18sep2023

ARM Thumb2 notation in noForth t

Register names in noForth

Take care:
In noForth `sp` is the data stack pointer (R1), `rp` is the return stack pointer (R13).

```
ip  sp  w   tos hop day sun moon    Lo-registers
R0  R1  R2  R3  R4  R5  R6  R7
```

`ip` noForth instruction pointer
`sp` data stack pointer
`tos` top of data stack

`w hop day sun` and `moon` are used by noForth but you can use them freely in CODE definitions.

```
ww  xx  yy  zz  does rp  lr  pc     Hi-registers
R8  R9  R10 R11 R12  SP  LR  PC
```

`does` address of DOES routine
`rp` return stack pointer
`lr` link register
`pc` program counter

`ww xx yy` and `zz` are not used by noForth.

General instruction format

- Operands are seperated by spaces instead of commas.
- The instruction name comes after the operands.
- Instruction names get a comma at the end.

```
tos day ands,          ANDS R3,R5
```

Addressing

See also the Overview at the end of this document.

Indirect addressing for loads and stores

```
day sun moon r)        R5,[R6,R7]
day sun 8 #)           R5[R6,#8]
{ day sun moon }       {R5,R6,R7}
```

Immediate literals

```
tos FF # adds,         ADDS R3,#0xFF
```

Readability

`r) #)` and `#` are there for readability, they are optional and can be omitted.

Macros

Defining forth words in assembly

```
1) CODE CELL+ ( adr -- adr+4 )
2)    tos 1 cells # adds,   \ tos ← tos+4
3)    NEXT,
4) END-CODE
```

1) `CODE` starts an assembly definition with the name CELL+ .
2) Assembly code. Note that you may use forth words between assembly words:
   `1 CELLS` leaves 4 on the stack.
3) `NEXT,` compiles the return to the caller of the word CELL+ .
4) `END-CODE` does some stack tests. When there is no error the new word `CELL+`
   will be included in the dictionary so that it can be found and used.

Postincrement)+ Predecrement -) Zero offset)

```
code DROP ( x -- )
    tos  sp )+ ldr,     \ tos ← sp)   sp ← sp+4
    next,
end-code

code DUP ( x -- x x )
    tos  sp -) str,     \ sp ← sp-4   sp) ← tos
    next,
end-code

code SWAP ( x y -- y x )
    day  sp ) ldr,      \ day ← sp)
    tos  sp ) str,      \ sp) ← tos
    tos day movs,  next,
end-code
```

Control flow

The words `if,` `until,` and `while,` preceded by a condition
(`=?` `cs?` `neg?` `vs?` `u>?` `<?` `>?`) assemble a conditional branch.
The word `no` after a condition inverts the condition.
These nestable structures are present in noForth:

```
condition if,  ..  then,
condition if,  ..  else,  ..  then,
begin,  ..  condition while,  ..  repeat,
begin,  ..  condition until,
ahead,  ..  then,
begin,  ..  again,
```

This means you no longer have to bother with processor branch instructions
and their labels. Examples:

```
code MIN ( x y -- z )
    day  sp )+ ldr,     \ day ← sp)   sp ← sp+4
    tos day cmp,
    >? if,              \ tos greater than day?
        tos day mov,    \ tos ← day
    then,  next,
end-code
```

```
code FILL ( adr len ch -- )
    day  sp ) ldr,          \ day ← len
    sun  sp 4 #) ldr,       \ sun ← adr
    sp 8 # adds,            \ sp ← sp+8
    begin,
        day 0 # cmp,
    =? no while,            \ day is not zero?
        tos  sun ) strb,    \ sun) ← ch
        day 1 # subs,       \ -day ← day-1
        sun 1 # adds,       \ sun ← sun+1
    repeat,
    tos sp )+ ldr,  next,  \ tos ← sp)   sp ← sp+4
end-code
```

Literal pool

The Thumb2 immediate literal # cannot handle large immediate numbers.
That's why we create a literal pool.

```
code set-msb ( x1 -- x2 )
      hx 80000000 ,       \ a very small literal pool
code> day w ) ldr,
      tos day orrs,  next,
end-code
```

When noForth starts execution of a word defined with `code` the body address (the address where the assembly code normally starts) can be found in register `w` . `code>` changes the starting point to just after `code>`, but does not change the address in `w` . This means that with `, h,` or `c,` you may store anything between `code` and `code>`. The execution will start at `code>` and the pool address will still be in `w`. It is up to you to decide how to access the data through `w` .

A subroutine or an interrupt is not entered via the inner interpreter. In that case we need the word `data>` to mark the beginning of a literal pool. It assembles a jump over the pool to just after `code>` with the pool address in register `w`.
Between `data>` and `code>` a literal pool can be inserted at any place in assembly code.

```
routine PIN-IRQ    ( -- )       \ All numbers are in hex
    { w day sun lr } push,
    data>   2 bitmask 200 * ,   \ GPIO2 clear mask
            400140F0 ,          \ Raw interrupt 0 address
            19 bitmask ,        \ GPIO25 bit masker
            D000001C ,          \ Output GPIO_XOR address
    code>
    w { day sun } ldm,  day sun ) str,  \ Reset GPIO2
    w { day sun } ldm,  day sun ) str,  \ Toggle GPIO25
    { w day sun pc } pop,
end-code
```

Overview of the ARM Thumb2 instructions in noForth t

Only Lo-registers can be used unless explicitly stated otherwise.
Instructions fit into 16 bits. Exceptions are marked with "(32)".

Move

	noforth	traditional	cycles	
rd ← rm	rd rm movs,	MOVS Rd,Rm	1	
rd ← i	rd i # movs,	MOVS Rd,#i	1	8-bit i=0,1..FF
rd ← rm	rd rm mov,	MOV Rd,Rm	1	any reg.
pc ← rm	pc rm mov,	MOV PC,Rm	2	any reg. rm

Load

- offset in i:

word	rd rn i #) ldr,	LDR Rd,[Rn,#i]	2/1*	5-bit i=0,4..7C
halfword	rd rn i #) ldrh,	LDRH Rd,[Rn,#i]	2/1*	5-bit i=0,2..3E
byte	rd rn i #) ldrb,	LDRB Rd,[Rn,#i]	2/1*	5-bit i=0,1..1F
rp-relative	rd rp i #) ldr,	LDR Rd,[SP,#i]	2/1*	5-bit i=0,4..7C
pc-relative	rd pc i #) ldr,	LDR Rd,[Rn,#i]	2/1*	8-bit i=0,4..3FC

- offset in rm:

word	rd rn rm r) ldr,	LDR Rd,[Rn,Rm]	2/1*
halfword	rd rn rm r) ldrh,	LDRH Rd,[Rn,Rm]	2/1*
halfw. signed	rd rn rm r) ldrsh,	LDRSH Rd,[Rn,Rm]	2/1*
byte	rd rn rm r) ldrb,	LDRB Rd,[Rn,Rm]	2/1*
byte signed	rd rn rm r) ldrsb,	LDRSB Rd,[Rn,Rm]	2/1*

2/1 -- 2 if to AHB interface or SCS, 1 if to single-cycle I/O port.*

Store

- offset in i:

word	rd rn i #) str,	STR Rd,[Rn,#i]	2/1*	5-bit i=0,4..7C
halfword	rd rn i #) strh,	STRH Rd,[Rn,#i]	2/1*	5-bit i=0,2..3E
byte	rd rn i #) strb,	STRB Rd,[Rn,#i]	2/1*	5-bit i=0,1..1F
rp-relative,	rd rp i #) str,	STR Rd,[SP,#i]	2/1*	5-bit i=0,4..7C

- offset in rm:

word	rd rn rm r) str,	STR Rd,[Rn,Rm]	2/1*
halfword	rd rn rm r) strh,	STRH Rd,[Rn,Rm]	2/1*
byte	rd rn rm r) strb,	STRB Rd,[Rn,Rm]	2/1*

2/1 -- 2 if to AHB interface or SCS, 1 if to single-cycle I/O port.*

Load and Store multiple registers

excl. base	rn { list } ldm,	LDM Rn!,{list}	1+n*
	rn { list } stm,	STM Rn!,{list}	1+n*
	{ list } pop,	POP {list}	1+n*
with return	{ list pc } pop,	POP {list,PC}	3+n*
	{ list } push,	PUSH {list}	1+n*
with link reg.	{ list lr } push,	PUSH {list,LR}	1+n*

n -- n is the number of elements in the list including PC or LR.*

Add

	noforth	traditional	cycles	
rd ← rn+rm	rd rn rm adds.mv,	ADDS Rd,Rn,Rm	1	
rd ← rd+rm	rd rm adds,	ADDS Rd,Rd,Rm	1	
rd ← rn+i	rd rn i # adds.mv,	ADDS Rd,Rn,#i	1	3-bit i=0,1..7
rd ← rd+i	rd i # adds,	ADDS Rd,Rd,#i	1	8-bit i=0,1..FF
rp ← rp+i	rp i # add,	ADD SP,SP,#i	1	7-bit i=0,4..1FC
rd ← rd+rm	rd rm add,	ADD Rd,Rd,Rm	1	any reg. rm
any reg. to pc	pc rm add,	ADD PC,PC,Rm	2	
with carry	rd rm adcs,	ADCS Rd,Rd,Rm	1	
rd ← rp+i	rd rp i # add,	ADD Rd,SP,#i	1	8-bit i=0,4..3FC
rd ← pc+i	rd pc i # add,	ADR Rd,label	1	8-bit i=0,4..3FC

Subtract

rd ← rn-rm	rd rn rm subs.mv,	SUBS Rd,Rn,Rm	1	
rd ← rd-rm	rd rm subs,	SUBS Rd,Rd,Rm	1	
rd ← rn-i	rd rn i # subs.mv,	SUBS Rd,Rn,#i	1	3-bit i=0,1..7
rd ← rd-i	rd i # subs,	SUBS Rd,Rd,#i	1	8-bit i=0,1..FF
with carry	rd rm sbcs,	SBCS Rd,Rd,Rm	1	
rp ← rp-i	rp i # sub,	SUB SP,SP,#i	1	7-bit i=0,4..1FC
rd ← 0-rn	rd rn neg,	RSBS Rd,Rn,#0	1	

Multiply

rd ← rd*rm	rd rm muls,	MULS Rd,Rm,Rd	1	

Compare

	rn i # cmp,	CMP Rn,#i	1	8-bit i=0,1..FF
any reg.	rn rm cmp,	CMP Rn,Rm	1	
negative	rn rm cmn,	CMN Rn,Rm	1	
AND test	rn rm tst,	TST Rn,Rm	1	

Logical (bitwise)

rd ← rd AND rm	rd rm ands,	ANDS Rd,Rd,Rm	1	
exclusive OR	rd rm eors,	EORS Rd,Rd,Rm	1	
OR	rd rm orrs,	ORRS Rd,Rd,Rm	1	
bit clear	rd rm bics,	BICS Rd,Rd,Rm	1	
move NOT	rd rm mvns,	MVNS Rd,Rm	1	

Shift and Rotate

- number of shifts in i:

logical l.shift	rd rm i # lsls.mv,	LSLS Rd,Rm,#i	1	5-bit i=0,1..1F
logical r.shift	rd rm i # lsrs.mv,	LSRS Rd,Rm,#i	1	
arithm. r.shift	rd rm i # asrs.mv,	ASRS Rd,Rm,#i	1	
logical l.shift	rd i # lsls,	LSLS Rd,Rm,#i	1	5-bit i=0,1..1F
logical r.shift	rd i # lsrs,	LSRS Rd,Rm,#i	1	
arithm. r.shift	rd i # asrs,	ASRS Rd,Rm,#i	1	

- number of shifts in rs:

logical l.shift	rd rs lsls,	LSLS Rd,Rd,Rs	1	
logical r.shift	rd rs lsrs,	LSRS Rd,Rd,Rs	1	
arithm. r.shift	rd rs asrs,	ASRS Rd,Rd,Rs	1	
rotate right	rd rs rors,	RORS Rd,Rd,Rs	1	

Branch

	noforth	traditional	cycles
conditional	=? if,etc.	Bcc label	1/2*
jump	ahead,etc.	B label	2
with link	addr bl,	BL label	3 (32)
with exchange	rm bx,	BX Rm	2
with link and exchange	rm blx,	BLX Rm	2

1/2 -- 2 if taken, 1 if not-taken.*

Extend

signed halfword to word	rd rm sxth,	SXTH Rd,Rm	1
signed byte to word	rd rm sxtb,	SXTB Rd,Rm	1
unsigned halfword	rd rm uxth,	UXTH Rd,Rm	1
unsigned byte	rd rn uxtb,	UXTB Rd,Rm	1

Reverse

reverse bytes in word	rd rm rev,	REV Rd,Rm	1
• byteswap in:			
both halfwords	rd rm rev16,	REV16 Rd,Rm	1
signed bottom half word	rd rm revsh,	REVSH Rd,Rm	1

State

change supervisor call	i # svc,	SVC #i	*
disable interrupts	cpsid,	CPSID	1
enable interrupts	cpsie,	CPSIE	1
read special register	rd spec mrs,	MRS Rd,spec	3 (32)
write special register	spec Rn msr,	MSR spec,Rn	3 (32)
breakpoint	i # bkpt,	BKPT,#i	*

** -- Cycle count depends on processor and debug configuration.*

Hint

send-event	sev,	SEV	1
wait for event	wfe,	WFE	2*
wait for interrupt	wfi,	WFI	2*
yield	yield,	YIELD	1*
no operation	nop,	NOP	1?

1? -- NOP is not reliable for timing issues.
** -- Excludes time spent waiting for an interrupt or event.*

Barriers

instruction sync.	isb,	ISB	3 (32)
data memory	dmb,	DMB	3 (32)
data synchronization	dsb,	DSB	3 (32)

See the ARMv6-M Architecture Reference Manual for more information about the ARMv6-M Thumb instructions.