# RISC Simulator by Peter Higginson

## *Introduction*

My original aim was to build something simple that showed “how a computer works” for use in schools. (I volunteer in some local schools under the UK STEM scheme.) I had seen a “moving blobs” demonstration but wanted to do a better example of the core CPU. My first try was in Scratch but there were limitations and I switched to JavaScript to overcome these. JavaScript also did not require any installation and would run on tablets (and maybe phones).

Initially I resisted designing my own assembly language because I found it hard to draw the boundary of what to put in and what to leave out. Hence I opted to use the LMC instruction set and produced http://www.peterhigginson.co.uk/LMC which has gained a lot of users (partly when one of the exam boards set some LMC coursework) and some schools have put it on their internal networks.

There are many limitations of the LMC. It uses base-10 storage of positive and negative numbers and has a very limited instruction set. The standard LMC descriptions are short on details so I chose to implement “MOD 1999” arithmetic which makes extended arithmetic difficult. I therefore looked at enhancing it to look more modern – with some of binary, logical operations, multiple registers, indirect addressing, functions calls, a stack and overflow tests (using NZCV flags). The problem was – who would use something so artificial?

The situation changed when a teacher on the CAS site asked for help with anything that met the extensions the AQA exam board were making by using parts of the ARM RISC instruction set. They even gave an example (which is “example” in the supplied programs of my RISC simulator http://www.peterhigginson.co.uk/RISC/). I therefore decided that there would be potential users for an enhancement to my LMC simulator that implemented a simplified ARM instruction set.

## *Basic Design and Word Length*

Computer word sizes vary and on some modern computers it can be hard to be sure what the word length actually is - is it the width of the data bus to memory (which can be 128 bits or more) or the number of bits in a register (and what if there are both 32 bit and 64 bit registers) or the smallest addressable unit (usually 8 bits). If instruction lengths did not vary, we could use that – but ARM has both 16 and 32 bit instructions (and the Raspberry Pi has 8 bit Java bytecodes as well).

For a practical one screen implementation, 16 bit words seemed the best compromise because it allows about 400 words in a block on the screen (I know it is not a power of 2). I wanted to keep the basic modes simple so I used base-10 addressing of memory and 40x10 seemed a good fit. It is clear from the AQA example that they are ignoring how ARM memory really works (with byte addressing where words go up in 4's) so I went for a 16 bit word addressed implementation. Only having word addressing also avoids an endian choice (ARM has both endian types).

I did not want to get into multiple types of load and store so I just stuck to 16 bit registers and 16 bit operations everywhere. In a few other areas I used the LMC implementation – those functions work the same way because I did not have a good reason to write new code. So the Assemble function works the same way, you can alter the PC and memory contents and input is only numbers. (I did not want the first program a novice had to write to be a text parser!)

## *Instruction Set Design*

I started by looking at the ARM 16 bit instruction set (Thumb) but diverged from its details quite quickly. In particular I wanted to use only 16 bit instructions. One simplification I can make is that because there are only 400 words of memory, I only need 9 bits for an address. So I can make all the conditional branches direct (rather than relative as in ARM) and have directly addressed load and

store register instructions. One teacher asked for a directly addressed ADD and I had space for that (as well as a direct SUB – ARM has neither).

Because the AQA example used some 3 register instructions, I have provided some of those (the ARM 16 bit instruction set only has ADD and SUB). I did not bother with some of the more esoteric instructions but I did want to provide a stack feature and to be able to use it the way a compiler would to store local variables and to allocate and remove space (so there is ADD/SUB SP #n and stack relative addressing).

To get indirect addressing on the LMC you had to use self-modifying code. While this may be good education, it is not recommended for normal programming. So I have provided both indirect and offset addressing. (Offset so that compiler implementations of structure and class addressing would work in the normal way.) I did not like the ARM representation so I used one that I first saw on a DEC PDP-11 and I think is more intuitive – i.e. 16(r4) instead of [r4,#16]. Note (r4) is the same as 0(r4) and [r4] is allowed which is the same as ARM. (Auto increment and decrement are beyond the level I am aiming at.) The practicalities of instruction set space mean that the offset range is 0-63.

To do function exit and entry the way a compiler would, I have provided multi-register PSH and POP. You can specify the registers in any order but they always go into memory as if the lowest numbered was pushed first. For speed and to have simple functions (if it were a real implementation) I kept the ARM use of an LR register. To meet the compiler entry/exit needs the PSH can include the LR (last) and the POP the PC (first – for speed) so a compiler does not need a separate RET instruction (although one is provided for simple functions that do not call other functions).

For simplicity I have stuck to 3 character instruction mnemonics and anything not an instruction is assumed to be a label (my LMC did that). The assembler does not do arithmetic on operands.

Normally you would use a service call (SVC 0) to end the program but I have stuck with the LMC HLT. My practical experience of assembly language programming (1970-1985 almost exclusively and some since) has shown the advantage of making small data items look like illegal instructions (or HLT) and small negative data items the same or something innocuous (e.g. don't put branch there) so I have done that. The alternative of using a trap pattern makes life harder for a novice.

There is an instruction list at www.peterhigginson.co.uk/RISC/instruction_set.pdf.

## *Condition Flags*

The flags register holds the result of the last ALU operation - negative (N), zero (Z), carry (C) and overflow (V). In many implementations (including ARM) it has additional bits allocated to other status and control functions and so is more like a real register. Without that you could regard it as part of the ALU – in any case it is special purpose.

I am not aware of any modern computer that does not use flags (usually NZCV) to record the result of one operation and then use a subsequent operation to examine the value of the flags in a conditional branch instruction. I wanted to include this feature and have used the same branch codes as the ARM (which are fairly standard). Which instructions set flags varies between computer types (and the ARM 32 bit instructions have a bit to set the flags or not). To keep programming simple, I have assumed that all data that goes through the ALU will set all the flags. (So V is set zero by logical instructions rather than being left with its previous value, for example.)

The main reason for minimising the number of instructions that set the flags is that modern CPUs do out of order execution (e.g. of instructions that use different registers) and the order of instructions that set or use the flags cannot be changed. For that reason it is rare to have flag setting on memory operations or on address calculations (so ADD/SUB SP,#n does not alter the flags). Note that MOV on 32 bit ARM goes through the barrel shifter and may set flags, in my implementation it does not.

(Wikipedia states that the Z flag takes longer to calculate but I am not sure I believe this – multiple input gates would be used and the result is not needed until the next instruction anyway.)

A concept that is important is that add/subtract is the same for both signed and unsigned but that the C flag indicates overflow if the numbers were unsigned and the V flag indicates overflow if the numbers were signed. Until I did this implementation I did not realise that on SUB and CMP the C flag is set to "not borrow" so that SBC works correctly as the exception case to SUB. I think it is important to be able to do extended arithmetic (e.g. 32 bits or more) and so ADC and SBC are provided.

## *Multiply and Divide*

The simple multiply (MUL) producing a 16 bit result is similar to ADD in its flag settings except that overflow can mark the loss of many bits. This is different to ARM where the equivalent 32 bit multiply instruction does not alter the C or V flags.

The ARM processors targeted at low cost embedded solutions do not have divide instructions – I can think of several reasons for this, one is that most divides are by a power of two and that can be done by shifts, another is that a simple (non-dedicated) implementation would not be much faster than software in cache and a third that any application requiring lots of arithmetic would incorporate a floating point unit (FPU) and that would include floating point division. (Interestingly the Thumb instruction manual gives a linear divide by 10 routine – the most common other case.)

I decided to provide an extended multiply instruction (MLX) that would produce a full 32 bit result (in two registers) from two 16 bit operands. This is more important on a 16 bit machine than on ARM where the basic operation is already 32 bits and an FPU might be present. Mine is an unsigned operation and clears all the flags apart from Z which it sets correctly.

For division I have provided simple division and modulus (MOD). Division is not the same for signed and unsigned so DIV is signed division and UDV is unsigned.

## *Features not Implemented*

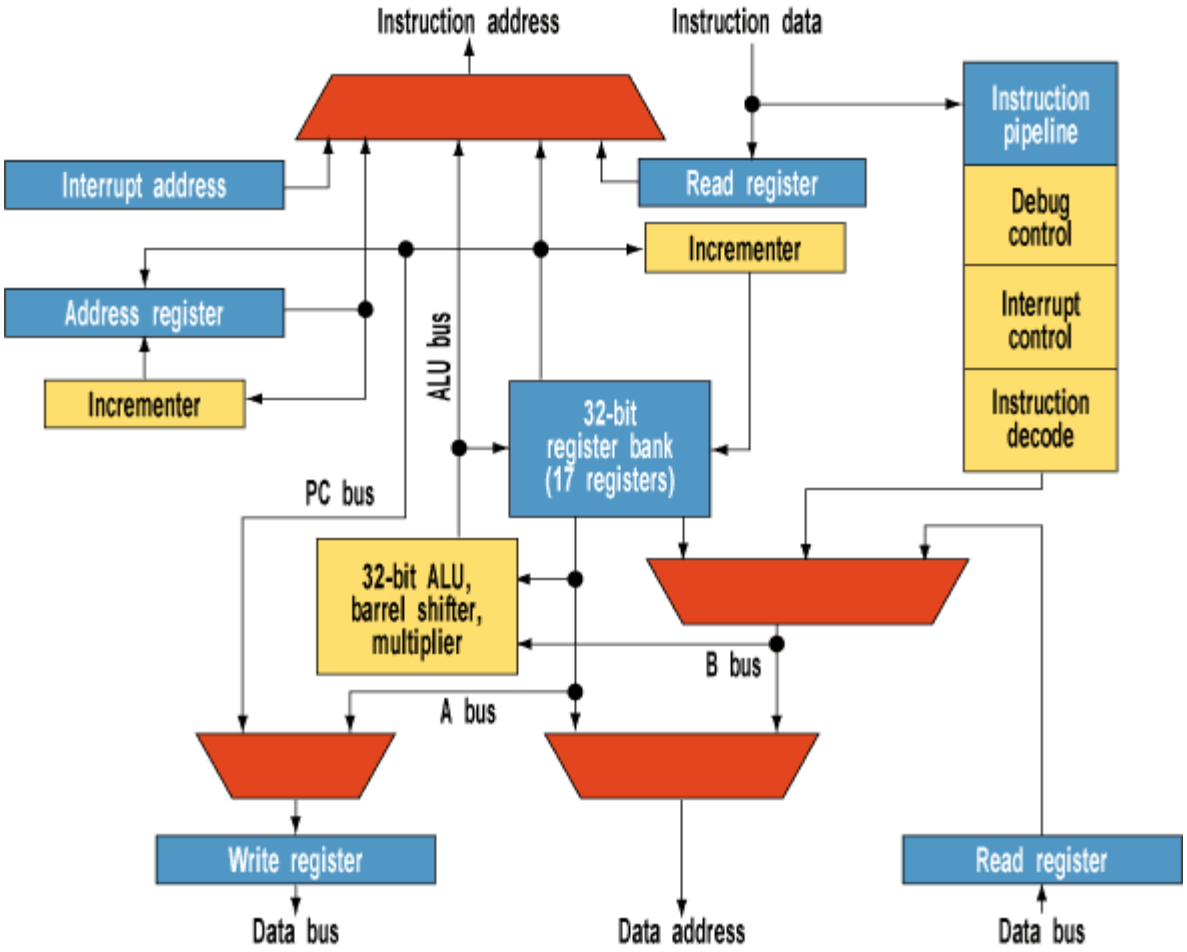
There are lots of things my simulation does not do - a real implementation would have a wider memory bus and be loading several instructions at once. We have no caches – instruction or data and we do not have any form of pipe lining – not even pre-fetch of the next instruction which most systems have. Most ARM implementations (other than the Cortex-M0/M0+) have separate buses for instructions and data (although the ARM address model is not true Harvard Architecture).

We do not have any features that would be needed by an Operating System such as interrupts, memory mapping or protection and system modes with duplicate register sets. 400 words is not enough memory for an operating system so we just keep things simple.

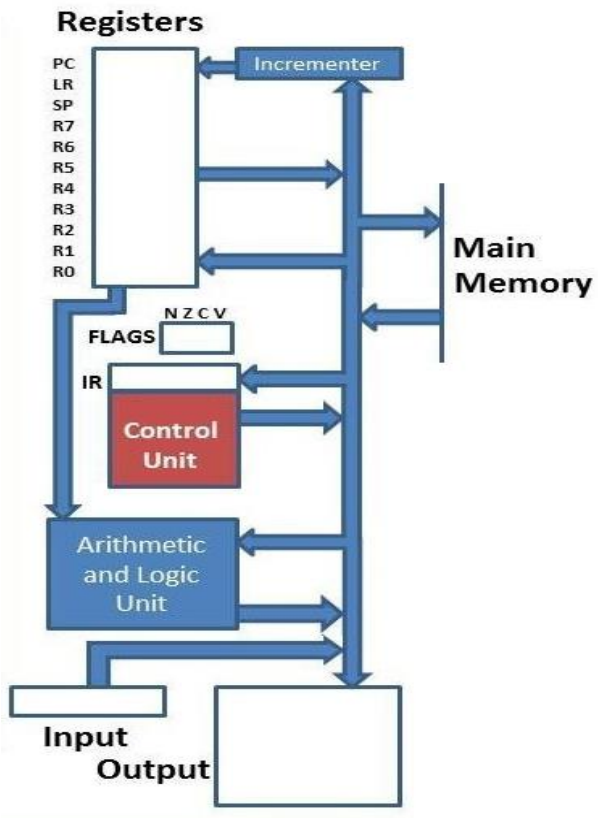
Real ARM systems have extensive speed enhancements such as branch prediction, out of order execution and delayed writes. You can get a flavour of a recent ARM design at http://www.design-reuse.com/articles/13397/arm-cortex-r4-a-mid-range-processor-for-deeply-embedded-applications.html (which I assume is based on an ARM press release).

## *Internal CPU Structure*

As far as I know ARM does not release internal structure diagrams. You can find lots of pictures on the net showing how to integrate an ARM CPU with memory, optional components (like memory protection and co-processors) but very little on internal structure. Google did find the following Cortex-M3 design on a university web site which I used as a base because it is at least plausible.




The Cortex M3's Thumbnail architecture looks like a conventional Arm processor. The differences are found in the Harvard architecture and the instruction decode that handles only Thumb and Thumb 2 instructions.




I have seen internal structure diagrams from other processors in the past and used this knowledge as well. You can see that the M3 has multiple buses – three named ones plus some others. Most descriptions I have seen divorce the barrel shifter from the ALU (Arithmetic and Logic Unit) which this diagram does not do. The M3 has separate instruction and data ports; I am trying to do an "M0 like" with a combined instruction and data port.

My simplified internal structure is shown on the left. I have made the B-bus (on the right hand side) the major interconnect and reduced complexity by using it for ALU output as well as one of the inputs. With no instruction overlapping this is possible but clearly a real implementation would increase parallelism and overlapping by separating it into two buses. Because there is no overlapping, the A-bus just has to move data from the registers to the ALU. To save writing another couple of pages of code, when only one bus is needed I use the B-bus whereas a real implementation would do the reverse and free the B-bus for something else in parallel.

In my RISC implementation there is a hidden bus that connects the registers (and the flags register) and none of the control logic is shown. You have to imagine the Control Unit having wires

connecting to and controlling all the other components. You have to imagine links from the ALU to the flag bits and from the flag bits to the Control Unit so the conditional branches can be executed.

Logic is relatively cheap so additional functional units are added to modern CPUs as needed. In the M3 diagram there are two Incrementers shown. These probably can increment or decrement by 1, 2, 4 or 8 (for different load/store instruction types). I have been told that on ARM the PC has its own private incrementer with a port that can write the PC independently of other register operations.

I have added a single incrementer that can write to either the PC or the SP. This is used by both Fetch and POP where the address sent to memory is then incremented and saved. For multi register POP multiple increments are required but using the incrementer multiple times is reasonably efficient. For PSH we need to subtract 1 (or n for multi register) and then send that address to memory – so we use the ALU for the subtract and then copy the B-bus back to the SP.

## *Assembly Options*

The assembler is very simple; I basically extended the LMC one and could not see any reason to provide all the features of the real ARM assembler. So all programs start at 0, anything not a known instruction mnemonic is treated as a label and anything after a valid operand ends might be ignored. A single / is enough to start a comment but for consistency with normal use, I always put // in examples.

Following assembly the code is shown correctly formatted with only lines containing instructions numbered (so it corresponds to the numbering in memory).

The assembler is two pass, pass one basically identifies the label (if any), the instruction code (or DAT) and the operands while ignoring any completely comment lines. The second pass can then match any label found in an operand and know its value because one instruction always takes one word. To be compatible with AQA you can define a label as “name:” and the : will be ignored.

As well as inputting a program into the Assembly Language area you can input numbers into the Program Counter and directly into memory. When you “Submit” the Assembly Language a memory clear and a reset are done and the program is assembled into memory. You can input hex as 0xnnnn everywhere a number is expected.

## *Signed, Unsigned or Hexadecimal*

This was one of the most difficult decisions. I showed a very early version where the Assembler put hex into memory to a few teachers and they thought it confusing. (If you re-wrote a memory location it changed to unsigned for example) One suggestion was that mouse hover might show the alternatives but that proved hard to implement and also there is no real hover on touch screens. I tried unsigned as a default but that made normal arithmetic look odd.

So I settled on signed (2's complement) as default and an option to show memory and Registers 0-7 in unsigned or hexadecimal. Since the PC, LR and SP only deal with addresses and the addressing is 0-399 unsigned, they only display as unsigned (except the SP can also be 400).

A blob takes and transports the mode it reads, unless it is an address in which case it is unsigned. In some cases (like MOV PC,Rn) the blob will move as data and change when it is written to the destination. For address calculations (like LDR R2,10(R1)) the address components might be hex on input to the ALU (depending on the current mode) and unsigned on output from ALU to memory.

I did also look at whether I could just use positive instructions but there was not enough instruction space to have half the number. I did consider trying to make the instructions a novice would use positive but realised that the base-10 representations of the instructions did not mean anything

anyway. (I did not want to have base-10 formatted instructions because a) no real computers do that and b) a common exam question is “how many instructions can you have with x bits” and I did not want to confuse students.)

The instruction register always shows as hex – there is no sensible alternative. I have now added a binary mode for memory but registers 0-7 and moving blob contents stay as they would in hex mode.

## *Input and Output*

As explained above input is only a number (but 0xnnn is accepted) and currently the device address is not checked (since there is only one input device). Output to device 4 is treated as signed but you can output unsigned (device 5), hex (device 6) or character (device 7).

ARM uses address mapping of I/O devices rather than separate input and output instructions but I felt that introducing the high address range normally used would be too complex at this level and LMC like instructions would be better.

I am seriously thinking about implementing simple interrupts.

## *Address Wrapping*

In common with many systems I have used (particularly embedded systems) the memory wraps if you address beyond the implemented range. (Technically it aliases; normally the top few bits decode to the type of memory and the bottom bits as required to the address within the memory bank. Any remaining middle bits are ignored.)

Any attempt to wrap the SP on a PSH or POP generates a fault and the system has to be reset in order to continue.

## *AQA Instruction Set*

The AQA Exam Board has produced an ARM like instruction set for use with its AS and A-level papers http://filestore.aqa.org.uk/resources/computing/AQA-75162-75172-ALI.PDF. To try to be more compatible, I have added an MVN instruction and permitted colon (:) to be used to end a label definition.

It is impossible for me to do instructions like ADD Rd, Rn, <operand2> in a 16 bit machine because operand2 would require 9 bits (it is a register or an 8 bit immediate) and the two registers a further 6 bits. I can do the ARM Thumb variants which are ADD Rd, #nnn and ADD Rd, Rn, Rm. You can live with 8 general registers instead of 12.

They don’t specify what <memory ref> can be (I suspect because the ARM meaning is complex) but the obvious cases are <label> and [Ra] which I support. (My <label> is a direct or absolute address whereas ARM only has relative addressing.) The same applies to branch instruction addresses.

In addition to providing the AQA link above and some Student notes, I am grateful to Graeme Mitchell for suggesting MVN, the new stack indication and binary mode. The stack indication is in green and assumes you only grow the stack from the default “top of memory”. Binary mode applies only to memory and expands the memory module to the right to accommodate it. Input to memory in binary mode is only in binary (elsewhere and in non-binary modes, you can input only numbers or hex preceded by 0x).

## *Some Implementation Details*

As part of teaching simple JavaScript to school children, I had produced a library to create and manipulate objects in a browser window. This has functions like rectangle, text, image and move for named objects and uses x/y co-ordinates. The library has 11 functions and is 300 lines of JavaScript.

The RISC emulator uses the library (but occasionally extends the basic functions e.g. by embedding more complex HTML in the text function). It has unfortunately grown to just over 4000 lines of JavaScript. The online version is run through a compress utility.

## *Acknowledgements*

The idea for the moving blobs to illustrate the computer's operation came from Romayne Sorhaindo's "CPU Fetch - Decode - Execute Cycle" project on the Computers and Schools web site (computingatschool.org.uk).

The original LMC layout came from Mike Coley (gcsecomputing.org.uk) and you can see echoes of it in the RISC version.  His help in getting started with the LMC version is gratefully acknowledged.

Thanks also to the many teachers who have used and commented on this and the LMC versions.

Peter Higginson
plh256 at hotmail.com
12 May 2016