ARMlite
Programming Reference Manual

By Peter Higginson

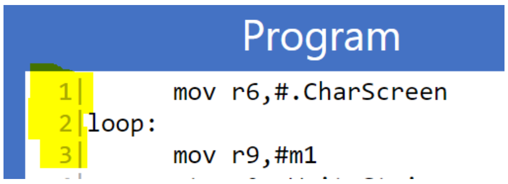
v1.2

Manual last updated: 29/10/2020

https://peterhigginson.co.uk/ARMlite/

Code layout & formatting

When code is successfully submitted, ARMlite will add line numbers as highlighted here:



```
1|       mov r6,#.CharScreen
2|loop:
3|       mov r9,#m1
```

The line numbers are purely to help the programmer navigate the code and to interpret error messages that specify a line number.

If the code is subsequently edited and re-submitted, ARMlite will automatically discard all existing line numbers, and create them anew. When inserting new lines of code, it is not necessary to give them line numbers, nor to adjust line numbers if code is deleted or moved. However, if you *do* edit a line number, you should leave the vertical bar character (also known as 'pipe') in place.

You can add one breakpoint by clicking on an instruction in the program area. The line number of the breakpoint is highlighted in red. ARMlite will stop after the instruction before the breakpoint (so in the case of a branch you can see where it branched from).

Comments

Comments are prefixed by a double slash: `//`. Comments may be added after an instruction or on a line of their own. ARMlite also allows `;` (for ARM compatibility) or a single slash `/` to indicate a comment.

Labels

Labels are used to specify the location of instructions for branching, and to specify locations of data for loading and storing.

Labels may be made up of upper- and lower-case alphabetic and numeric characters. They *must* start with a letter (in common with ARM, underscore counts as a letter). The label definition is followed by a colon e.g. `Loop1:` , and may be added in front of an instruction, or on a line of its own (in which case, the label refers to the location of the first instruction or data value that follows the label). You may have only one label per memory location. If the following instruction (or `.word`) requires alignment, the label before it is aligned (for example after `.byte`, `.ascii`, or `.asciz`).

Internal labels defined by the assembler all start with a dot (`.`). Register names are not permitted as labels. It is strongly recommended that you avoid using instruction names as labels.

Case sensitivity

Labels are case sensitive. Instructions, directives, and register names are not case sensitive: they may be written in upper-case, lower-case or a mixture.

Spaces

Any blank lines will be automatically be removed when assembly code is submitted. A space is required between the instruction and the first operand (if any). If the instruction takes multiple operands, these are separated by commas - additional spaces are not required, but will be ignored by ARMlite. Similarly, the colon defining a label must not have a space before it. If the label is on the same line as an instruction, it is common practice to have one or more spaces after the colon, before the instruction – though this is not strictly required.

Memory

ARMlite has a total of 1MB of memory i.e. 1048567 (8-bit) bytes in the range `0x00000` to `0xfffff`.

A single word of memory is made up of four successive bytes i.e. 32 bits.

All addresses specified in ARMlite code are *byte* addresses. Where a word of memory is to be loaded or saved (`LDR` or `STR`) the address must correspond to a word boundary i.e. it must be divisible by 4 - otherwise you will get an 'Unaligned access…' error.

Registers

ARMlite has sixteen 32-bit registers. There are thirteen general purpose registers – `R0` through `R12` – and three registers that also have special purposes `SP` (Stack Pointer), `LR` (Link Register) and `PC` (Program Counter). The `PC` is initialised to `0` and holds the address of the next instruction to be executed. It is incremented by `4` as each new instruction is fetched from memory to move on to the next instruction. The special registers *may* be used as general registers by the programmer (and may even be referenced as `R13` through `R15`) but doing so carries risks that you will corrupt the normal operation of the processor, so it is best avoided.

Instructions & Directives

ARMlite implements these instructions (listed alphabetically here):

```
ADD, ADDS, AND, B, BCS, BEQ, BGT, BL, BLT, BMI, BNE, BVS, CMP, EOR, HALT, LDR,
LDRB, LSL, LSR, MOV, MVN, ORR, POP, PUSH, RET, RFE, STR, STRB, SUB, SUBS, SVC
```

The following keywords are not 'instructions' (they are not translated into processor operations) but are 'directives' to the assembler (in alphabetical order):

```
.ALIGN, .ASCII, .ASCIZ, .BLOCK, .BYTE, .DATA, .WORD
```

For ease of programming and to align with mnemonics which are widely used elsewhere, the following synonyms are accepted:

```
BIS (=ORR), XOR (=EOR), OR (=ORR), JMS (=BL), HLT (=HALT)
```

Moving values between registers and memory

MOV

`MOV` ('Move') is used to place a specific value (an 'immediate value') into a specified register, or to copy a value from one register to another. `MOV` *cannot reference memory addresses.* The format is:

```
MOV Rd, <operand2>
```

Where `Rd` specifies the destination register, and `<operand2>` may be a source register or an immediate value (prefixed by `#`). Examples:

`MOV R0,#25`	moves the 'immediate' decimal value `25` into `R0`
`MOV R5,R3`	copies the value from `R3` into `R5`
`MOV R4,SP`	copies the value from the `SP` register into `R4`

Immediate values may be specified in decimal, hex, or binary format (e.g. `#0x3f`, `#0b1101`) .

With `MOV` on ARMlite the immediate value can be -2^{25} to $+2^{25}$ (+/-33554432). This is large enough that any user or I/O label (or HTML colour) can be given as the immediate to a MOV. (Note that `<operand2>` with other instructions has a much more limited range – see later.)

LDR, STR, LDRB, STRB

`LDR` ('Load Register') copies a value from a memory location into a register. `STR` ('Store Register) does the reverse: it copies a value from a register into a memory location. `LDR` *and* `STR` *cannot be used with immediate values*.

For `LDR` and `STR` the value copied is a 32-bit word. For this reason, the memory address specifies the first of the four successive bytes where the word is stored - *the address must therefore be divisible by four*.

Both instructions take two operands, the first specifies the register and the second specifies the memory address. In the simplest form, known as 'direct addressing', the memory address reference takes the form of a label or a number (specified in decimal, hex or binary). Examples of direct addressing:

`LDR R0,0xfc` loads the contents of the word starting at memory address `0x000fc` into `R0`.

`STR R1,120` stores the contents of `R1` into the word starting at memory address (decimal) `120`.

`LDR R3,0x81` is an invalid instruction because address `0x00081` is not a valid word address (it is not divisible by four). The Assembler will trap this. Using indirect addressing, such as `[Rx]` it is possible to generate an invalid address dynamically, in which case this will result in a run-time error rather than an error on submission.

`STR R3,myLabel` stores the contents of `R3` into the word defined by the label `myLabel`.

`LDR R4,#0xfc` is an invalid instruction because `LDR` does not work with an immediate value.

`LDRB` and `STRB` are like `LDR` and `STR`, but they work at byte level. `LDRB` will load just a single byte of memory (into the least significant 8 bits of a register – setting the other bits to `0`), and `STRB` will store the least significant 8-bits of a register into a specified byte of memory (not altering any neighbouring bytes). `LDRB` and `STRB` do *not* therefore require that the specified memory address be on a word boundary (divisible by four). For example:

`LDRB R3,0x81` *is* a valid instruction, and will load the contents of the single byte at address `0x00081` into the least significant 8 bits of R3.

'Little endianness'

The contents of memory are presented on screen as whole words, not individual bytes. However, ARMlite uses 'little-endian' format: when a 32-bit word is stored in memory, in four successive bytes, it is the *least significant* 8-bits that are stored in the first byte, and the *most* significant 8 bits that go in the fourth byte. This pattern is adopted by *most* modern processors because it has the property that the same value can be read from memory at different lengths without using different addresses and simplifies hardware in processors that add multi-unit values a unit at a time. (E.g. 64 bit arithmetic using a 32 bit adder or 16 bit arithmetic using an 8 bit adder.)

You might never become aware of this little-endian pattern, unless, for example, you deliberately stored a *word* using `STR`, and then chose to load it back as a sequence of bytes, using `LDRB`.

Indirect addressing

The four load/save instructions support 'indirect' addressing. Indirect addressing is most commonly used to implement 'pointers'.

For indirect addressing the memory address is not specified directly in code, but by a value held in a register. The register holding the memory address is specified as the second operand, surrounded by square brackets. For example:

`LDR R0,[R1]` will load into `R0`, the *contents of the memory address that is currently held in* `R1`. It does not mean 'load into `R0` the contents of `R1`'.

`STRB R3,[R4]` will store the contents of `R3` into the *memory address that is currently held in* `R4`.

Note that for `LDR` and `STR` the *contents* of the register must specify a *word* address (i.e. be divisible by four) – or you will get an 'Unaligned access…' error at runtime. This constraint does not apply to `LDRB` and `STRB` since those are accessing data at the byte level.

Indexed addressing

Indexed addressing is a variant of Indirect addressing, whereby the *memory address* to be used is calculated from two registers, commonly conceptualised as a 'base address' plus an 'index'. This addressing mode is commonly used to iterate over a sequence of data values held in successive words (or, for LDRB, STRB, successive bytes – for example representing a 'string' of characters), or to implement a mechanism roughly equivalent to an array in high-level language programming. For example:

`LDR R3,[R7+R10]` will add the contents of `R7` (which specifies the 'base address') to the contents of `R10` (the index) and use the result as the memory address, from which to retrieve the value to be loaded into `R3`.

`LDR R5,[R9-R1]` will subtract the contents of `R1` from the contents of `R9` and use the result as the memory address, from which to retrieve the value to be loaded into `R5`.

ARMlite will also recognise the following syntax, which is the ARM standard:

```
LDR R3,[R7,R10]
LDR R5,[R9,-R1]
```

Direct addressing

Neither ARM nor ARMlite has a native direct address mode. What looks like a direct address is converted by the assembler to the form `[PC+/-n]` where `n` is the delta from the instruction after next to the required address or label. So, for example `LDR R0,label` is converted by the assembler into `LDR R0,[PC+n]` (or `[PC-n]`). (The Assembler works out the correct `+/-n`.) For programs less than about 1000 instructions, this is always possible.

For larger programs, you may need to use a spare register:

```
MOV R9,#label
LDR R0,[R9]
```

For `[PC+/-n]` the value of n can be between 0 and 4095.

Comparison, Addition & Subtraction

Values in registers may be compared, added, or subtracted using the instructions: `CMP`, `ADD`, `SUB`, `ADDS`, `SUBS`.

CMP

`CMP` ('Compare') compares the values in two registers numerically, setting Status bits based on the outcome. A compare instruction is *typically* followed by a Conditional branching instruction. CMP takes two operands. The first is always a register. The second is either another register (it would make little sense to compare the contents of a register to itself!) or an immediate value. For example:

`CMP R0,R1`	will compare the values in `R0` and `R1`, setting status bits accordingly.
`CMP R0,#89`	will compare the value in `R0` to the immediate value (decimal) `89`.
`CMP #3,R2`	is an <u>invalid</u> instruction as the first operand must be a register.

(It may be helpful to realise that, under the covers, `CMP` is subtracting the second operand from the first, and setting the Status bits the same way as if a `SUBS` instruction had been used - except that `CMP` discards the resulting number when finished instead of storing it in a register)

ADD and SUB

`ADD` will add the values from two registers (or from one register, plus an immediate value) and place the result in a destination register (which may be one of the source registers). `SUB` ('Subtract') works in a similar fashion, but subtracting the second value from the first. The first operand is always the destination register, and if an immediate value is specified, it must be the last operand. For example:

`ADD R1,R0,#3`	Adds the immediate value `3` to the value in `R0` and puts the result in `R1`
`ADD R2,R7,R5`	Adds the values in `R7` and `R5` and puts the result in `R2`
`ADD R4,R4,#1`	Adds one to the value held in `R4` (we say 'Increments R4 by 1')
`ADD R5,R9,R9`	Adds the value in `R9` to itself (doubles it) and puts the result in `R5`
`SUB R1,R0,#3`	Subtracts the immediate value `3` from the value in `R0` and puts result in `R1`
`SUB R2,R7,R5`	Subtracts the value in `R5` from that in `R7` and puts the result in `R2`
`SUB R2,R5,R7`	Subtracts the value in `R7` from that in `R5` and puts the result in `R2`
`SUB R4,R4,#1`	Subtracts one from the value held in `R4` (we say 'Decrements R4 by 1')
`SUB R5,R9,R9`	Subtracts the value in `R9` from itself (producing zero) and puts result in `R5`
`ADD R1,#3,R0`	Is invalid because the second operand may not be an immediate value.

Note that `ADD` and `SUB` do not detect any form of overflow – this is just lost and the lower 32 bits of the answer kept.

ADDS, SUBS

`ADDS` and `SUBS` work the same way as `ADD` and `SUB`, respectively, except that they also set (that's what the `S` stands for) the Status bits. Thus, conditional branches may be used after an `ADDS` or `SUBS` without needing to have a `CMP` as well.

(On a real ARM most data processing instructions, including `MOV`, have an `S` form that sets the status bits. ARMlite restricts this to `CMP`, `ADDS` and `SUBS`.)

Status bits

The Status Register holds the result of the last `CMP`, `ADDS` or `SUBS` operation - Negative (N), Zero (Z), Carry (C) and oVerflow (V).

The use of the status bits depends on how the bit patterns compared (or added or subtracted) are interpreted.

To take the simple cases first, Z is set if all 32 bits of the result are **Z**ero. If the 32 bits represent a number, then Z would be set if that number is 0. On the other hand, if the 32 bits represented an instruction or four characters (or something not a normal integer) the Z bit would probably tell you nothing useful.

Following a `CMP`**,** the Z bit is set if the compared values were equal (which might be useful for comparing 4 characters at a time, for example).

Similarly, N is set if the top bit of the result is set. You can use this for purely logical purposes (where the 32 bits are not a number), but the normal use is that in two's complement form, a 1 in the top bit indicates that the number is negative (hence **N**). (For an unsigned number it just tells you it is greater than 2147483647 which is not needed very often.)

V is only useful (on its own) if the numbers being added or subtracted are in two's complement representation. V is `0` if the answer is right and V is `1` if the answer is wrong. For example, if you add two large (2's complement) numbers the adder will wrap round and the result will be negative. Clearly this is wrong, and V is set to show you this, provided you used `ADDS`. (Note V is also used in combination with Z and N to formulate the `BLT` and `BGT` instructions – see below.)

C is only useful if the numbers being added or subtracted are in *unsigned* representation. (This is also needed for extended arithmetic – for example when adding the lower 32 bits of a 64-bit number – irrespective of whether the whole number is unsigned or two's complement.) C=1 tells you for `ADD` that the answer is wrong on an unsigned interpretation. C is therefore equivalent to the 33rd bit in the result.

Because subtract was originally done by negating and adding, the C bit for subtract and compare follows what would have then happened. So for `SUBS` and `CMP`, the C bit works the other way up and C=0 means the answer is wrong on an unsigned interpretation (which is guaranteed if you are limited to numbers >= 0 and you subtract a larger number from a smaller).

The Z flag is tested by the `BEQ` and `BNE` instructions for "Branch if Equal" and "Branch if Not-Equal" with respect to the parameter values of the last `CMP` instruction or compared to zero for result of the last `ADDS` or `SUBS` instruction. ARMlite provides the conditional branches `BCS`, `BVS`, `BMI` which allow you to test unsigned overflow (`BCS`), signed overflow (`BVS`) and negative (i.e. top bit of the result). These are particularly needed with `ADDS` and `SUBS`. (Note real ARM has more conditional branch instructions than this, and AQA has fewer than this.)

There is a special case after compare (`CMP`) where the particular instructions `BGT` and `BLT` are available to test "Greater Than" and "Less Than" respectively for 2's complement numbers only. You can use these after `SUBS` but they don't have any use after `ADDS`.

It is beyond the scope of this manual and book, but `BLT` tests (N set and V clear) OR (N clear and V set) and `BGT` tests Z clear AND (N and V both set or both clear). Trust us – they do the right thing.

Bitwise manipulation

The following operations will manipulate values at bit level: `AND`, `EOR`, `LSL`, `LSR`, `MVN`, `ORR`. Each of these, except `MVN`, takes three operands (like `ADD` and `SUB`): a destination register, a source register, and a second source register *or* immediate value. `MVN` takes two operands: a destination register, and a single source register *or* immediate value.

AND, ORR, (OR), EOR, (XOR)

Perform a logical operation on each pair of bits in the two operands: the logical AND, logical OR, and logical exclusive-OR, respectively. Note that OR is recognised as a synonym for the ORR instruction, and that XOR is recognised as a synonym for the EOR instruction.

LSL

'Logical Shift Left'. Moves each bit one place to the left, placing a 0 in the right-most bit and losing the left-most bit off the end. If the value represents a number, the effect of shifting one place left is to multiply the number by two (but any overflow is lost without detection).

LSR
'Logical Shift Right'. Moves each bit one place to the right, placing a 0 in the left-most bit and losing the right-most bit off the end. If the value represents an unsigned number, the effect of shifting one place right is to divide the number by two, rounded down if appropriate.

MVN
`MVN` ('MoVe with NOT') is like `MOV`, adopting the same pattern for operands. The difference is that `MVN` will apply a `NOT` operation to each of the bits from the source before storing the result in the destination register. (This operation is sometimes also called 'inverting' or 'flipping' the bits.) Note that you can `MOV Rd,#-n` with a much larger range of `n` than `MVN` has and it is two's complement.

Immediate range with instructions other than MOV
Several instructions can take an immediate value as the last operand (sometimes listed as `<operand 2>` which may be a register or an immediate value). With `LSL` and `LSR` the immediate value is limited to `#31` because shifting any more than 31 bits would always produce a zero result. With the other instructions, things are complex. Values in the range `#0` to `#256` are always permitted. Values outside this range are allowed if they can be expressed as 8 consecutive bits on an even boundary (allowing wrap around). It is easy to see this in hex so that `0xff0`, `0x3fc00`, `0xd000000d` will be fine but `0x1fe` is not on an even boundary and `0x101` is nine bits wide. The assembler will give you an error if an immediate does not fit.

Examples of data processing instructions
For each of the following examples, `R0` starts with `0b1110` and `R1` starts with `0b1011`, and all values not shown to 32 bits have all leading zeros.

`AND R2, R0, R1`	results in `R2` containing `0b1010`
`ORR R2, R0, R1`	results in `R2` containing `0b1111`
`EOR R2, R0, R1`	results in `R2` containing `0b101`
`LSL R2, R0, #2`	results in `R2` containing `0b111000` (the value of `R0` shifted left 2 places)
`LSR R2, R0, #1`	results in `R2` containing `0b111`
`MVN R2, R1`	results in `R2` containing `0b11111111111111111111111111110100`
`AND R1, R1, #13`	results in `R1` containing `0b1001`
`LSL R2, R0, R1`	results in `R2` containing `0b111000000000000`

Unconditional branching
Branch instructions potentially override the default behaviour of moving on to the next instruction in sequence – by specifying a new location to branch to. A branch instruction will implement this by altering the value held in the PC (Program Counter) register. Unconditional branches always make this change (akin to a 'GOTO' statement); conditional branches specify the location to move to if a specified condition is true, otherwise execution continues with the next instruction in sequence as usual.

B
`B` ('Branch') is an unconditional branch. It takes a single operand, being either a label that will be translated into a memory location by the assembler, *or* the number of instructions (words) to jump relative to the current location. For example:

`B loop1` will transfer execution to the instruction defined on the same line (or immediately below) the label `loop1`.

`B .+3` will jump forward by three instructions (i.e. skip over the next two)

`b .-5` will jump back by five instructions

The branch encoding is done relative to the current instruction (+8) but the range is +/-2^{23} bytes – far bigger than the ARMlite memory. Note that `.+n` and `.-n` calculate a distance. They do not check or count instructions in your program. (So, don't use `.asciz` "somewhere in here" for example.)

Note that, as with `LDR` and `STR`, when specifying the address as a label (first example above) the label must be aligned.

Conditional branching

The following instructions all represent forms of conditional branching: the branch is only performed if a specified condition is met, based on the values of one or more of the Status bits. If the condition is not met, execution continues with the next sequential instruction.

In all cases, these instructions take a single operand, of the same form as for the unconditional branch (`B`) instruction, described above. (If you need more complex destinations you can calculate the address in a register and then do `MOV PC,Rn`).

BEQ, BNE, BLT, BGT

'Branch if EQual', 'Branch if Not Equal', 'Branch if Less Than', 'Branch if Greater Than'. These four instructions will perform the branch based on the relationship of the two values compared by the previous `CMP` instruction (or based on the result of the previous `SUBS` or `ADDS` if that was more recent). This determination is based on the Status bits.

BCS, BVS, BMI

'Branch if Carry Set', 'Branch if oVerflow Set', 'Branch if Minus'. These instructions will perform the branch based on the value of a single Status bit: the C ('Carry') bit, the V ('oVerflow'), and the N ('Negative') bit, respectively.

Working with subroutines

BL

'Branch with Link (back)' is the operation for invoking a subroutine. It makes an unconditional branch to the specified location, but first saves, in the `LR` ('Link Register') the address of the instruction to return to when the subroutine has completed i.e. the instruction *following* the `BL`.

Be careful to save and restore `LR` if you call other subroutines.

RET

'Return (from subroutine)'. `RET` is a convenience (on ARMlite) for the instruction `MOV PC,LR`. When this instruction is encountered, the processor will copy the value of `LR` into the `PC`. The result will be that the next instruction to be executed will be one following the `BL` instruction that made the call to the subroutine.

RFE

'Return From Exception' is used on ARMlite to return from an Interrupt processing routine (see Interrupts).

PUSH, POP

`PUSH` and `POP` work with the system stack, which utilises the top part of the addressable memory. The 'stack pointer' is maintained in the `SP` register. When ARMlite is started afresh, `SP` starts at the value `0x00100000` which is `1` beyond the address of the last byte of main memory. For each word that is pushed onto the stack, `SP` is pre-decremented by `4`, so that it now points to the topmost word in the stack. `SP` is then incremented again (by `4`) after each time a value is 'popped'. This behaviour may be observed by single stepping through this simple example:

```
MOV R0, #1
PUSH {R0}
MOV R0, #2
PUSH {R0}
POP {R0}
POP {R0}
```

Observe the values of `R0` and of `SP` as you step through. Also, if you set the memory page number to `ffe` (box at the top left of the memory area) you will see the values (`1` & `2`) being temporarily stored in the stack at the very end of main memory (bottom of the screen).

`PUSH` and `POP` instructions may be used within general programming, if your code implements an algorithm that needs a stack, or if you need to hold maintain more *temporary* variables than you have available registers. (Note that if you are using Interrupts on ARMlite the system may push and pop invisibly to you.)

The most common use of `PUSH` and `POP` are in subroutines and interrupts. At the start of the subroutine, values of some, or all, of the registers are pushed onto the stack, so that those registers may be used for different purposes within the subroutine. Before exiting the subroutine (with `RET`) the original values of registers are restored by popping them from the stack. Note that you can save an instruction by pushing the `LR` (which you need to do if you call subroutines) and popping the value directly into the `PC` at the end instead of using `RET`.

`PUSH` and `POP` may take a single register as an operand. (There is nothing to stop you from deliberately pushing `R0` and popping the value back into `R1` but that would not be common practice). Usually, you would supply a list of registers, comma-separated and surrounded by curly braces. Only the general registers (R0 to R12) and the `LR` should be used with `PUSH` and `POP`, with the single exception that you can use `PC` instead of `LR` in a `POP`. For example:

`PUSH {R1-R4,R7,LR}`	will push the values of registers `R1`, `R2`, `R3`,`R4`, `R7` and `LR`.
`POP {R1-R4,R7,LR}`	will restore those same registers.

Note that if you were to push register values individually, via separate `PUSH` instructions, you would need to ensure that you pop them in reverse order, but when a list of registers is specified, as here, `POP` automatically restores them in the correct order (as long as the same register set is specified – note the lowest numbered register is pushed last).

Other

HALT

`HALT` will halt (pause) execution at a defined place. When halted you may hit continue (the run button) - though it would only make sense to do so if there were program instructions following the `HALT`. `HALT` may be used to stop the program counter from running into a data area (and producing erroneous or unpredictable result) and/or for debugging code. (A breakpoint has the same effect, without requiring modification to the code, but ARMlite supports only a single breakpoint, whereas you can have multiple HALT instructions).

ARMlite will recognise both `HALT` (the AQA instruction) and `HLT` (the ARM standard)

SVC

`SVC` stands for SuperVisor Call. It is a way of a user program generating a trap into a BIOS or an Operating System (see interrupts below) without knowing the internal structure of the BIOS or OS.

Since ARMlite does not have user modes or protection, and most programs will be small and stand-alone, it is unlikely to be used often.

Use of PC register

You can use the `PC` as a source or destination register for most ARMlite instructions (except extended `MOV` with a negative immediate which does not make any sense and overlaps with other encodings). In many cases these are discouraged or do not work on a real ARM because the CPU is reading instructions ahead of the actual execution and either there is no logic to deal with the anomalies or the CPU would have to stall to sort them out.

There are two deliberate cases that always are needed (and so work) a) for a `BL`/`JMS` (i.e. subroutine call) or interrupt, the `PC` saved is a pointer to the next instruction that would have been executed and b) for `PC` relative instructions like branch and direct addresses (which are implemented as `PC` relative). In case b) (and all other cases) the `PC` used is a pointer to the instruction after next (i.e. current instruction + 8). The assembler allows for the +8 when generating branches and direct addresses, but if you do something like `MOV R4,PC` you will get current instruction+8.

If you write to the `PC` (`B`, `JMS`, `POP`, `RTE`, `RET` and `MOV` are the only recommended instructions) then you go (jump) to the address given. (If you put an unaligned value in the `PC`, it will generally be ignored and because branches are relative, it will persist. *It is well known for assembler hackers to count to 4 in this way.*)

Directives: `.WORD .BYTE .ASCII .ASCIZ .BLOCK`

These assembler directives define memory locations, and the initial contents to be given to those memory locations. Each directive may be given its own label for convenience (where they are to be used to hold variables or constants, for example), or you might declare multiple directives under one label, and access the individual items using Indexed addressing. (Note that unlike some advanced Assemblers, you cannot do arithmetic – just one value or string is permitted.)

They are best explained by example:

`myAge: .WORD 17` defines a word of memory with the label `myAge` holding the initial value of (decimal) `17`. Note you can omit the `.WORD` and just put the number or label name. You can use a label name to get its value stored in the word location (only with `.WORD`).

`.BYTE 0x1f` defines a single byte of memory holding the initial value of `0x1f` (note that this value cannot exceed `255` or `0xff`).

`.ASCII "Hello World!"` defines an ASCII string (one character per byte). The string may be delimited either by single or double quotes. You can use `\` as an escape for `\n`, `\\`, `\'` or `\"` in the normal way.

`.ASCIZ "Hello World!"` defines an ASCII string (one character per byte) but with a 'null' (Zero value) byte added on the end, which is not a printable character. This is how you would typically define a string message to be written to the console using `.WriteString` (see Console) because the zero byte will indicate when ARMlite should stop reading characters.

`myName: .BLOCK 128` will block out the next 128 bytes (16 words) to be used for data and assigns `myName` to the start of the block . (The assembler zeros all memory so they will be zero the first time you run the program.)

Directives: .ALIGN .DATA

These directives control the assembler. If you give a `.ALIGN` a label it will take the address of the *start* of the align. If you give .data a label it will be attached to the following instruction (or give an error if that has its own label).

`.ALIGN 256` will align the *following* instruction or data to the *next* byte address that is divisible by `256` i.e. the next *page* of memory if necessary. (Note the assembler will align instructions and `.WORD` by 4 automatically.)

`.DATA` sets the limit of code that is protected from modification by the program (`STR` or `STRB`). By default, the assembler sets this before the first data declaration (.byte, .word or .block) but you can use this directive to increase or reduce that. It is recommended to put all invariant declarations immediately after the instructions and followed by a `.DATA`.

Input/Output

Console

The simplest way to output text is to the Console, where new text is always appended to any existing text, and the output scrolls when it exceeds 4 lines. In the following examples, `R0` contains the decimal value `3000000000`:

`STR R0,.WriteUnsignedNum`	writes out `3000000000`
`STR R0,.WriteSignedNum`	writes out -`1294967296` (value interpreted as two's complement)
`STR R0,.WriteHex`	writes out `0xb2d05e00`

`MOV R1,#65`
`STRB R1,.WriteChar` writes out the character `A` (ASCII 65). It is recommended that you use `STRB` to make it clear that you are writing a single character, defined by the least significant 8 bits in the register. (STR will work, but will still write only one character.)

```
MOV R1,#0x0A
STRB R1,.WriteChar  //writes a 'Newline' character
MOV R2,#msg
STR R2,.WriteString
HALT
msg:   .ASCIZ "Hello World!"
```

In the example above # indicates that it is *not* the message contents that is moved into R0 (which would not work anyway) but the *memory address* for the start of the string.

Input

Input may be requested from the console as follows:

`LDR R0,.InputNum` requests user to enter a number, which goes into `R0`. The number may be entered as decimal (signed or unsigned), hex, or binary (with prefixes). However, it must be an integer, and within the range that may be represented in one word.

`MOV R0,#myName`
`STR R0,.ReadString` expects the user to enter a string (of up to 127 characters). On hitting the Return key, the string will be transferred, one byte per character, to the memory addresses starting at the address that has been specified in `R0` (set up in the line above to the address of the `myName` label. At the end of the string, the system will automatically write a null (Zero byte).

`MOV R0, #myName`
`STR R0, .ReadSecret` works as `.ReadString` but the input typed is hidden

Note that you should define `myName` as
`myName: .BLOCK 128` or more because 128 bytes is the limit that ARMlite will transfer.

Reading key presses

For interactive applications such as games, it is useful for the user to be able to hit keys without requiring the program to pause and request an input.

`LDR R0,.LastKey` will load into `R0` the key code for the most recent key pressed (since the program has been running). `LDRB` may also be used, since the value takes just one byte.

`LDR R0,.LastKeyAndReset` does the same, but clears the record of that keypress. Calling it again without the user having pressed another key would result in a zero value in `R0`.

The Key Code for `a` (or `A`) is 65 and for `1` is 49. By default, only numbers and letters are enabled. (See `.KeyboardMask` for how to enable other keys but beware that using any key with a Browser function may behave unpredictably and most punctuation characters give something completely obscure.)

Reading/Writing files

`LDR R0,.OpenFile` opens a file to read (for security though a user selection box) and puts the length of the file in bytes) into `R0`, or -1 if the operation fails. (Note that ARMlite uses your browser to read and write files and the exact behaviour depends your browser settings. If the browser returns an error to ARMlite you will get -1 in the register, but it seems more normal for ARMlite to be stalled until you provide a valid file.)

`LDRB R1,.ReadFileChar` reads the next character of the file into `R1`. (`LDR` will also work but you only get one character)

`LDR R0,.FileLength` reads the number of characters remaining to be read in the file, into `R0`.

`STR R3,.WriteFileChar` adds the contents of `R3` as four characters to the file output buffer. (`STRB` would write the lowest 8 bits from `R3` as a single character).

`MOV R4,#myFileName`
`STR R4,.WriteFile` writes the characters already in the buffer to a file. The file name to be used is specified as a string, with `R4` pointing to the start of the string (which should zero-terminated, for example by declaring the filename using `.ASCIZ`). (Note that the Browser should ask you to select or confirm the name and location unless your browser settings say otherwise.)

`MOV R5,#-1`
`STR R5,.WriteFile` clears output buffer without writing anything.

Pixel graphics

The pixel graphics display is the lower of the three Input/Output areas and has three modes of operation. In all cases `STR` is used to write a 24 bit HTML Colour Code from a register to a pixel address (the top 8 bits of the register are ignored).

Low-res Mode – Direct Addressed

This is a 32-column x 24-row mapping with the individual blocks (of pixels) named as `.Pixel0` through to `.Pixel767`. So:

```
mov r0,#.green
str r0,.Pixel0
str r0,.Pixel31
str r0,.Pixel736
str r0,.Pixel767
halt
```

writes a green block to each of the four corners of the area. You could similarly treat `.Pixel0` as the start of the area and use indirect, or indexed addressing in this example:

```
      mov r0,#.red
      mov r1,#.Pixel0
      mov r2, #0
loop: str r0,[r1+r2]
      add r2,r2,#4
      cmp r2,#128
      blt loop
      halt
```

Notes:

- Low-res Mode (Direct Addressed) is always available in parallel with one of the other two modes
- You can read `.Pixelnn` (using `LDR`) and it will be `0xffffff` initially (=white) or whatever value you wrote to it (Even if you changed the actual colour displayed using a different mode).

Mid-res Mode – 64 x 48

Both other modes are mapped to memory that starts at the address `.PixelScreen`. By default, the display is configured to 'mid-res', a 64-column x 48-row grid of pixels.

For example (at default resolution):

```
MOV R0, #0xff0000
MOV R1, #.PixelScreen
STR R0, [R1]                Sets the top-left pixel to pure red.

MOV R0, #.darkgreen
MOV R1, #.PixelScreen
ADD R2, R1, #256
STR R0, [R2]                sets the next pixel down to dark green
STR  R0, .ClearScreen       clears the pixel area (the register value is ignored)
```

Clicking the pixel area

It is possible to enable the pixels for interrupt or polling when clicked (see also `.PixelMask` and `.PixelISR` in the Interrupts section). To find out which pixel has been clicked on interrupt read (`LDR` only) `.LastPixelClicked`. If you are polling, then you can test bit `2` of `.PixelMask` and use (`LDR` only) `.LastPixelAndReset` to read the value. (You could poll `.LastPixelAndReset` – you will get `-1` if there has been no click.) Note polling must be enabled by setting `.PixelMask` to `2`.

Note that the screen is cleared when you first enable (or completely disable) pixel clicks. If the screen is in default mid-res mode `.LastPixelClicked` will give you a value between `0` and `3071` (or `-1` if no clicks yet). In hi-res mode the range is `0` to `12287`. You do not have to write anything – the blank pixels are clickable once you have done the enable.

Hi-res Mode 128 x 96

Hi-res mode can be set by the program as follows:

```
LDR R0, #2                  use 1 for mid-res and 2 for hi-res.
STR R0, .Resolution          (this also clears the pixel area)
```

A program can read `.PixelAreaSize` (using `LDR`) and will get either `3072` or `12288` to show what is configured.

Hi-res mode works in the same manner as 64 x 48 mode.

(Note you can use `LDR` to read a pixel value in mid-res or hi-res modes and you will get the displayed value, even if it was set by low-res mode.)

HTML Colour Codes

To assist with the HTML colour codes, the following are recognised by the assembler:

`.background (white), .aliceblue, .antiquewhite, .aqua, .aquamarine, .azure, .beige, .bisque, .black, .blanchedalmond, .blue, .blueviolet, .brown, .burlywood, .cadetblue, .chartreuse, .chocolate, .coral, .cornflowerblue, .cornsilk, .crimson, .cyan, .darkblue, .darkcyan, .darkgoldenrod, .darkgray, .darkgreen, .darkgrey, .darkkhaki, .darkmagenta, .darkolivegreen, .darkorange, .darkorchid, .darkred, .darksalmon, .darkseagreen, .darkslateblue, .darkslategray, .darkslategrey, .darkturquoise, .darkviolet, .deeppink, .deepskyblue, .dimgray, .dimgrey, .dodgerblue, .firebrick, .floralwhite, .forestgreen, .fuchsia, .gainsboro, .ghostwhite, .gold, .goldenrod, .gray, .green, .greenyellow, .grey, .honeydew, .hotpink, .indianred, .indigo, .ivory, .khaki, .lavender, .lavenderblush, .lawngreen, .lemonchiffon, .lightblue, .lightcoral, .lightcyan, .lightgoldenrodyellow, .lightgray, .lightgreen, .lightgrey, .lightpink, .lightsalmon, .lightseagreen, .lightskyblue, .lightslategray, .lightslategrey, .lightsteelblue, .lightyellow, .lime, .limegreen, .linen, .magenta, .maroon, .mediumaquamarine, .mediumblue, .mediumorchid, .mediumpurple, .mediumseagreen, .mediumslateblue, .mediumspringgreen, .mediumturquoise, .mediumvioletred, .midnightblue, .mintcream, .mistyrose, .moccasin, .navajowhite, .navy, .oldlace, .olive, .olivedrab, .orange, .orangered, .orchid, .palegoldenrod, .palegreen, .paleturquoise, .palevioletred, .papayawhip, .peachpuff, .peru, .pink, .plum, .powderblue, .purple, .red, .rosybrown, .royalblue, .saddlebrown, .salmon, .sandybrown, .seagreen, .seashell, .sienna, .silver, .skyblue, .slateblue, .slategray, .slategrey, .snow, .springgreen, .steelblue, .tan, .teal, .thistle, .tomato, .turquoise, .violet, .wheat, .white, .whitesmoke, .yellow, .yellowgreen`

Character display

There is a 32 x 16 memory-mapped character display that overlays the pixel graphic display, to allow text and graphics to be displayed in the same screen.

```
MOV R0,#65
MOV R1,#.CharScreen
STRB R0,[R1]          Writes the character A at the top left of the screen
```

```
MOV R0, #66
MOV R1, #.CharScreen
ADD R2, R1, #1
STRB R0,[R2]          Writes the character B next to the A
```

Note that each character occupies one byte (in contrast to the pixel display where each pixel occupies four bytes). With the character screen it is possible to set up four characters in one register and write all four to consecutive positions using `STR` instead of `STRB` - but it is usually simpler to work with single characters and `STRB`.

Only printable ASCII characters are shown. Pixels set using `.PixelScreen` will show up underneath text that occupies the same part of the display.

Note that `.ClearScreen` clears both the pixel and the character area and the character area is disabled when you enable clicking on the pixel area. You can use `LDRB` or `LDR` to read values you have written, and you will get zero if you read before writing.

Other interactions with the system

.Random

`LDR R0,.Random` will generate a random 32-bit pattern and put it in `R0`. Bitwise manipulation operations may be used to reduce its range.

.Time

`LDR R0,.Time` loads the current time into `R0`, as the number of seconds elapsed since 1st Jan 2000.

.InstructionCount

`LDR R0,.InstructionCount` gives the number of instructions executed since the program was first run.

.CodeLimit

`LDR R0,.CodeLimit` gives the highest memory address used for storing the program (including any data areas specified in assembly code). You will get an error if the stack attempts to write to memory below this value.

Interrupts

ARMlite implements simple interrupts, and the similar `SVC` call.

To use any of these you must first set up an interrupt service routine (ISR) for the ones you need. Five are available (write with `STR`):

`.SysISR` – system call handler

`.PinISR` – handler for the clickable button

`.KeyboardISR` – handler for keyboard input

`.ClockISR` – handler for clock interrupts

`.PixelISR` – handler for clicks on the pixel area

There are several control (I/O) registers. (`SVC` is an instruction and so is always enabled (but you will get an error if the ISR is not setup). The others require enabling (write with `STR`):

`.ClockInterruptFrequency` - `0`=off, otherwise the number of milliseconds. E.g. for an interrupt every second set 1000. Note that it is an “interval” (i.e. from the RFE or enable) not absolute time.

`.KeyboardMask` – the low bit is interrupts on/off (on interrupt use `.LastKey` for the keyboard position). Bit 1 (as `1`) enables the arrow keys and bit `2` enables all keys (incl. function keys!).

`.PinMask` – the low bit is interrupts on/off for the clickable button. (This also has a poll option – to use set bit `1` to enable polling and bit `2` is set when clicked – use `LDR`.)

`.PixelMask` – the low bit is interrupts on/off for clicks on the pixel area. (This also has a poll option to use set bit `1` to enable polling and bit `2` is set when clicked – use `LDR`.)

`.InterruptRegister` is the `master` interrupt enable – the low bit (as 1) enables interrupts.

To use interrupts, you need to set up one or more ISR, enable the interrupts you need, and then set the master interrupt enable. On interrupt, the stack pointer must be valid and two words are pushed onto the stack. First the flags and a record of the master interrupt bit (which will be set to `1`, apart from on an `SVC` call, which may be either) and then the program counter (i.e. the address to go back to). (Note a real ARM does not do exactly this but the effect is the same.) The master interrupt bit is cleared to prevent further interrupts and the `PC` is set to the ISR address.

At the end of the interrupt service routine it should execute an `RFE` (Return From Exception) instruction which will restore the `PC`, the flags and the master interrupt bit to the values before the interrupt (or the `SVC`). Note that `RFE` unstacks two words (as pushed on interrupt) whereas `RET` would take the `LR` and not restore the flags or the interrupt status (as well as leaving the stack in the wrong place).

In anything other than trivial examples, interrupt routines need to save and restore any registers they use – particularly `LR` if they call functions. (You can get some very hard to find bugs if clock or other interrupts change registers that the interrupted program is using.)

Simulation configuration

There are a few options which can be set as query parameters in the URL request – for example `?data=unsigned&mem_k=64`. They are:

`alcom`	Controls how comments are aligned when following an instruction. The default (if not present) is 24 characters (after the line number). The range can be set between 6 and 300.
`binsiz`	When binary is selected for the display of registers and memory, some platforms need fontSize to be reduced and this is reduced from 100% to 92% on platforms other than Win32. binsiz takes a value between 80 and 100 to set the percentage reduction of registers and memory in binary mode.
`data`	Controls the way memory and register contents are displayed. The options are unsigned, signed, hex, binary and decimal (same as signed). The default is hex.
`debug`	Enables extra debugging messages (mainly alerts). Default is `0`.
`mem_k`	Sets the main memory size. Range `1` to `1024`, default `1024` (for 1M bytes).
`profile`	Adds the class "profile-xxxx" to the body tag, no default. See Profiles, below. (Note any profile setting sets a fixed width Program area.)
`progw`	By default the width of the Program area depends on the size of the browser window with a minimum of 300 pixels. This option allows a fixed width to be set between 300 and 3000.
`slow_delay`	Controls the speed of execution in "Slow" mode. Range 2 to 999, default 40. Setting 999 gives an instruction approximately every 4 seconds and setting 2 gives you about 130 instructions/second (the maximum speed without turning the display off). (Note you can also click the Slow button multiple times to speed up dynamically.)

Profiles

This is an area for future expansion. At present there is one profile option, `player` (other than the default) which can be selected by appending this query string to the URL:

```
?profile=player
```

This profile is intended for users of an application (typically a game) written for ARMlite. It differs from the default view as follows:

- The only visible program control is **Load**, allowing a program file to be loaded and submitted. (It follows that the program file must not generate any submission errors.)
- The run controls show **Play** (Run), **Pause**, and **Stop** only.
- The Pixel and Char screens are magnified x2 and shown to the right of the other controls.

References

Assembly Language Programming by Richard Pawson with Peter Higginson
https://community.computingatschool.org.uk/resources/6172/single

ARMlite Experimental and Advanced Features https://peterhigginson.co.uk/ARMlite/ARMlite Experimental and Advanced Features.pdf