ARMv8 assembler – Aarch32

Introduction

The assembler here is based on the integral of wabiForth. So sometimes there are references to that system, for which I apologise.

The assembler is pretty complete, missing are NEON opcodes, and literal pools. The NEON opcodes will be added in future. The literal pools have lost there relevance with modern ARM-processors and will not be added. On the following pages the available assembler words are listed.

Adapting the assembler to your own environment

The assembler comma's the opcodes at HERE using the word EOFOPC. It depends on the processor you use, but this will probably not function correctly in your system. It usually is necessary to manage the cache-system of your ARM-processor. This is essential as the ARM processor has no cache-coherency between the instruction and data-caches at all.
The way caches are managed is system specific so you have to adapt EOFOPC part yourself. In wabiForth there is a word **OPCODE**, which, among other functionality, comma's an opcode and manages the caches.

Format and syntax:

The assembler follows the established way of assembling in Forth by having the registers followed by the opcode.

An example:

```
ADD r1, r2, r3     \ add r2 and r3 with the result in r1
```

in the standard ARM-assembler notation.

Would become:

```
r1, r2, r3, ADD,
```

Notice the comma's after every word in the assembler. They are there to avoid confusion between Forth definitions and assembler definitions.

There are only a few exceptions to the 'comma'-convention:

- destination labels end with a colon (fi: **label0:**)
- ordinary numbers are written as usual (fi: **23**)
- the word **LBL[** has no comma as it manages the label-system and does not assembly anything

Assembler words are not immediate. This means that you have to put them between brackets within a new definition.
There are pro's and con's to having the words compiling or immediate. The big advantage of normal compiling words is that it is much easier to develop macro's. No need to resort to endless POSTPONE statements.
The disadvantage is that you have to put brackets ('[' and ']') around blocks of assembly words.

ASSEMBLER WORDS

Control words:

ASM[(--): initiates the assembler, must be used once before using the assembler.

LBL[(--): resets the label-addresses. Normally used once per definition, but if labels are not used inside a definition than the reset can be left out. A label-reset makes further resolution of previous jumps impossible, so make sure that each branch has the corresponding label. Multiple definitions can share labels. This makes it possible to branch from one definition to a label inside another.

Assembling words:

Registers:

The ARM processor in ARM32 mode has 16 registers. Register 13 to 15 each have two names. These names are synonyms and can be mixed.

```
       r0,
       r1,
       r2,
       r3,
       r4,
       r5,
       r6,
       r7,
       r8,
       r9,
       r10,
       r11,
       r12,
       r13,  =    sp,
       r14,  =    lr,
       r15,  =    pc,
```

Conditional execution:

```
       eq,
       ne,
       cs,   =    hs,
       cc,   =    lo,
       mi,
       pl,
       vs,
       vc,
       hi,
       ls,
       ge,
       lt,
       gt,
       le,
       al,  ( default, optional )
```

Data Processing Opcodes:

```
       and,       ands,
       eor,       eors,
       sub,       subs,
       rsb,       rsbs,
       add,       adds,
       adc,      adcs,
       sbc,      sbcs,
       rsc,      rscs,
       orr,      orrs,
       mov,      movs,
       bic,      bics,
       mvn,      mvns,

       tst,
       teq,
       cmp,
       cmn,
```

Operand 2:
This is one of the the signature features of the ARM-processor.

Operand 2 can either be:

- register
- register with it's content shifted immediately
- register with it's content shifted defined by another reg
- immediate, a 8 bit value rolled right with 0, 2, 4, etc

```
        I#,       (expects two numbers on the stack!! see examples)

        lsl#,
        lsr#,
        asr#,
        ror#,
        lslr,
        lsrr,
        asrr,
        rorr,
```

16b immediate:

```
        movt,
        movw,
```

Multiply:

```
        mul,      muls,
        mls,
        mla,      mlas,
        smull,    smulls,
        smlal,    smlals,
        umull,    umulls,
        umlal,    umlals,
        umaal,

        smlabb,   smlabt,    smlatb,    smlatt,
        smlad,    smladx,
        smlalbb,  smlalbt,   smlaltb,   smlaltt,
        smlald,   smlaldx,
        smlawb,   smlawt,
        smlsd,    smlsdx,
        smmla,    smmlar,
        smmls,    smmlsr,
        smmul,    smmulr,
        smuad,    smuadx,
        smulbb,   smulbt,    smultb,    smultt,
        smulwb,   smulwt,
        smusd,    smusdx,
        smlsld,   smlsldx,
```

Division:

```
       sdiv,
       udiv,
```

Load and Store: (all 40 available opcodes)

```
       ldr,       str,
       ldrb,      strb,
       ldrh,      strh,
       ldrsh,
       ldrsb,
       ldrd,      strd,
```

Load and Store acquire/exclusive

```
       lda,       stl,
       ldab,      stlb,
       ldah,      stlh,

       ldaex,     stlex,
       ldaexb,    stlexb,
       ldaexh,    stlexh,
       ldaexd,    stlexd,

       ldrex,     strex,
       ldrexb,    strexb,
       ldrexh,    strexh,
       ldrexd,    strexd,

       ldrt,      strt,
       ldrbt,     strbt,

       ldrht,     strht,
       ldrsht,
       ldrsbt,
```

Addressing modes:

```
       +],        -],
       +]!,       -]!,
       ]+!,       ]-!,
       i+],       i-],
       i+]!,      i-]!,
       ]i+!,      ]i-!,
```

Block Load and Store:

```
       ldmed,
       ldmib,
       ldmfd,
       ldmia,
       ldmea,
       ldmdb,
       ldmfa,
       ldmda,

       stmfa,
       stmib,
       stmea,
       stmia,
       stmfd,
       stmdb,
       stmed,
       stmda,
```

Register-block definition:

```
        {,
        r-r,

        },
        }!,
        }^,
        }!^,
various:
        sel,
        clz,
        usad8,
        usada8,
        clrex,
        setendle,
        setendbe,


Processor state:
        cps,
        cpsid,
        cpsie,

        processor modes
        user,
        fiq,
        irq,
        supervisor,
        monitor,
        abort,
        hyp,
        undefined,
        system,

        a,i,f flags
        <a>, <ai>, <af>, <aif>, <i>, <if>, <f>,

CoProcessor:
        mcr,
        mcrr,
        mrc,
        mrrc,

        CoProcessor registers:
        p14, p15,
        c0, c1, c2, c3, c4, c5, c6, c7, c8,
        c9, c10, c11, c12, c13, c14, c15,

Move to/from special registers:
        mrs,
        msr, ( only reg to banked_reg - no immediate )

        special registers:
        spsr, cpsr, apsr, ( only for mrs, !! )
        r8_usr, r9_usr, r10_usr, r11_usr, r12_usr, sp_usr, lr_usr,
        r8_fiq, r9_fiq, r10_fiq, r11_fiq, r12_fiq, sp_fiq, lr_fiq,
        lr_irq, sp_irq,
        lr_svc, sp_svc,
        lr_abt, sp_abt,
        lr_und, sp_und,
        lr_mon, sp_mon,
        elr_hyp, sp_hyp,
        spsr_fiq,
        spsr_irq,
        spsr_svc,
        spsr_abt,
        spsr_und,
        spsr_mon,
        spsr_hyp,
```

External debugging:

```
       ldc,      Rn, 8bIMM, {index_mode} ldc, -> 8b immediate
                 mandatory for every mode of opcode. Leave out p14 and
                 c5.
       stc,      ditto
```

Barriers:

```
       csdb, ( no options )
       pssbb, ( no options )

       dmb,
       dsb,
       isb,
```

options for domain of memory barriers:

```
            sy, ( default, optional )
            st,
            ld,
            ish,
            ishst,
            ishld,
            nsh,
            nshst,
            nshld,
            osh,
            oshst,
            oshld,
```

Hints:

```
       sev,
       sevl,
       wfe,
       wfi,
       yield,
       nop,

       pldw,     for literal variant use pc,
       pld,      ditto
       pli,      ditto
```

Exception handling:

```
       eret,
       bkpt,     16b value before opcode mandatory - no condition
       hvc,      ditto
       smc,       4b value before opcode mandatory - condition allowed
       svc,      24b value before opcode mandatory - condition allowed


       rfe,      rfe!,   \ rfe & refia are synonyms
       rfeia,    rfeia!,
       rfeda,    rfeda!,
       rfedb,    rfedb!,
       rfeib,    rfeib!,

       \ for all following: put processor mode before opcode
            -> fi: hyp, srsia,
       srs,      srs!,   \ srs and srsia are synonyms
       srsia,     srsia!,
       srsda,     srsda!,
       srsdb,     srsdb!,
       srsib,     srsib!,
```

Cyclic redundancy check:

```
       crc32w, =  crc32,
       crc32h,
       crc32b,
       crc32cw, = crc32c,
       crc32ch,
       crc32cb,
```

Bitfield manipulation:

```
       bfc,
       bfi,
       sbfx,
       ubfx,
```

Extent (and add): use ror#, to specify which byte – 0 is default

```
       sxtb,
       sxtab,
       sxth,
       sxtah,
       uxtb,
       uxtab,
       uxth,
       uxtah,
```

SIMD extent (and add):

```
       sxtb16,
       sxtab16,
       uxtb16,
       uxtab16,
```

Saturated:

```
       qadd,
       qdadd,
       qsub,
       qdsub,

       ssat,
       usat,
       ssat16,
       usat16,
```

Packing and unpacking:

```
       pkhbt,
       pkhtb,
```

Parallel additions and subraction:

```
       sadd16, qadd16, shadd16, uadd16, uqadd16, uhadd16,
       sadd8,  qadd8,  shadd8,  uadd8,  uqadd8,  uhadd8,
       sasx,   qasx,   shasx,   uasx,   uqasx,   uhasx,
       ssax,   qsax,   shsax,   usax,   uqsax,   uhsax,
       ssub16, qsub16, shsub16, usub16, uqsub16, uhsub16,
       ssub8,  qsub8,  shsub8,  usub8,  uqsub8,  uhsub8,
```

Byte and Bit Reverse:

```
       rev,
       rev16,
       rbit,
       revsh,
```

Branch:

bx, (reg --) - branches to address in register

blx, (reg --) - linked branch to address in register - this is a jump to subroutine to the address in register

Conditional Branch:

Some branches have two different names for the same function. They are put next to each other in the following list

```
beq,
bne,
bcs,  = bhs,
bcc,  = blo,
bmi,
bpl,
bvs,
bvc,
bhi,
bls,
bge,
blt,
bgt,
ble,
b,    = bal,

bleq,
blne,
blcs, = blhs,
blcc, = bllo,
blmi,
blpl,
blvs,
blvc,
blhi,
blls,
blge,
bllt,
blgt,
blle,
bl,   = blal,
```

Labels:

There are 16 destination labels. The word LBL[resets these. All 16 can be used within each new definition. Technically it is possible, but with limitations, to use the same destination-label more than once within a definition. But that is an advanced topic, and with 16 destination-labels it should hardly ever be necessary.

LBL[(--): resets the label-addresses.

```
label0:
label1:
label2:
label3:
label4:
label5:
label6:
label7:
label8:
label9:
label10:
label11:
label12:
```

```
label13:
label14:
label15:
```

There are 16 source-labels. One for each destination-label. Source-labels can be used as often as needed within an individual definition. They are put in front of a branch-opcode.

```
>lb0,
>lb1,
>lb2,
>lb3,
>lb4,
>lb5,
>lb6,
>lb7,
>lb8,
>lb9,
>lb10,
>lb11,
>lb12,
>lb13,
>lb14,
>lb15,
```

Examples of assembly-code

This example shows how **conditional execution** saves opcodes and branches. With 3 opcodes, ascii characters are converted to capitals:

```
: 2caps [
  v, top, 0 97 i#, sub,
  v, 0 26 i#, cmp,
  top, top, 0 32 i#, lo, sub,
] ;
```

This example is ROT in assembly. It shows the use of **normal and immediate indexed addressing** for Load and Store opcodes:

```
: myrot [
  v, top, mov,
  w, dts, ldr,
  top, dts, 4 i+], ldr,
  v, dts, str,
  w, dts, 4 i+], str,
] ;
```

This is an example of **an optimising word.** It combines the function of the words ROT and DROP into 1 word to reduce execution time. ROT uses 5 opcodes and DROP uses 1 opcode. But combined into 1 word, ROTDROP, only 2 opcodes are needed. And this combined word is 3 cycles faster than ROT DROP separately. The example also shows the use of CODE: and INLINABLE. If these are not used then the word is slower than ROT and DROP. It also shows the use of the 'post-increment with an immediate' addressing-mode.

```
code: rotdrop [
  v, dts, 4 ]i+!, ldr,
  v, dts, str,
] ;
```

The following 2 examples compare a conditional branch vs conditional execution. Both routines perform the same action, namely they check if the top is equal to 23. But one uses a conditional branch and one uses conditional execution. The latter is the shorter and faster option.

Example with **conditional branch** - around 4 cycles execution-time

```
code: 23=
lbl[ [                \ lvl[ resets the labels
  top, 0 23 i#, cmp,
  top, 0 movw,
  >lb1, bne,          \ branch on not-equal to label1
  top, 0 0 i#, mvn,
label1:
] ;
```

Example with **conditional execution** - ~0-2 cycles execution time

Please note that under certain circumstances this code effectually runs in zero cycles. This happens when this code can be run while the CPU is doing something else. This is called "executing in the shadow". The programmer has no direct influence on this, other than making sure that complex opcodes (for instance loads and stores) are located before fast code. With a bit of luck the fast code is executed in effectually zero cycles.

```
code: fast23= [
  top, 0 23 i#, cmp,
  top, 0 0 i#, ne, mov,
  top, 0 0 i#, eq, mvn,
] ;
```

Example with **2@** (**twofetch**) which does NOT need aligned addresses. The **2@** implemented in wabiForth can only read data from a word-aligned address. (see the chapter on Hardware specific wabiFORTH limitations). The following code implements a version of 2@ which is a bit slower but handles unaligned addresses correctly. It is faster than an implementation in wabiForth.

```
code: 2@nonaligned [
  v, top, 4 i+]!, ldr,
  w, top, ldr,
  top, v, mov,
  w, dts, 4 i-]!, str,
] ;
```

Example of an assembly-macro. The following example shows how to make a simple macro. On the ARM-processor getting an 32b immediate into a reg takes two opcodes, MOVW and MOVT. This macro takes a 32b value from stack and generates the two opcodes directly.

```
: ldv32, ( reg n -- )     \ no square brackets!!!
  2dup
    16 lshift 16 rshift movw,
  16 rshift movt,
;
```

Please note that square brackets are not used in a macro!

It is also good to know that on a ARMv8 processor movw, followed by movt, is the fastest way of getting an immediate into a register. On the Cortex-A53, the processor in the RPi 3b+, these two opcodes run in parallel, when put in the order MOVW directly followed by MOVT. And in that specific case together take only 1 cycle execution-time.

LDV32, can be used as a normal opcode in the assembler. The following silly example shows how to put the number 123456789 into the top register.

```
: tld [ top, 123456789 ldv32, ] ;
```

If your enter:

```
  0 tld .
```

123456789 will be printed as tld changes the 0 into 123456789.

Example of the use of **shifts for operand 2.** The word implements a 1-seed random-generator according to Marsaglia.

First the function is implemented using wabiForth:

```
variable seed
2345 seed !

: forthrandom ( address_seed - rndm_val ) ( 38c )
  dup >r @
  dup 5 lshift xor
  dup 9 rshift xor
  dup 7 lshift xor
  dup r> ! ;
```

The same routine, but now assembly. Using assembly it is 3.5 times faster.

```
variable seed
2345 seed !

code: asmrandom ( address_seed - rndm_val ) ( 10c )
[ w, top, ldr,
  w, w, w, 5 lsl#, eor,
  w, w, w, 9 lsr#, eor,
  w, w, w, 7 lsl#, eor,
  w, top, str,
  top, w, mov, ]
;
```

Example of the **signed byte extend** opcode. In wabiForth it is used to get the IMMEDIATE flag (a signed byte in the header) for a given XT and converts it to a valid 32b value.

```
  code: getimmediate ( xt -- flag )
  [ v, top, 35 i-], ldrb,
    top, v, sxtb, ]
  ;
```

Adding and subtracting a value to/from a double number. The example especially shows the use of ADDS (instead of ADD) to set the carry. Also note the two values in front of **i#,** to construct the values 1 and 4. The wabiForth assembler uses the two values seperately rather than combine into the one resulting value. Any other value which fits in the 4_8 immediate scheme can be used.

adding 1 to a double:

```
code: 1md+ ( d -- d+1 )
[ w, dts, ldr,
  w, w, 0 1 i#, adds,
  w, dts, str,
  top, top, 0 0 i#, adc, ]
;
```

subtracting 4 from a double:

```
code: 4md- ( d -- d-4 )
[ w, dts, ldr,
  w, w, 0 4 i#, subs,
  w, dts, str,
  top, top, 0 0 i#, sbc, ]
;
```

Reading data from a 64b system-register. This example reads data from the **PMCCNTR** register. This is a 64bit counter which counts the actual number of CPU-cycles. The data is put as a double on the stack. The function only needs 3 opcodes.

```
code: cycles ( -- no_of_cpu_cycles )
[ top, dts, 4 i-]!, str,
  p15, 0 w, top, c9, mrrc,
  w, dts, 4 i-]!, str, ]
;
```

Example of the use of saturating aritmetic. Watch out where to put a comma and where not!

To Be Worked Out!!!

```
r0, 4 r4, 3 asr#, ssat,
```

Example of a longer assembly routine. This is the **sieve-benchmark** as published by BYTE in 1983. It counts the number of primes upto 16k and does that 10 times. In 1983 the fastest computer in existence, the CRAY-1, did this, using Fortran, in 0.11 seconds. The Raspberry Pi, using assembly, does it in 877 micro seconds. Which is 126 times faster. It shows the unbelievable progress computing has made.

This routine is not picked as an example because of it's beauty. Rather because it shows a number of useful things when using the assembler.
For instance it is a nice example of mixing assembly and normal wabiForth words. It shows the use of brackets around blocks of assembly words, even if a block only consists of 1 opcode. It also shows the use of macro's. And how to save and restore registers in an assembly routine.

```
\ the assembler macro's -----
: prologmax r13, {, r0, r4, r-r, r6, r8, r-r, v, w, }!,
  stmfd, ;
: nextmax r13, {, r0, r4, r-r, r6, r8, r-r, v, w, }!,
  ldmfd, ;
: pushdts ( reg -- )       \ pushes register on top
    top, dts, 4 i-]!, str,
    top, swap mov, ;

lbl[                       \ reset of labels
```

```
\ ten times the sieve -----
: asmsieve \ ( odd# -- #primes, mcs )
[ prologmax ]             \ brackets needed!!
  mcs swap
  dup 8 +
  allocate                \ array in high memory

[ r1, top, mov,           \ top in r1
  top, dts, 4 ]i+!, ldr,  \ update stack position

  r6, top, mov,           \ top in r6
  top, dts, 4 ]i+!, ldr,  \ update stack position

  r0, 10 movw,            \ r0=M -> 10 iterations

\ do 10 iterations
label1:
  r2, 0 movw,             \ zero for each iteration
  top, dts, 4 i-]!, str,
  top, r1, mov,           \ r1=start array

  w, r6, 2 lsr#, mov,     \ w=r6 right shifted by 2
  w, w, 0 1 i#, add,
  top, dts, 4 i-]!, str,
  top, w, mov, ]          \ push content of w on stack

  [hex] 01010101 [decimal]
  fill                    \ flags in array

[ r3, 0 movw,
  r4, 0 movw,

label3:
  r7, r1, r3, +], ldrb,   \ get r3'th element
  r7, 0 0 i#, cmp,
  >lb4, beq,

  r7, r3, r3, add,
  r7, r7, 0 3 i#, add,    \ prime is i+i+3
  r8, r7, r3, add,        \ k=prime+i

label6:
  r8, r6, cmp,            \ if k>odd#
  >lb5, bgt,

  r4, r1, r8, +], strb,   \ flags[k]=0
  r8, r8, r7, add,
  >lb6, b,                \ repeat if k<=no of odd#

label5:
  r2, r2, 0 1 i#, add,    \ r2=count of primes
```

```
label4:
  r3, r3, 0 1 i#, add,
  r3, r6, cmp,
  >lb3, blt,               \ end sieve-loop

  r0, r0, 0 1 i#, subs,    \ 1 from r0=m
  >lb1, bne,   ]           \ end 10*-loop

  mcs swap -
[ r2, pushdts ]            \ primes# on stack
  swap
[ nextmax ] ;
```

Enter:

```
8190 asmsieve . .
```

and you will get something like 877 and 1899 printed. The first number is the time in microseconds to complete 10 counts. The second is the actual count of primes.