ARM Assembly Reference (using FASMARM)

Computer Systems (COS10004) – Assignment 2

Official ARM Documentation: http://www.keil.com/support/man/docs/armasm/
FASM Documentation: https://flatassembler.net/docs.php
FASMARM Documentation and changes from FASM: https://arm.flatassembler.net/ReadMe.txt

Name: Zac McDonald
Student ID: 102580465

Registers

When using assembly, most instructions operate on values stored within the following 32-bit registers. In 32-bit ARM there are 16 registers.

Register	Purpose	Shortcut
r0-r6	General purpose	-
r7	Syscall number	-
r8-r10	General purpose	-
r11	Frame pointer	fp
r12	Intra procedural	ip
r13	Stack pointer	sp
r14	Link register	lr
r15	Program counter	pc

Registers 0 to 10 are for general purpose and are the registers you will use the most. Register 7 has an additional use as the “Syscall” number register. That is that when calling system commands (i.e. console IO within an OS), r7 determines the operation.

Registers 11 to 15 have more specific uses. Most important are the stack pointer, link register and program counter. The stack pointer points to the top of the software stack and can be changed to allocate more memory to the stack. The link register is set using the bl instruction and stores the address of the next instruction. It is used for returning from functions. The program counter stores the address of the current instruction.

Some instructions have limitations on the usage of pc and lr, check official documentation for more information.

Labels, Constants and Comments

If we want to store the address of an instruction, we can use labels. They allow us to easily move the program counter to a predetermined location and can be used to define functions or conditional paths. Labels can contain any characters followed by a colon; however they must be unique within the program.

```
label_name:
mov r1, #0
```

Constants and comments work in assembly as they do in most other languages. Constants must have unique names and can define numbers in both hexadecimal and base10. Comments are proceeded by a semicolon.

```
; The syntax for a comment
CONSTANT_NUMBER = 16
CONSTANT_HEX = $10
```

Your code can be split among multiple files, and then all included into one at compile time. Be aware that inclusion is essentially a direct copy/paste.

main.asm

```
; Some functions

include “other.asm”
```

Stack Operations

ARM provides a stack for storing and recalling 32-bit values from RAM using the PUSH and POP instructions. Note that the assembler will automatically reverse the POP command parameters, this allows for easier readability in code. Both commands take a list of registers that can include individual registers or register ranges.

```
PUSH { r0, r4-r12 }
; Do something
POP { r0, r4-r12 }
```

Branching

To move around the program, we can use branching. Branching lets us move the program counter to a provided label address. This allows us to create loops, control flows and functions.

The base branch instruction (b) simply moves the program counter.

```
infinite_loop:
    ; Do stuff
b infinite_loop
```

The branch and link instruction (bl) moves the program counter and sets the link register to the next instruction address. This can be used for returning to the next instruction after a function call.

```
bl some_function
; Do more stuff

some_function:
    ; Do function stuff
b lr
```

The final branch form is branch and exchange (bx). It can also be used as branch, link and exchange (blx) to include linking behaviour. This exchange refers to switching between the ARM and THUMB instruction set (discussed next). It allows us to branch to a location and switch to its instruction set. If you use the THUMB instruction set, you will *need* to return with bx or blx to switch back to ARM.

THUMB

So far, we've only covered the ARM instruction set. This is the set of instructions you will be using most often as it has the most versatility. The ARM instruction set stores 32-bit instructions and works with all the 32-bit instructions, while THUMB is stored as 16-bit instructions and has many limitations (see official documentation). The THUMB instruction set allows us to store commands more densely and execute them faster.
Remember, you must use bx or blx to branch and switch to the destination instruction set.

Using FASMARM, choosing an instruction set works as follows:

```
CODE32      ; Switch to ARM
; Some code

THUMB       ; Switch to THUMB
; Some code
```

Functions

There are many methods of writing functions in assembly. This is my suggested function syntax to aid code readability and reusability.

I prefer to immediately use the stack to store the link register, and to return by popping this stored value directly into the program counter. This allows us to seamlessly call additional functions or recursively call functions without worrying about losing the original link register location. Using the stack also allows us to easily borrow and return registers that the ABI states we mustn't change.

```
Some_Function:
; Description of function
; returns r0 = result
; params r0 = ...
; params r1 = ...
PUSH { lr, r4-r5 }
    ; Some code
POP { pc, r4-r5 }
```

Note that we have pushed and popped the values of r4 and r5, this is so that we can have additional registers to use without interfering with their original values (see ABI).

If using different instruction sets, this could be modified slightly while retaining the advantages of using the stack.

```
THUMB
Some_Function:
; Description of function
; returns r0 = result
; params r0 = ...
; params r1 = ...
PUSH { lr, r4-r5 }
    ; Some code
POP { lr, r4-r5 }
bx lr
```

Note that your functions should follow the ABI standard (discussed next) for them to interact with other software.

Application Binary Interface

The ABI standard defines how we should access data structures and functions from machine code.

The ARM ABI standard can be found here: https://developer.arm.com/architectures/system-architectures/software-standards/abi

Most importantly, relating to register usage and functions, the ABI states that registers 0 to 3 should be used for passing arguments and registers 0 and 1 for return values. It also states that registers 4 to 10 should not have their values changed by functions.

The ABI provides many more standards beyond this, relating to the stack and use of the named registers. See the official documentation for these as I only wanted to highlight the standard for the general purpose registers.

Storing and Recalling

Instructions related to the moving, storing and loading of data.

Move:

```
mov r1, #4
mov r1, r2
mov r1, label_name
```

The mov command lets us move a value into a register. The first operand is the destination register and the second is a flexible operand.
We can mov constants such as a number or label address as well as the value stored in another register.
Note that constants have a maximum range of 0-65535 (16-bits).

Load Register:

```
ldr r1, [r0]
ldr r1, [r0, #1] ; r0 + 1 byte
```

The ldr command allows us to load a value from some address. Note the square bracket syntax which denotes a pointer, i.e. [r0] is the value at the memory location, r0. Inside this pointer, we can also provide a *byte* offset.

Store Register:

```
str r1, [r0]
str r1, [r0, #1] ; r0 + 1 byte
```

The str command allows us to store a value from a register into a given memory location (provided by a pointer with optional byte offset).

Load/Store Register options and limits:

```
; Load a byte (8-bits)
ldrb r1, [r0]
```

```
; Load a half (16-bits)
ldrh r1, [r0]

; Load a double (64-bits)
ldrd r0, r1, [r2]
```

The str and ldr commands have additional settings that we can use for loading data of varying size (they default to 32-bits). All options work for both commands.
Note that for the double store/load the registers (not pointer) must be consecutive registers with the first being even.

ldr and str have the additional limitation of only being able to operate on 4-byte intervals. As they use 32-bits at a time, the pointer address (not including offset) **must be on a 4-byte boundary.** If it is not the program will crash.

To ensure that our data is at this interval, we can use the following:

Align:

```
align 4
some_data:
; data
```

The align compiler instruction ensures that the next instruction is on the given byte interval.
I am unsure if it is a 2^n or n interval, however using "align 4" guarantees that the data will be on a 4-byte interval in either case.
In the compiled code, the skipped lines are filled with dummy commands such as "mov r1, r1" to provide padding.
In addition to ensuring addresses are usable by the ldr command, align 4 also allows a direct mov of a given label.

Address:

```
adr r0, some_label
```

The adr command allows us to store the address of a label into a register directly without dealing with limitations of mov and ldr. It has its own limitation however, of only being able to use registers within a set range from the instruction. If the address is outside this range, a compile time error is thrown, align and use mov instead.

Logical Operations

Common logical, bit manipulation instructions.

OR:

```
orr r1, #32
orr r1, r2
orr r2, r1, #32
```

The orr command (with two parameters) performs a bitwise OR operation between the operands and stores the value in the first operand. With three parameters, the OR operation is on the second and third operands with result stored in the first.

This can be used alongside FASM compile time commands to mov constants that exceed the mov commands 16-bit limit.

```
NUM = $3F200000
mov r0, NUM and $FF
orr r0, NUM and $FF00
```

AND:

```
and r1, #32
and r1, r2
and r2, r1, #32
```

The and command has the same two and three parameter behaviour as orr, except obviously performing a bitwise AND operation.

XOR:

```
eor r1, #32
eor r1, r2
eor r2, r1, #32
```

The eor command is the same, performing a bitwise Exclusive-OR (XOR) operation.

NOT:

```
mvn r1, #32
mvn r1, r1
```

The mvn (move NOT) command performs a bitwise NOT operation on the second

operand and stores the result in the first operand.

Logical Shift Left:

```
lsl r1, #2
lsl r1, r2
```

The lsl command shifts a registers binary value left (padding with zeros) a given number of times. The result is stored in the first operand.

Logical Shift Right:

```
lsr r1, #2
lsr r1, r2
```

The lsr command works the same as lsl, except shifting right (padding with zeros).

Arithmetic Operations

Common arithmetic instructions.

Add:

```
add r1, #2
add r1, r2
add r3, r1, r2
```

The add command adds two values together and stores the result in the first operand. Like the logical commands, with three registers the operation is between the second and third operands with the result stored in the first operand.

Subtract:

```
sub r1, #2
sub r1, r2
sub r3, r1, r2
```

The sub command works the same as add except performing subtraction. It subtracts the second operand from the first.

Add/Subtract 64-bit values:

```
adds r4, r0, r2
adc  r5, r1, r3
```

To add or subtract 64-bit values, we need to use the "add with carry" (adc) or "subtract with carry" (sbc) command in conjunction with the associated add/sub command with an 's' suffix.

Firstly, in this example we assume a 64-bit value is stored in r0 (lower), r1 (upper) and another in r2, r3. We then want to add them and store the result into r4, r5. The 's' suffix tells the add/sub command to set condition flags (discussed later). In this case we care about the carry-flag. If the addition would cause a carry over the first 32-bit boundary, the carry-flag would be set. The adc/sbc command is then used to add/subtract while respecting the set carry-flag.

Multiply:

```
mul r1, #2
mul r1, r2
mul r3, r1, r2
```

The mul command works the same as add except performing multiplication.

Multiply and Add/Subtract:

```
mla r0, r1, r2, r3
mls r0, r1, r2, r3
```

The mla and mls commands take 4 parameters and perform a conjunction of multiplication and addition/subtraction. The second and third operands are multiplied, then the fourth operand is added (mla) or subtracted (mls) from their product. The result is stored in the first operand.
Note all parameters *must* be registers.

Comparisons

Comparison operators and conditional execution.

Compare (subtract):

```
cmp r1, #2
cmp r1, r2
```

The cmp command subtracts the second operand from the first and sets conditional flags. It is functionally equivalent to 'subs' except the result is not stored.

Compare (add):

```
cmn r1, #2
cmn r1, r2
```

The cmn command is the same as cmp except performing addition, making it equivalent to 'adds' without storing the result.

Test Equivalence:

```
teq r1, #2
teq r1, r2
```

The teq command checks if two values are equal and sets the conditional flags. It is functionally equivalent to 'eors' without storing the result. This means that the equal (eq) suffix is used to detect a success.

Test Bits:

```
tst r1, #2
tst r1, r2
```

The tst command checks if the bits in the second operand are set in the first, updating conditional flags. It is functionally equivalent to 'ands' without storing the result.
This means that provided the second operand only contains 1 high bit, we check if that bit is set in operand one with the not-equal (ne) suffix.

```
; Check if the 2nd bit is set
tst r0, #2
bne success
```

Conditional Suffixes:
To create conditional logic in ARM assembly, we can use the instructions that set the conditional flags. Once these flags are set, we can add one of the following suffixes to any command, causing that command to only execute if that condition is met. Some of the common suffixes are shown below, more can be found in the official documentation.

Suffix	Meaning
eq	Equal
ne	Not equal
mi	Negative
pl	Positive or zero
hi	Unsigned higher
ls	Unsigned lower or same
ge	Signed greater than or equal
gt	Signed greater than
le	Signed less than or equal
lt	Signed less than

As an example, the following will wrap the value in r0 between 0 and 5 using essentially an 'if-elseif', this could be used for looping through an array.

```
cmp r0, #0
moveq r0, #5
b end_if
cmp r0, #5
moveq r0, #0
end_if:
```

Data Structures

Storing and accessing data in memory.

Data Types:

```
db 20  ; 8-bit byte
dh $0E2F  ; 16-bit half word
dw $3F200000  ; 32-bit word
```

FASMARM allows us to define data of several types (see its documentation). Shown are how to define bytes, half-words and words.

Note that these should be used in conjunction with a label and can be accessed using ldr when aligned.

```
mov r0, number_of_things
ldrb r1, [r0]

align 4
number_of_things:
db 20
```

Arrays:
Like in other languages, arrays can be used to store multiple values of the same type. They are accessed in the same way as the last example, except utilising an offset.

```
align 4
byte_array:
db 18, 17, 27, 22
```

```
mov r0, byte_array
mov r1, #1 ; index to load
ldrb r1, [r0, r1]
```

Note that for accessing elements, the index is in bytes, meaning for half-words the index should be doubled (2 bytes per element) and for words the index should be multiplied by 4. Also remember the ldr/str size settings: b and h suffixes.

Structs:
Structs are used similar to arrays, however we may store different types of data. The ldr/str 4-byte boundary limitation will be a problem when loading mixed types. For this reason I'd recommend **not** to use mixed types unless using intermediary labels and ensuring alignment for both half-words and words.

```
align 4
some_struct:
dw $FFEAABFF
dh $0E2F
db $FF
db #30
db #1
```