Needs real testing.
Spent a lot of time looking at how to call machine code from C++, using either inlined assembly to call the buffer address, or casting the buffer containing the machine code to a C function which can then be called as any other C function. The former is preferred since it's more explicit but may prove nastier once parameters are involved. Have so far failed to call a C function back from the machine code, but using CALL with a register containing the function address (rather than an immediate encoded offset to a function) is sure to fix this.

13/2/01
Started to investigate calling C functions from asm
Under the interpreter, used the command line interface with the filer_run command to load paint/draw/configure without having to use the filer which is suffering from bugs.
Eliminated need for updating r15 in block data transfer instructions which allows a single valaddr armlet to throw and address exception if an address is invalid
Met Graham
Spoke to Sara about IR being closer/further from x86 i.e. allow r0=r1+r2 or just r0=r0+r1. Recommendation that I should keep it more general, in the style of compiler IR's, this allows optimisations and is the more general case, also allows sensible portability. This problem is normally tackled at code generation.
Experimented with calling C functions from inlined asm and also C++ functions with MSVC's name mangling

12/2/01
Lost several days to 4000 word CU essay
More design of IR
More discussion with GB

6/2/01
meeting with sara – ARM spec from Leeds/Cam, heuristic identification of loops/if-then-else, optimisations separate, keep it simple – drop back to interpreter
speed comparison with RS – release build win2k, RS 2.74 MIPS my interpreter 2.36 MIPS at rest.
DebugOpt my interpreter 2.24,
GB's idea of limiting recompiling to a page, or dynamic-destination branch so as to have larger recompiled areas, not have problems with pages changing, make it easier to dump sections of code
Started implementation of dispatcher and profiler classes

5/2/01
meeting with graham – for next week do dispatcher and profiler and IR generation completely.
considering hashes
considering LRU algorithm

4/2/01

emails with dynarec group, Julian's comment "wow"

3/2/01

The day started badly when it became apparent that the emulation had outgrown my preferred method of debugging. Running my interpreter several times rarely produced the same result and a similar phenomenon was found with the red squirrel interpreter. I expected this to occur due to some kind of timing mechanism in the rest of the RS system. For example, a timer could be started to tick every second updating parts of RS. When the call to IOC.clock() is made from the interpreter this update would affect the execution of ARM instructions. Due to the way timing works, the timer would rarely tick at exactly the same number of instructions into the execution each time the emulator is run. This results in the direct comparison of trace dumps no longer being a possibility for debugging since differences no longer constitute bugs. This is not a distaster since debugging is still possible by intensive analysis of the results though this was expected to slow progress dramatically.

Returning to the exception on the coprocessor instruction (noticed last night) trace outputs quickly showed that this was happening around the 16 millionth instruction execution when the ARM was attempting to read from a register of coprocessor #1. This coprocessor had been initialised in the interpreter's constructor method to point to a dummy coprocessor. Accidentally the dummy coprocessor was only locally defined in the constructor rather than as a class member and hence when the constructor finished, the dummy coprocessor object was destroyed, causing the exception when it was accessed. The fact that this went undetected is not surprising, coprocessors accesses are extremely rare under RISC OS. Upon fixing this and testing again I was more than a little surprised to see RISC OS boot to the desktop!

Brief tests show that things are not perfect, disk access appears to be broken (in the desktop at least, though command line access seems fine) and BASIC has some bugs. However the command line, task manager, menus, fonts and general desktop seems fine which is a major achievement. In terms of the milestones laid down in the progress report, this achieves the second of six development tasks and three of the four compatability measures. The next step is development of the recompiler to intermediate representation and possibly some more specification of milestones, since I feel the milestones I laid down in the progrss report are a little far apart to say the least.

Updated the website.

2/2/01

Pressed forward with an attempt to make some real progress with the debugging. Added a parameter file to my interpreter and the one from red squirrel so that recompilation between tests was no longer necessary, which speeds up the rate of testing.

This new found efficiency resulted in a lot of progress using the binary search across the execution space: Fixed a bug where the MCR instruction was always going to throw an undefined instruction trap whether the coprocessor being accessed is present or not (coprocessor instructions had not been tested upto this point). Fixed another bug where interrupts were not being correctly triggered. Fixed a bug in the interface to Red Squirrel so that when the program is quit it didn't hang. To give an example of the complexities of the debugging, here is the final bug found today.

My interpreter's incorrect results:

```
0x3811750: TEQP       PC,#3 - 0xeae044ca, 0x0, 0x3200000, 0x8000,
0x3800001, 0xcd00, 0x0, 0x17d00, 0x8000, 0x36e010c, 0x40, 0x0,
0x3200000, 0x1f0136c, 0x0, 0x3811754, : cond=0x0 I=0x8000000 F=0x0
line=6531120
0x1c: B            &381134c - 0xeae044ca, 0x0, 0x3200000, 0x8000,
0x3800001, 0xcd00, 0x0, 0x17d00, 0x8000, 0x36e010c, 0x40, 0x0,
0x3200000, 0x1f0136c, 0x0, 0x20, : cond=0x0 I=0x8000000 F=0x0
line=6531121
```

Red Squirrel's correct results:

```
0x3811750: TEQP       PC,#3 - 0xeae044ca, 0x0, 0x3200000, 0x8000,
0x3800001, 0xcd00, 0x0, 0x17d00, 0x8000, 0x36e010c, 0x40, 0x0,
0x3200000, 0x1f0136c, 0x0, 0x3811754, : cond=0x0 I=0x8000000 F=0x0
line=6531120
0x1c: B            &381134c - 0xeae044ca, 0x0, 0x3200000, 0x8000,
0x3800001, 0xcd00, 0x0, 0x17d00, 0x8000, 0x36e010c, 0x40, 0x0,
0x3200000, 0x1f0136c, 0x3811757, 0x20, : cond=0x0 I=0x8000000 F=0x0
line=6531121
```

The difference is on the penultimate line of each interpreter's output. The value for r14 should be 0x3811757 after the TEQP PC,#3 instruction, but in mine was left as 0x0. TEQP performs an exclusive-or on the two parameters, in this case the PC and the number 3, then updates the PSR with the result. TEQP itself was operating correctly, when the PSR was updated, the mode flags were changed to supervisor mode. After that instruction, an IRQ exception occurred. This is not visible on the output but obvious since the PC (at the start of the line) for the B instruction is 0x1C which is the IRQ vector (0x18) + 4 (taking account of pipelining). Examination of the IRQ exception implementation then ensued and it was found that when the IRQ exception changed the mode to IRQ mode, it incorrectly updated r14 with the flags taken from after this mode change when it should have been from before this mode change.

The reader should find it hard to discern exactly what I have described here and this is indicative of the problems I had in locating the problem. This only goes to highlight the unusual complexities and detective skills needed to debug this sort of software.

This last bug fix led to a breakthrough of sorts when the cursor that occurs immediately before the RISC OS memory prompt appeared (it should be noted that the emulator then crashes out with an unhandled exception on a coprocessor instruction).

1/2/01

Spent some time assessing progress made so far in terms of the milestones laid out in the progress report. Also looked at the dependencies between them in light of the delays in success with the interpreter. Spent some time considering my designs for the dispatcher and profiler and am fairly happy with the framework as it is laid out. I need to do a little research on fast hashing functions according to the characteristics of the addresses of code to be generated. Also some research on limiting the size of the cached code buffer so that it doesn't become too large, ideally some kind least-recently-used algorithm from conventional processor cache technology seems suitable, how this can be implemented in software is another matter.

Note, lost the last few days to AI coursework.

28/1/01

Further execution and debugging has got me as far as validated upto instruction execution 4,694,588. The screen now correctly turns from pink to blue and then to black but the RISC OS cursor and memory display doesn't appear. The binary chop method across the execution space is now likely to be the best way forward.

27/1/01

Early on in the day it became apparent that disassembling from a given address onwards was not a satisfactory way of invoking the disassembly since the early parts of RISC OS consist of some loops which run for many interations and thus later stages of the looping could not be disassembled. Instead I implemented an executionCount (initally used the instr_count variable to be compliant with RS variable but it is unsuitable as it's reset every second when updating the MIPS display on the RS status bar) incrementing every instruction, so that the currently executing instruction has a unique number. This worked sufficiently well to dump and compare many hundreds of thousands of instructions.

One major bug was identified at instruction number 981244 where it seemed that an STRB r2[r2,r2] instruction was causing a data abort exception, this would occur if the byte could not be written to memory. Having confirmed the address and value were ok, it seemed likely there was an issue with the STRB instruction. A dummy program was written on the Risc PC (in a similar way to the previous LDM/STM test) and run on both, working perfectly. Some minor analysis went into the RS MEMC class to see what the issue was. Eventually this was narrowed down to a double negation on the returned value from MEMC (when it should have only been a single one) which was masked by my accessor method, used to make my interpreter independent of RS's MEMC class. The dummy program did not catch this since the issue was directly with the MEMC interface which is not used when flat-model memory is being used.

If bugs become less frequent (at the moment I seem to be finding one every few 100,000 instructions executed) I propse to use a binary search over the execution space (by execution count number). This idea is based on the assumption that an incorrect instruction execution will corrupt (at least in some small way) all subsequent execution – this has been my experience in all the bugs found so far. I can therefore search larger amounts of code faster using a binary search rather than the linear search (albeit not comparing every instruction) employed so far.

At the end of the day have validated up to instruction execution 1,478,461 and tomorrow will execute to beyond the tight loop which is being executed repeatedly at that point.

26/1/01

Uncertainty over the effectiveness of the multiple data transfer instructions led me to write a simple program, tested on the Risc PC which I then ran on the interpreter to find that it worked fine.

Set up Red Squirrel's interpreter and my own so that for each instuction executed, it's address, disassembled instruction, and all registers and condition flags are output to a text file. Each version is then run as if the RISC OS computer were just turned on (i.e. from the same state that the emulator should be started from) and the output data collected, normally in the order of 10 million instructions at a time. The two files are then run through a comparison program which highlights any differences, these differences are taken to be errors in my interpreter. With much detective work involving; looking at the routines used to emulate the rogue instruction, the desired

result and the state of the emulated processor before and after it's execution, the source of the problems can be tracked down.
Several bugs have been found in this way, such as the fact that the interrupt disable flags should both be set on start up. This was only detected when the result of an add instruction taking r15 (the PC and PSR) as an operand was slightly incorrect. It was noted that the bits not being set in the result were the ones for the interrupt disable flags and from their the correction was straightforward.
Having executed several tens of millions of instructions in this way, I finish today having verified the interpreter correct upto the following loop of code:

```
0x380205c: LDMIA   R6!,{R2,R3}
0x3802060: EOR     R2,R2,R4
0x3802064: ORR     R0,R0,R2
0x3802068: EOR     R3,R3,R5
0x380206c: ORR     R0,R0,R2
0x3802070: ADDS    R5,R5,R5
0x3802074: MVN     R4,R5
0x3802078: BCC     &380205c
```

25/1/01
Compared the operation of about 90% (4000+ lines) of the interpreter code with that of Red Squirrel which is known to work. Many bugs were discovered, generally caused by small omissions or typos, though RISC OS still does not boot. Some progress has been made by disassembling each instruction as it is executed and manually looking for discrepancies. This will continue tomorrow on a smaller scale, possibly using Red Squirrel's built in debugger to step through the code instruction by instruction. Another possible solution is to try is to run the complete Red Squirrel and output a disassembly of each instruction executed and compare this to the same output for my interpreter. This is likely to be a good way to highlight issues
The delay in progress is a cause of much frustration, however there are still many avenues to exhaust and just one single correction could lead to a break through. The hope is that with an accurate interpreter complete, such tedious testing will not be necessary for much of the dynamic recompiler, since it's execution could be directly compared to the reference interpreter for differences.

24/1/01
Debugged the disassembler by running it on a large program, dumping the output to a file and then manually comparing it to a known good disassembler for discrepancies, this highlighted several problems but it seems very good now. Coprocessor instructions are still not supported because of time constraints but I do not anticipate this to be a problem as they are not likely to be the study of intimate debugging because of the way they are implemented.
Having examined the initial start up of RISC OS using the disassembler in conjunction with GB's rough notes on the purpose of different sections of code, I feel that, as expected, more systematic testing of the interpreter is required. Although I could plough on with the later stages, as stated before it is important that I have a solid model to reference when developing the IR and code generation.
Read through the articles in Computer which proved interesting but didn't highlight anything particularly new.
Also read up on Julian Brown's recent adjustments to his phetacode (the intermediate representation for his ARMphetamine project) in which he had developed some of the

things I was intending to implement, especially adding instructions to make the emulated flag handling more explicit.

23/1/01

Attempted to integrate my interpreter into RedSquirrel.

Removed SWI (Software Interrupt) 'faking' which was being used to test simple programs. This is where OS calls (SWIs) are intercepted by the emulator and simple commands are imitated rather than emulated, for example OS_NewLine is known to output a new line character to the output stream so this can be done without the need for complete hardware simulation.

One notable moment was when I first removed GB's CArm class for his ARM interpreter, renamed my CArmInterpreter class to CArm and tried to build Red Squirrel, this generated 4184 errors, a personal best. Note, this was mainly due to typedef's and accessor methods which hadn't been made use of in my class but had been used everywhere else in RS, and were declared in the old CArm header file. After some struggles to compile the code and a lot of accessor methods having to be created to interface with the Red Squirrel debugging components and removal of the test wrapping code I had created, the emulator was seen to run. The display changed to pink which is the first part of the RISC OS POST (Power On Self Test). Single stepping will be needed, as well as instruction by instruction verification with the manuals and a return to the simple test interface in order to find the errors. This could prove time consuming.

Received the ARM3 data sheet from ARM which means that I now have a genuine reference source as opposed to inferring details from later data sheets and manuals, as good as this has been. Downloaded several Java JIT research papers concerning fast recompilation and some articles from 'Computer' concerning binary translation.

22/1/01

Before attempting running RISC OS performed some further testing on the interpreter by running the divide routine used in the ARMphetamine report as an example, it worked fine first time. Borrowed and implemented the interrupt signalling code from Red Squirrel so that other red squirrel components can signal for the ARM processor to generate an interrupt.

Spent some time examining the assembly generated by MSVC++ to check on the quality of code it is generating before I leave the interpreter. When starting the interpreter I took the decision to use C++ inlined methods rather than C #define macros for instruction templates (to give stronger type-checking and better programming style). In particular I wanted to check that it was correctly inlining the instruction template code since this could lead to massive overheads if not performed and would have resulted in me having to convert my validated functions to macros.

An example of this is where the getField() method has clearly been inlined in this section of code for decoding the instruction with the `'shr'` (shift right) and `'and'` instructions.

```
; 173  :          // decode instruction type from bits 20-27
; 174  :          switch( getField(currentInstruction, 20, 27) )
; 175  :          {

      mov   eax, ebx
      shr   eax, 20                       ; 00000014H
      and   eax, 255                      ; 000000ffH
```

```
      cmp   eax, 255                        ; 000000ffH
      ja    $L48016
      jmp   DWORD PTR $L51516[eax*4]
$L47308:
```

In some cases, simple functions were not being inlined as a result of the compiler making a bad judgement as to when not to stop inlining. Judicious use of the __forceinline keyword has been used on small functions to force the compiler to inline them. Though this has little bearing on the dynamic recompiler, it is important to give some thought to the interpreter's performance in order that later evaluation using speed comparisons is as fair as possible.
Interfaced the disassembler to the interpreter so that instructions executed can be viewed as assembly rather than hexadecimal values, necessary for testing anything but the shortest of programs. This highlighted several issues with the disassembler, several of which have been fixed. Tomorrow I will attempt to run RISC OS and check for bugs in the interpreter.

21/1/01
Implemented all exceptions and filled in the gaps where I'd left them out previously, this includes interrupt handling. Completed all the semantics in for the decoding switch table, including many logic, arithmetic and coprocessor instructions. Added support for adjustments to the TRANS line which affects the MMU's logical to physical translation so that supervisor mode instructions can access memory as if it were being done in user mode.
Implemented rudimentary coprocessor instruction support to work with red squirrel's coprocessor emulation. Complete coprocessor emulation is not something I feel this project should delve into for various reasons a) it doesn't add anything interesting to the dynamic recompilation b) it adds more work to the project without adding any real functionality c) it is highly under-utilised as a feature of the ARM architecture in many systems.

19/1/01
Implemented Single Data Swap instructions for both byte and word, filled in several more entries to the switch table and wrote a few more template logic instructions.

18/1/01
Completed single data transfer instructions. Implemented block data transfer instructions. Need to look at Red Squirrel's fixes for LDM (with S bit set) in the manuals and other emulators. These fixes were discovered from issues when running parts of RISC OS which apparently conflicted with the manuals.

17/1/01
Succesfully ran the first program on the interpreter, printing out rows of 1,2,3,4 and 5 stars to the debug output. This tested the CMP, ADD, MOV, B instructions as well as conditional execution for the LE (less than or equal to) condition. The next things to add are single and multiple data stransfer instructions and fill in the rest of the switch table. Now that the basics are working and I have proven that the fetch execute loop works I also want to finish off and tie in the disassembler written previously to aid debugging of longer programs. It should be noted that although it may seem sound trivial to get this far, many of the complex templates needed for every instruction are

now working (even if not yet rigorously tested), as well as the pipelining effects on prefetching instructions, which is something of a headache.
The first run program was quickly followed by a GCD test which worked first time and additionally tested the condition codes GT, LT and NE.
Initial work was done on adding support for most of the single data transfer instructions as well as templates for the various permutations of these instructions (0x40 – 0x6F complete).

16/1/01
Read up on and correctly implemented all the variations of accessing register 15 depeding on pipelining and access of the PSR, comparing Red Squirrel and the ARM ARM for intricate implementation details, this is extremely complex and time consuming. Implemented several logic and arithmetic instructions including MUL and MLA as well as branch, and inparticular improved the access to and generic methods for updating the condition flags for instructions which need them. To test progress I took a simple program to output lines of 1,2,3... asterisk characters to the display and implemented simple SWI faking (to handle character output and new line). This lead to the development of the simple test wrapping, the memory interface being abstracted within the ArmInterpreter class so that program is not inherently reliant on Red Squirrel's MEMC class (which is too complex for initial testing), and some flat-model memory with a file loaded to load raw executables into the interpreter. Some instructions have been executed but there are issues with the incrementing of the PC from one instruction to the next, I suspect related to the pipelining effects.
Looking at the ARM ARM for the MUL instruction (section 3-58), it seems to state that if r15 is used as any of the arguments the result is unpredictable. It therefore seems probable that all checks for whether the PSR should be fetched and the +4 added for pipelining effects if r15 is an argument could be removed, (also whether rd == rm in MULs) as it won't ever occur in real world code. In the event that it did the result could not be called inaccurate since it's unpredictable. On consulting Graeme Barnes he seemed to think this was fair and also put paid to my doubts that some pathological case like the RISC OS POST (Power On Self Test) might indeed test these boundary cases and gave me a copy of his rought research on the POST. Haven't yet taken account of the affects on PSR flags from the barrel shifter being used to represent immediate constants – this must not be forgotten.

15/1/01
Met with Graham for the first meeting of the term. Briefly discussed the scheduling of the project and the viability of the threaded interpreter working with the intermediate representation as a satisfactory conclusion to the project in the event of difficulties in completely the translation to native code. The thinking being that this is essentially the ultimate desired conclusion but doesn't really add anything in terms of issues and methods to the threaded interpreter except a lot more work. In the meantime I will continue to follow the project schedule as closely as possible.

Finished of operand 2 fetching in data processing instructions, sources conflict on the exact details of ASR and ROR in cases when for example the shift amount is > 32, basing it on Red Squirrel.

14/1/01

Finished building switch table of ARM3 instructions. Started adding the semantics for them, inparticular the complexities of the data processing instructions where the flag calculations and affects on the PC change depending on exactly which registers and/or immediates are used to specify barrel shifting.

Received the ARM ARM from Mark Burton for which I'm very grateful.

13/1/01

Set up the latest source of Red Squirrel to compile and run. Started implementing the interpreting emulator in Red Squirrel (though self contained). Once mature enough will start attempting running basic code on it externally to Red Squirrel to test basic operation and after that will work at getting RISC OS to boot.

Initial considerations were the decoding of instruction. Decoding on bits 20-27 seemed obvious as a result of the denisty of different bit fields which determine the exact instruction and in many cases several of its arguments. I briefly looked at including the condition code in this lookup but since it's purpose does not vary from instruction to instruction, it can be considered before the instruction decoding takes place.

Started building switch() table of ARM3 instructions to be decoded using RedSquirrel as a reference and confirming with the ARM610 datasheet – this took some time.

12/1/01

Asked Mark Burton to send that copy of the ARM ARM.

Latest developments from RS.

Drew up outline of final report, particularly introduction.

20/12/00

Most of the last two weeks has been lost to various job interviews which has resulted in a ridiculous delay in progress with the project. The cause of this delay is notable since two of the interviews that took place are relevant to the project.

The first was at Broadcom, a U.S. based company which has recently purchased element 14, the reformed version of Acorn. In the process of my assessment I was interviewed by Sophie Wilson who played a lead part in the design of the ARM instruction set and John Redford who wrote the software PC emulator for Acorn machines. I was also able to discuss my project with two of their software engineers working on the suite of tools for their FirePath processor, and inparticular the progress they are making in simulating that processor for use by clients in the absence of a fixed hardware platform.

The second relevant interview was at ARM, talking to their software engineering department as well as a senior R&D engineer from their tools department with whom I could discuss my project and their work on modelling ARM processors (although demands for accuracy normally prohibits any use of dynamic recompilation).

4/12/00

Finishing touches to the progress report, proof reading, correcting, amalgamating graphics and adding references and footnotes.

3/12/00

Wrote the majority of the progress report, diagrams etc.

2/12/00
Made a start on the progress report and got a rough list of subjects together.

1/12/00
Read some of the book 'Introduction to assembly language programming' to ensure that I'm not ignoring any crucial problems in my thinking. Issues such as the fact that most x86 instructions, rather than having a destination register and various source registers, simply operate on a destination register destroying it's old contents, this will have to be performed in the dynarec by means of temporary variables to prevent destruction of the emulated ARM register's state. Flag adjustments may be frustrating and have issues such as the already identified carry/borrow issue with ARM SBC and x86 SBC setting the flag differently, also the issue of the overflow flag being in a separate byte of the EFLAGS x86 register and so more tricky to manipulate. Choosing instructions is something else to consider where there are different uses/costs associated with different but similar/equivalent x86 instructions, such as the classic ADD register,1 or INC register, I suspect gains from this kind of choice will be negligible and thus not making a decision in the recompiler will be preferable. Since many x86 instructions can take their operands (or even results) from/to memory, not being able to allocate all ARM registers to x86 ones may not be so disasterous since they can be accessed directly in the register file.

28/11/00
Limited time today so read through some papers from Cambridge which Sara leant to me on the subject of optimising compilation, particularly one by Auslander and Hopkins on their PL.8 compiler and the choices made for the intermediate representation, emphasising research on the target architecture and the breakdown to a 'simpler abstract computer' in a similar way to phetacode. I've still had no revelations as to how I can nicely represent the idiosyncracies of machine code in some abstract language and have no sources other than ARMphetamine which attempt to either. Also a paper on data flow analysis which seems to confirm the complexities of it and most interestingly a paper by Mycroft and Norman (of Norcroft compiler fame) about optimisation which nicely categorises different types, debates the relative methods and raises the issue of the order in which to perform such optimisations and deals with various other problems such as register allocation, intermediate code generation and a particular highligh on the conversion to linear code of multiple blocks that end in unconditional branches.
In my experience most ARM subroutines end with an LDM which restores several registers from memory, including the PC, which would normally prevent chaining of basic blocks (something I wish to do given the atomicity of each recompiled chunk in light of discussions with Sara), however there will be occurrences of unconditional branches and at the very worst, inlining such subroutine code into larger chunks is not impossible.

27/11/00
Met with Sara Kalvala and discussed the relative merits of different IRs, optimisations and methods of profiling especially identifying basic blocks. Her opinions seemed to agree with mine which was very reassuring. Details in log book. Also met with Graham and discussed the format of the forthcoming progress report and the timescale taking into account the the unanticipated problems with GUI development and my belief in the simplicity of profiling. Although currently behind schedule, given that no

work is pencilled in for December apart from the progress report ground should be able to be made up there without too much difficulty.

26/11/00

Having played more with the GUI I am finding it something of a struggle to get the GUI working, as although Visual C++ does make many things easier it is still a massive package and the documentation and usage take some getting used to. I feel I may have to allocate several days to working on this and getting to grips with it if use of the final product and debugging of the emulation is to be bearable.

In the meantime I have done more research into the decoding of ARM instructions, which is something of a mess. Several other emulators/disassemblers have been studied and I am progressing with the disassembler (although unable to test it until the GUI is working) to enhance my knowledge and get something practical done. The more complex it becomes, the more I am convinced that a significant speed up can be gotten by simply not having to do all the decoding, let alone any other optimisations.

Dug up Julian Brown's macros for his run time assembler used for generating x86 machine code directly without a slow classic assembler, being able to use them with his permission will save a lot of trouble. I'm also in search of a decent x86 disassembler since this will be very useful for analysing generated code and debugging and there's little benefit from me writing one myself.

While examining RISC OS Ltd's new release of documentation relating to a future version of RISC OS that will run on 32bit-PC ARMs such as Xscale I was interested to find they noted that the infamous ARM Architecture Reference Manual is available on the ARM developer suite CD which I have toyed with but hadn't noticed this before, this document should provide a comprehensive and authoritative reference if the other sources are ambiguous or unclear.

At the end of the day the disassembler is well underway with the only instructions left to handle being the co-processor instructions, single data swap (and the extended BX, long multiples, and halfword/signed byte data transfers which aren't present on the ARM610 anyway). I've also documented fairly comprehensively any ARM assembly features left out of the disassembler. This does need extensive testing, probably manually against sample disassembly from the StrongEd text editor for RISC OS machines which should eliminate the majority of the bugs.

23/11/00

With a desire to get this project off the ground and make some progress (and being crucially aware that I am falling behind schedule in hoping to have an interpreting emulator finished by tomorrow) I've started work on the relatively simple task of building myself an ARM disassembler. This could conceivably be taken from another project but I feel that it'll be a relatively graceful start to the implementation of the project and by designing it myself will allow me any flexibility I want. At current stage I have identified that I'll decode on bits 24-27 of the instruction, with secondary decoding phases for instructions which are not conclusively decidable on so few bits e.g. Data Processing and Multiply. I've built a simple GUI using VC++ and have started implementing the simple instructions to decode such as SWI and the condition codes. This can conceivably be finished by tomorrow night.

After some discussions from Neil Bradley on the dynarec list he advises against an intermediate representation in a dynarec, rather assembling blocks of information about each source instruction and emitting native code directly from this, leaving

optimisation to the code generated based on optimisation analysis from the source profiler. This is an interesting idea and could well be more efficient. It doesn't lend itself quite so well to the standard compiler techniques and documentation which is by and large based around an IR and DAG analysis, and is a more risky technique if I run into problems as I won't even have an IR and optimisations to show for my efforts. On the positive side, the blocks of data about the source might lend themselves better to a threaded interpreter than an IR in the style used by ARMphetamine.

22/11/00
Analysed the different ways that the condition code values match to the values of the PSR flags for conditional instruction execution. Was dismayed to find that as I was hoping was the case, that the condition code could be thought of as equivalent to a mask for the flags was not the case. Rather there are various combinations of flag settings which match some condition codes, this is not a real problem since the x86 instructions have similar condition codes and a function table from the given condition code can easily be used to check the more complex combinations of flag settings.

20/110/00
Identified the key issues surrounding the interpreting emulation, specifically identifying the fastest ways to decode ARM instructions, and the various ways of storing the ARM flags. Notes made on the subject in log book.

7/11/00
Read most of the chapter 8 and 9 in 'the dragon book', concerning IR representations, code generation and simple optimisations. Made some notes in the log book concerning feelings about the ideas expressed there, which I won't repeat here. The reading did clarify a lot of the work that I'll have to do. More reading on these subjects is needed but I feel some decisions could be made soon. One of the things I'm concerned about is whether a conditional branch need definitely be the end of a block of recompiled code (as it is for the 'basic block' concept) since in several cases small loops will occur which could give great speed up if the entire loop, including continue-looping-condition checking were recompiled, whether this is practical is another matter.

6/11/00
Came across two interesting and related projects, Sleeve and riscose. Sleeve being an ARM instruction interpreter, 26bit and user mode only (ina similar style to ARMphetamine's compatability) and more interestingly riscose which is a high level implementation of many of the RISC OS SWI calls and the Shared C Library, the idea being to be able to execute RISC OS binaries under ARM Linux with a combination of emulation and high level code. This high level implementation may prove interesting/useful if I ever get to the stage of trying to add high level emulation to my project in the latter stages.

3/11/00
Looked at extended architecture and instruction format of long multitples, branch with exchange and half word data transfers which appear to be the only new instruction types in the v3 and v4T architectures. Added details to my ARM decoding document. Thumb can certainly be dealt with as a switch to a new decoder when encountered by

the interpreting emulator. Jazelle, although highly unlikely to be coped with would be handled in a similar manner. This would clearly be a major issue for static recompilation (probably more so than the program/data identification problem) and may prove to be frustrating when building an ARM disassembler, I'll have to check if ARM's SDK has any special features for the user to intervene to distinguish ARM and Thumb instructions.

1/11/00
Trying to establish what parts of the vast array of different ARM features should be implemented or planned for from Furber. Need to make a decision soon and lay out which features are required, desirable, should be taken into account for possible extension and which can be safely ignored.

30/10/00
Experimented with LSL and LSR and studied the ARM610 data sheet on this subject.

29/10/00
Read through in depth the ARM610 data sheet concerning the decoding of ARM instructions. I've chosen the ARM610 at this stage since it's v3 of the ARM architecture which has the 32 bit PC of later ARMs but without the extra (and irrelevant to dynarec issues) complexity of 64bit Multiplies, Thumb instruction set, 5 stage pipeline etc. which characterise the more complex but very popular ARM7TDMI core, though these will be considered. Put together a sheet on the encoding of each type of instruction and the meaning of the internal bit flags. This is one of the key things to be done before the emulation can proceed. This needs to be finalised with a look at the later ARMs instruction sets to check for any problems which may arise if I don't plan for extensions and try and add them later, such as the BLX which jump's to Thumb code, use of the previously deprecated NV condition code and other such novelties. The other main thing to be analysed on the ARM is the idiosyncracies of the barrel shifter which although initially looks simple has hidden unexpected features affecting the condition flags amongst other things. Both these things need to be looked at in depth and documented (sufficiently to start an interpreted ARM emulator) tomorrow.

27/10/00
Installed evaluation copy of the ARM developer suite, particularly to look at the debugger functions of ARM's ARMulator program, an accurate emulator for debugging purposes. Also confirmed that the ARM610 datasheet (and presumably later data sheets) contain specific information about ARM machine code instruction formats which will be needed for decoding. I need to take a look at later ARMs and look at which version I will support. Getting directly on with the project, priorities will be to look at the idiosyncracies and side effects of the barrel shift (since this is known to be compex) and the methods that can be used for setting the x86 flags and similarities to ARM flags.

26/10/00
Read shortish section on IRs in Advanced Compiler Design & Implementation by Steven Muchnick. Discusses high, medium and low level IRs and the purposes of each, it was also some comfort to read; 'Intermediate-language design is largely an art, not a science.' i.e. much of the design is down to personal preference and there are

no hard and fast rules as long as the IR is well built for it's purpose. From the descriptions of different IRs, any kind of dynamic recompiler will have a low level IR since there is little or no knowledge about source program functionality. This low level approach is described as often having a one to one mapping of instructions to the target, or perhaps expanding IR instructions into fixed code sequences, much as anticipated. There is a description of Extended Backus-Naur Form (XBNF) at the front of the book, used in the IR section, which I recognised in this context as being similar to ARMphetamine's IR description and is likely the way I'll conceptually write my IR even if the storage format is a binary representation for fast access and look up (parsing strings is too slow an option). More reading is needed on IRs.

25/10/00

Had a flick through some of the basics of the Intel Pentium II Basic Architecture document about memory and the background to segmented memory and the easier protected memory version. Had some interesting details about the layout of the EFLAGS register and the instructions that can be used to modify the PSR flags. As described from ARMphetamine, the x86 has Sign flag (Negative), Zero flag, Carry flag, and Overflow flag amongst others, which the ARM uses. The rules by which these are set for individual instructions are likely to vary from the ARM, and I know for a fact that the ARM's bizarre way of Carrying from subtractions is different. The doc confirms that only the Carry flag can be manipulated directly by a strangely large number of instructions (STC, CLC, CMC, BT, BTS, BTR, BTC) whilst the other flags are much less adjustable. Adjustments to other flags may want to be emulated in software rather than using native x86 flags, depending on two factors:

a) similarities between ARM and x86 flag setting rules
b) ease of access to x86 flags both to read and write

This needs to be studied in detail but due to the limited number of instructions on the ARM, shouldn't take too long.

I took a quick look at parts of the source to 1964, a Nintendo 64 (source MIPS R4300i) dynamic recompiling emulator for x86 computers running Windows. Most of the code seems to be preassembled (and in fact literally hex values) of covers which are used for translation of MIPS source and is extremely messy and would be hard to debug.

Also read most of chapter 4 and 5 of Art of Assembly language which is getting increasingly less useful. I may try reading later chapters in the hope that it will provide some insights to x86 without all the rhetoric though the Intel manuals seem more readable than I was expecting.

24/10/00

Thinking some more about the IR of ARMphetamine I'm concerned about how I might get the extra features of the processor, necessary for an emulator of a real machine, into the IR. I suppose it would be possible but could leave the IR a little messy with many explicit instructions to modify the state of the emulated machine. I'll have to look at ARMphetamine's appendix on the IR and consider how NB's dynarecs do the IR for use with real systems before deciding further.

23/10/00

I took another look at some of the documents on the Sun website about techniques used to speed up Java technology, in particular previous JIT and now the Hotspot compiler. The first thing I established is that the older JIT JVMs (which have been

available for at least 2 years) are relatively simple: basically the JIT interprets the byte code until it comes to a method call when it then translates the method to native code and executes it for every method. There is no analysis of how many times that method or areas inside that method are run, so translating and running native code is often slower than simply interpreting byte code due to the overheads of translating which are never recouped by multiple calls to the native code. It also appears that very little optimisation is done on the native code generated (hence my description of translation rather than recompilation). ARM code doesn't have explicit method calls like Java byte code does, though this feature could be applied to every branch instruction though the idea of no code analysis seems foolish. The other option open to JIT JVMs was to recompile the entire program at run time which is not very useful for Java's dynamic loading classes functionality (and also drastically slows down the class loading), this is more a static translator rather than dynamic recompiler and not suitable for emulation.

Hotspot on the other hand is a far more advanced recompiling JVM. It starts off interpreting and analyses the code for information used in recompiling it also appears not to operate just on methods but on blocks of code that are performance 'hot spots'. The emphasis appears to be on only recompiling the critical much executed areas of the program but doing a lot of performance on them in order to be able to perform more advanced optimisations and not waste time recompiling code unnecessarily. Hotspot also uses 'dynamic deoptimisation' in order to deal with classes loading which may inherit from and overwrite previously recompiled functions (the closest Java comes to the self-modifying code problem) by dumping those recompiled versions. The class loader of course knows which recompiled blocks are being overwritten, monitoring which parts of recompiled program memory are overwritten in ARM code will not be so trivial.

Some of the 'classic' optimisations done in Hotspot are:

Method inlining – It may be possible to detect start and end of simple functions by variations of the starting STM and finishing LDM that save and restore respectively the PC. I suspect this will prove quite complex in practice with interaction between several functions though identifying these could be useful for other optimisations.

Dead code elimination – eliminating tests and routines that can never be executed.

Loop invariant hoisting – taking invariants from inside the loop out of it to prevent profitless recalculation. For the majority of immediate values there is little time penalty on an ARM as they are encoded into the instruction. It may be possible to take these outside the loop on the x86 in order to get similar benefits.

Common subexpression elimination – recognising expressions calculated previously and reusing the calculated value. This would probably be quite complex to recognise so may well to be avoided.

Constant propagation – is this the same as 'copy propagation' in the dragon book? If so it recognises use of duplicate variables so that the assignment of one could be eliminated, I suspect that this would be very nasty to recognise.

Having had a quick look at the Dragon book's (Compilers by Aho, Sethi, Ullman) section on optimisation it is apparent that some of these methods are not suitable for high-speed compilation on low-level code though others might prove otherwise. It is important to remember that much of this code will already have been through an optimising compiler and so many foolish programmer-induced-redundancies removed. The trick will be for my recompiler not to introduce redundancies into the recompiled code, it is possible that much optimisation will be practical on the final target code as opposed to the source or intermediate layers.

22/10/00

After reading the section in Furber about the Thumb instruction set it seems that reather than being a coprocessor, Thumb is actually just a 16 bit representation of a subset of the ARM instruction set, almost all Thumb instructions have ARM equivalents (with the occasional required exception on p193). ARM processors supporting Thumb have a decompressor that when in Thumb mode just looks up the relevant ARM instruction for the current Thumb representation before executing it. This would mean that in my dynarec Thumb is just a problem for the decoder and would not significantly alter the output x86 and perhaps not even the IR. Other reading suggests Jazelle implemented in a similar way and is not a coprocessor either, though due to it's recent addition and use in specific applications I do not expect to pay much more than lip service to Jazelle.

At great length read Julian Brown's dissertation for ARMphetamine making several pages of handwritten notes. His approach of splitting functionality of complex instructions (i.e. conditionally executed, making use of barrel shifting and the logic/arithmetic operation) into separate intermediate representation (IR) instructions which are then simply translated to equivalent x86 instructions is an interesting one and a good candidate for my project. He apparently suffered problems with the barrel shifting complications due to his IR not being designed to cope but did get it working. There are many aspects of interesting work done as well as detailing many limitations and things I'd like to explore such as which ARM flags to emulate in native x86 flags and which not to, also his suggestion of predicate conditional execution doing more data flow analysis to see what's required rather than just calculating all flags for a conditional instruction. Other things were confirmed to me such as the easy way out of dropping back to the interpreter when direct PC manipulation is performed.

The IR is problematic since if I wish to make my project at least a little retargetable (as yet I'm undecided on this) there is no clear IR which can efficiently be transcribed into various target processors. A very interesting paper which highlights several things I need to consider particularly if I am to take my project further to emulate a full system, ARMphetamine was never really designed to do this.

I've read an introduction to x86 sent to the dynarec.com mailing list by Neil Bradley (NB) which gave some insight into how things work. Despite the behemoth that is the Pentium instruction set I think the complexity can be avoided and I'm not quite so scared of the whole thing. Art of Assembly language is proving a little too verbose in it's explanation and is for people wholly new to assembly languages, I need to find a new resource to learn more detail from.

Reading some past discussions from the dynarec list (that Victor had archived away) I came across an old debate on register allocation routines for dynarecs. NB particularly favours statically allocating emulated registers to native ones in order that you can then jump to the middle of a recompiled block (at an instruction boundary) and know what's going on. As Julian Brown pointed out, this obviously isn't practical for emulating processors such as the ARM because of the very large number of emulated registers relative to the native regs. NB debated that you can just store a small subset of them in the target registers and do everything else to/from RAM. I'm unsure if that would be practical knowing the complexity of some ARM operations and the fact that there are so many true general purpose registers. This ability to re-enter cached blocks and static/dynamicly allocated registers is fairly fundamental and needs thinking about.

21/10/00

According to GB, "[Endianess] is controlled by the 'control register' - register 1 coprocessor #15. Bit 7 says whether the memory architecture is little or big endian. As such big/little endianness is part of the MMU rather than the processor. I got this from the arm7500 data manual page 4-15. On the back of this I've done some more reading in Furber about coprocessors on ARMs. It turns out that all the extensions to the instruction set such as FP, and I believe Thumb and Jazelle too are implemented as coprocessors on the ARM chip (although can also be on the main board) through a relatively standardised interface. Furthermore, if a coprocessor isn't present and an instruction for it is executed, the undefined instruction trap allows that instruction to be transparently although more slowly emulated in software (as was available on RISC OS as the FPEmulator). This is good news in that if the coprocessor interface is implemented in my project I won't have to worry about these extra instruction sets at all except as more instructions in the common set. There is a system control processor that seems to have some control over standard system features such as the cache in an ARM3, more research will be needed on coprocessors and how to allow extension.

Did a little resource harvesting, managed to get some docs on big endian, FPEmulator, ARMulator, start-up configuration, fixed point arithmetic, Thumb instruction set introduction and many more from

http://www.arm.com/sitearchitek/armwww.ns4/html/documentation?OpenDocument

and the odd doc from comp.sys.arm

Flicked through doc on ARM's recently announced Jazelle technology to provide a Java Byte code executing coprocessor. This is particularly interesting as it shows where ARM feel the boundaries between a hardware coprocessor and software emulation in ARM code (for more complex instructions, FP, new, divide etc.) is drawn when trying to execute CISC-like Java Byte code on a RISC. ARM claim a competitive performance with a JIT JVM (similar to dynarec) but without the memory and compiling time overheads.

To do some directly relevant work, I read to the end of Chapter 3 of 'The Art of Assembly Language' which gave me some insight into the relative mess of diversity of x86 asm but the fact that the basics are just that, basic. I'm a long way from being an assembly language programmer but it's a start.

20/10/00

After more discussions with GB it's clearly established that I'm going to have difficulty with interrupts, in Red Squirrel at present interrupts are checked for after every instruction. This level of accuracy may well be unnecessary though could prove otherwise. The problem is as GB pointed out that in the case of the code:

```
.loop
      movS r0,#1
      bNE loop
```

an infinite loop is created which could be broken only by an interrupt (such as a reset) though if this were compiled to x86 code (which I **still** haven't learnt much about) it would cause an infinite loop and not be stopped by an emulated interrupted. To cut it short, all loops will have to periodically check for interrupts, either by doing this in the recompiled x86 code (this is my preferred option) or by dropping back to the fetch-execute-decode emulation to check. I should really look at how ARMphetamine does this though since it's an incomplete emulation this may not be handled. This must be a problem for other dynarecs for other systems though.

GB also highlighted the problems for a dynarec of being able to alter the PC directly on the ARM with instructions such as:

```
MOV PC, r14             ; the code can do anything to r14
LDMIA r13!, {pc}        ; again code can and does modify the stack
before returning
LDR pc, [r1, r0 LSL #2] ; branch to r1[r0]
ADD pc, pc, r0;
```

these may result in having to drop back to the fetch-decode-execute emulator before checking for already recompiled code for the appropriate new value of the PC. I expect many of these problems will be clarified once I've had a good look at other dynarecs, specifically ARMphetamine and better understand the limits of what previous dynarecs have done with such things as jump tables etc.

19/10/00

After some discussion with Graeme Barnes (GB), author of Red Squirrel, I've established that 'If any application wants to be fully compatible with every acorn, it must be little-endian' and hence the potential endian switching problem isn't so much of a concern. I'm also pretty sure, though this is unconfirmed, that although the ARM can operate in both modes it's hard wired for each system it's used in.

18/10/00

Finished specification document with details of the stages in designing a dyanrec as I see it now. Added a lot of detail as to the problems with incredible variations in ARM processors and what I hope to implement. Also details of follow up ideas for features to a working dynarec which would be novel and useful, though may never get anywhere near implementation. Timetable has been set out quite conservatively with little spare time at all, though the month off for Christmas may provide much needed room for manoeuvre. Now this document is finished I feel I must get down to learning the basic skills I need for the project, and in particular, x86 asm.

Read my way up to date on old emails from the dynarec.com mailing list (over 400), came across an interesting discussion that it may be faster on x86 when emulating an add or subtract to test bits in the AH register directly than to try and force them into the native flags and use them from there. Also note the problem that the ARM and x86 use opposing carry systems for subtract instructions.

17/10/00

Wrote background section of specification in more detail outlining briefly what dynamic recompilation is. Read two more chapters of Furber's book detailing the organization and differences between 3 and 5 stage pipelining versions of the ARM core as well as the instruction set section where I'm hoping to find details of expansions to the ARM processor which need considering in the design even if not implemented.

I've ordered the ARM software development kit evaluation version which contains a version of ARM's own 'ARMulator' emulator for debugging code. Although designed for very different reasons from a dynamic recompiler (absolute perfection and versatility for debugging ARM software as opposed to raw performance) it may provide some useful insights.

I'm slightly concerned that I still don't know much about x86 assembly and haven't read all the docs available to me about dynarecs. I feel my priority must now be to

urgently read directly on the subject, particularly the ARMphetamine dissertation and x86 asm tutorial before getting the specification tied down for the rapidly approaching deadline for it.

16/10/00

Read through first three chapters of Steve Furber's ARM system-on-chip architecture, dealing with background to development of ARM processor (relative to other RISCs and CISCs), an overview of the ARM platform's major features, not branch-delay pipelining etc. and an introduction to ARM assembly programming which refreshed my mind on several of the key features of ARM which may cause me problems if not anticipated, such as:

- ?? the powerful and flexible barrel-roller on the second operand for most instructions.
- ?? the high visibility of the PC and PSR flags.
- ?? pipelining effects on PC leading to unpredictable results.

Also contains some interesting statistics (p21) about frequency of use of various types of instructions on an ARM processor in a print preview program and can be expected to be broadly similar.

12/10/00

Read through paper 'Binary Translation: Static, Dynamic, Retargetable?' by Cifuentes and Malhorta, largely concerned with academic projects and running legacy software on modern systems partcularly with retargetable runtime environments. Contains some simple diagrams on various binary translators as well as discussion of the merits of the various methods.

Having been subscribed to an email list for at dynarec.com (for the purpose of discussing dynamic recompilers) since early June, I spent some time reading over 600 previous emails to glean information. The group have been working towards building a dynamic recompiler for the 68k->x86 with one of the leading members having previously worked on a dynarec for the Z80->x86. Although there is quite a lot of noise on the group, it made me more aware of some of the various approaches people are taking with dynarecs, such as intermediate representations, how to store decoded instruction information etc. there is some way to go but this may prove a useful resource for brain storming and x86 assembly problems if questions are phrased in the correct way.

11/10/00

The 'Generator – A Sega Genesis Emulator' project by James Ponder is of interest as it is the most recent previous emulation project undertaken at Warwick in 1997, and also examines to some extent the idea of dynamic recompilation of a source 68000 processor. Several points of interest:

- ?? The document has some interesting points about building an interpreting emulators look up table of functions mechanically rather than by hand to avoid the inevitable human errors in writing 65536 functions. Unfortunately a complete look up table is obviously not feasible for the 32 bit instruction ARM.
- ?? The idea of using the decode table that builds opcode functions to also generate information for the disassembler is a good one.

?? On the subject of dynarecs some discussion is given to block identification, James is of the opinion that blocks must end when the PC is modified. I believe this may not be the case but need to investigate further. There is also discussion of elimination of redundant flag calculation removal which is probably irrelevant on an ARM dynarec due to the PSR flags being set conditionally for every instruction.

?? Interesting idea of having two versions of each opcode function in the interpretive emulator, one which calculates all flags and one which calculates no flags as this commonly occurs to save on memory rather than do every permutation of flag calculation. By detecting blocks and applying this either/or flag calculation to the block, a faster interpretive emulator has been produced.

?? Checks for two instruction loops which are waiting for interrupts, as this is common and wastes time, then immediately generates the interrupt being waited for.

I spent some time collecting information on fields of interest to which dynamic recompilation may be relevant. To this end I've downloaded a copy of the source code to Sun's HotSpot Java Virtual Machine (JVM) which is reknowned as a high speed 'just-in-time' (JIT) compiling JVM. I also downloaded the first and second edition JVM specification to clarify anything not found in the documentation. I've collected several links to other university research groups and projects concerned with JIT JVMs in a hope that they will help eliminate features of JVM JITs that aren't suitable for a dynarec for a real processor leaving me with the few, if any features which are useful.

I took a look at Transmeta's webpage concerning their Crusoe processor which appears to use a similar technique to dynarecs at a very low level to emulate an x86 processor on a 'Very Long Instruction Word'-type processor. Unfortunately Transmeta seem particularly tight lipped about their work due to the potential commercial value of it.

10/10/00

My first meeting with Graham Martin (GM) and have given him a basic first draft of the project specification for him to comment on as it has to be submitted by the 20/10/00. He suggested that planning the timescale will prove difficult which was something I had already found and will need some thought. We agreed that it seems prudent that ideas of also working on a highly optimised interpreted 6502 emulator in ARM assembly whilst taking on such an adventurous project as a dynarec will not be a good idea but can certainly be considered once the dynarec is well underway. We briefly discussed whether it is advisable for me to read Julian Brown's 'ARMphetamine' project documentation since it's a very similar title to mine and decided that in the interests of not 'reinventing the wheel', and so long as Julian is given adequate credit for any references to his work, this won't be a problem. It's also probably a good idea for me to talk to a compiler expert about fast compilation techniques sooner rather than later, Sara Kalvala seems the obvious person. For our next meeting I'm planning to have read up on the available dynarec documentation and previous similar projects and to have gained some experience of working with Intel x86 assembly language.

9/10/00

Set up Acorn RiscPC and installed GCC compiler and text editor for writing test programs so I can now write test source programs in C and ARM assembly.

Downloaded ArmSI, a free piece of software which acts as a hardware identifier and benchmark speed tester for Acorn machines (circa 1993). I've written a basic first draft specification which is more a brain storming document than anything else and lacking in several areas, but will give me something to work from as I learn more about dynamic recompilers (dynarecs).

Previous to this project

Previous to starting on the project I know a little about programming in ARM3 assembly in User mode using the ARM BASIC assembler on a RISC OS machine and also using the asm() function with C++ in the GCC cross-compiler targetting the Psion Series 5.

I have little knowledge of supervisor and other modes and the more specific architecture such as SWI handlers, exception handling and will need to explore the use of extended ARM architecture, FPUs, Thumb instruction set etc.

I have some experience of C and C++ though most of my Object-oriented programming has been done in Java so I'll need to brush up on the C++ way of doing things such as templates if needed. Although I have used Visual C++ to briefly build small utilities I know very little about writing GUI software in Windows though have written GUIs for EPOC, RISC OS and Java.

As far as emulation experience goes, as I approach this project it is at the end of a year-long development of an interpretive Nintendo Gameboy emulator for the Psion Series 5, written entirely in C++.

I know little if anything about x86 assembly language, other than that it's a bit of a mess (relative to ARM assembly).