ARM Assembly Operations

Simplest Complete Program

Compile with `gcc -o filename filename.s` and then run with `./filename`

```
.global main
main:
    mov r7, #1          @exit system call
    svc #0
```

Basic operations

The right column gives the command followed by its arguments. Argument `dr` is the register in which to store the result. Operands `or` must be a register (e.g. `r1`). Operands `oi` can be a register or an immediate (e.g. `#5`). The argument `#0` for `svc` must be this value.

Add	`add dr, or, oi`
Subtract (`or` − `oi`)	`sub dr, or, oi`
Reverse subtract (`oi` − `or`)	`rsb dr, or, oi`
Multiply (`dr` and `or1` cannot be the same)	`mul dr, or1, or2`
Divide signed numbers[1](`or1 / or2`)	`sdiv dr, or1, or2`
Divide unsigned numbers[1] (`or1 / or2`)	`udiv dr, or1, or2`
Copy (from `oi` to `dr`)	`mov dr, oi`
Compare `or` to `oi` and set comparison flags	`cmp or, oi`
Branch to `label`	`b =label`
Branch and link	`bl =label`
Return	`ret`
System call (see table below)	`svc #0`

`svc #0` is controlled by the contents of register `r7`:

1	Exit program
3	Read string (`r2` bytes long) and store using address in `r1`. `r0` must be #0 (standard input)
4	Print string (`r2` bytes long) whose address is stored in `r1`. `r0` must be #1 (standard output)

Conditional Suffixes

All instructions can be used conditionally (based on the last call to `cmp`) by adding one of these suffixes.

If flags are set to "equal"	`eq`
If flags are set to "not equal"	`ne`
If flags are set to "greater than or equal"	`ge`
If flags are set to "less than or equal"	`le`
If flags are set to "greater than"	`gt`
If flags are set to "less than"	`lt`
Always execute (suffix has no effect)	`al`
Never execute (creates a nop)	`nv`

Bitwise Instructions

Bitwise and	`and dr, or, oi`
Bitwise or	`orr dr, or, oi`
Bitwise exclusive or	`eor dr, or, oi`
Bit clear	`bic dr, or, oi`
Logical shift left (`oi` must be immediate)	`lsl dr, or, oi`
Logical shift right (`oi` must be immediate)	`lsr dr, or, oi`
Arithmetic shift right (`oi` must be immediate)	`asr dr, or, oi`
Rotate Right (`oi` must be immediate)	`ror dr, or, oi`

[1] Requires the compile flag `-mcpu=cortex-a7` for `gcc`; see `https://forums.raspberrypi.com/viewtopic.php?t=320122`

Memory instructions

Switch to the text segment	`.text`
Switch to the data segment	`.data`
Enter Thumb mode	`.thumb`
Enter ARM mode	`.arm`
Store `str` as a null-terminated string	`.asciz "str"`
Reserve `oi` bytes of space (`oi` must be immediate)	`.space oi`
Create word (`or` can be a string)	`.word or`
Load word from *`address`*	`ldr dr,` *`address`*
Load address of *`labelText`*	`ldr dr, =#`*`labelText`*
Store word at *`address`*	`str or,` *`address`*
Load byte from *`address`*	`ldrb dr,` *`address`*
Store byte at *`address`*	`strb or,` *`address`*
Push register values to the stack	`push` {*`reglist`*}
Pop register values from the stack	`pop` {*`reglist`*}

`address` can have any of the following forms:

address is stored in the register	`[or]`
address is the sum of register and the value of `oi`	`[or, oi]`
address is stored in the register. Increment after loading/storing	`[or]!`
finds address of the label	`=`*`labelText`*

`reglist` can have any of the following forms:

single register	`or`
range of registers	`or1-or2`
list of registers or ranges of registers	`or1, or2-or3, or4`

NEON Instructions

Using NEON SIMD requires working with a different set of registers. The registers `d0`–`d31` are double-word registers that store 64 bits each. The registers `q0`–`q16` are quad-word registers that store 128 bits each. Note that the `d` and `q` registers are different names for the same data storage; `d0` and `d1` are the two halves of `q0` and so on. Loading and storing into these registers requires special instructions:

Load into NEON	`vld1.`*`suf`* {reglist}, *`address`*
Store from NEON into *`address`*	`vst1.`*`suf`* {*`reglist`*}, *`address`*

where *`suf`* is a suffix that is the number of bytes to be loaded or stored and *`reglist`* is a list of registers or ranges of registers (using `d` and `q` registers).

Once you have your data in the NEON registers, you can use standard arithmetic and logic instructions on it with a few alterations: Add the prefix `v` to the instruction, use either `d` or `q` registers as the parameters, and add a suffix to the end. These suffixes give the size of each element (in bits) within the `d` or `q` register: `.8`, `.16`, `.32`, or `.64`. Optionally, you can also add `u` or `i` to specify that each data element is unsigned or signed. For example,

```
vadd.u8 d0, d1, d2
```

adds the four 8-bit unsigned integers stored in `d1` and `d2`, storing the result in `d0`.

To use NEON instructions, you must add the flag `-mfpu=NEON` to the compilation command.