ENSC 254 – Lab 2

May 27, 2021

1 Introduction

We continue using the same code that we arrived at at the end of the last lab. In that lab we determined by hand the machine code to implement a byte subtraction algorithm on the ARM architecture, as well as the modified machine code after replacing a *mov* instruction and a *mvn* instruction – each using immediate data – with *ldrb* instructions where the base register was ultimately the PC (the Program Counter). When the base register is the PC, the addressing mode is sometimes called pc-relative addressing (https://en.wikipedia.org/wiki/Addressing_mode#PC-relative_2)

```
wl       equ   4   ;@ bytes (Word Length is 4)
;@ start at address 0
         mov   r2, #1
         ldrb  r0, [pc,#____] ;@ Load from 0x21 into r0
         ldrb  r1, [pc,#____] ;@ Load from 0x20 into r1
         mvn   r1, r1         ;@ take 1's comp
         add   r1, r1, r2     ;@ add 1 for 2's comp
         add   r0, r0, r1
         strb  r0, [pc,#3]    ;@ Store r1 at address 0x23
         mov   pc, #2*wl      ;@ take us back to 'ldrb r1, …'
```

2 The pseudo-direct addressing mode

In the previous lab and a recent lecture, we discussed how loads and stores are handled on the ARM architecture. A value is loaded from memory at an address provided by the result of a register, which is the base memory address, to which is added or from which is subtracted an offset, which can be an immediate value (possibly zero) or derived from a register.

Since we are using the PC as the base register and using immediate offsets, we can use some conventions to make our symbolic program more readable by writing it as:

```
wl        equ   4   ;@ bytes (Word Length is 4)
;@ start at address 0
          mov   r2, #1
          ldrb  r0, in2        ;@ Load from 0x21 into r0
Loop:     ldrb  r1, in1        ;@ Loop is a label for address 8
          mvn   r1, r1         ;@ take 1's comp
          add   r1, r1, r2     ;@ add 1 for 2's comp
          add   r0, r0, r1
          strb  r0, [pc,#3]    ;@ Store r1 at address 0x23
          mov   pc, #2*wl      ;@ take us back to 'ldrb r1, …'

in1:      dcb 3
in2:      dcb -1

          space 1              ;@ bring us to the needed address
output:   space 1              ;@ was at address 0x23
          end
```

"Loop", "in1", "in2", and "output" are called labels and are just names for memory locations relative to the beginning of the program. So *Loop* has a value of 8, and in fact for our program on our simulator, *Loop* ends up exactly at memory location 8. "space" is a directive that indicates that the specified number of bytes should be reserved in memory, often for variables. "dcb" (define constant byte) is a directive that indicates that the supplied value should be put in a byte in memory. Just like instructions go one word after another in memory, these *dcb* directives put bytes in memory one byte after another.

The load and store instructions now use a pseudo addressing mode, similar to the generally known Absolute/Direct addressing mode (see https://en.wikipedia.org/wiki/Addressing_mode#Absolute/direct). Note that for load and store instructions, an ARM processor always uses some form of "Base plus offset" addressing (see https://en.wikipedia.org/wiki/Addressing_mode#Base_plus_offset.2C_and_variations) so that is why we call this a pseudo addressing mode. It looks like Absolute/Direct addressing in the symbolic program, and a conversion is required to generate for the machine code an offset from the PC. I call this pseudo addressing mode the pseudo-direct addressing mode.

3 Other ways of getting data into a register

Using pc-relative addressing, one can load from or store to a location in the address space around the program counter. If this portion of the address space is in RAM, which is how we have configured our simulator for our case, then the location can hold a variable if that is desired. Oftentimes, variables end up far away from the instructions, and therefore the program counter, and so often pc-relative addressing is not possible for variables on ARM-based systems.

Sometimes a program needs simple constant data. For example, our program loads the value 1 into r2. We could replace the instruction, which uses immediate data and occupies a total of one word,

```
            mov   r2, #1
```

with the instruction

```
            ldrb  r2, in3
```

and, near the bottom of the code, the following additional assembler directive.

```
in3:     dcb      1
```

So this is an example of moving a constant from immediate data to a byte in memory. We might want to put all our constants together in memory. We are switching from a *mov* instruction with immediate data to an *ldr* instruction with pseudo-direct addressing and, as we now know, that will need to be translated to a real *ldr* instruction with the ARM "base plus offset" addressing.

The *mov* instruction with immediate data works fine for loading a constant byte into a register, but *ldrb* has a sister instruction, *ldr*, that loads a word (occupying 4 adjacent bytes) into a register and here we can have problems if we try to use the *mov* instruction instead of *ldr*. Not every 32-bit constant can be generated with the immediate data in a *mov* instruction. For example, the value 0x129 cannot be generated with the immediate data in a *mov* instruction. In order to efficiently load any 32-bit constant into a register, another pseudo addressing mode is available. I call it the pseudo-immediate addressing mode because it looks like immediate addressing. See https://en.wikipedia.org/wiki/Addressing_mode#Immediate.2Fliteral . (Note that in the ARM documentation they usually simply call this scheme a pseudo instruction.) Replacing the above *mov* instruction with an *ldr* instruction with pseudo-immediate addressing would look like:

```
            ldr   r2, =1
```

The instruction on the line above (with particular constants such as, in this case, the small constant 1) is translated to:

```
            mov   r2, #1
```

If, instead, we had the *ldr* instruction in these program fragments

```
;@ start at address 0
            …
            ldr   r4, =0x129
            …
            mov   pc, #2*wl
```

the translated program fragments would look something like:

```
;@ start at address 0
            …
            ldr   r4, literal129    ;@ ldr r4, [pc, #0x???]
            …
            mov   pc, #2*wl
            …
literalPool:
literal129: dcw   0x00000129
```

dcw is a directive that indicates that the supplied 4-byte word should be placed in memory. A literal pool (named *literalPool* in our case) is usually part of what results from translating a set of *ldr* instructions with pseudo-immediate addressing, such as our instruction

```
            ldr   r4, =0x129
```

Automated tools that generate machine code, called assemblers, always use the ARM *ldr* instruction and a literal, and never a *mov* or *mvn* instruction, when the "immediate" data used in pseudo-immediate addressing is a symbolic address, such as our label *literalPool*. This is because the assembler doesn't exactly know the absolute memory location for the label. That won't be decided until another tool called the linker is later invoked to place the assembled code in memory.

Note that our literal pool is placed somewhere beyond our instruction "mov pc, #2*wl", which redirects program execution to continue at an instruction in some other portion of memory (back near the top of the program in our case), and so the content of the literal pool should never be executed by the processor. This is important, as the one or more literals in a literal pool are not intended to be executed as one or more processor instructions.

So, an *ldr* instruction with pseudo-immediate addressing can be thought of, in some situations, as being translated to an *ldr* instruction with pseudo-direct addressing and in addition a literal placed in memory using a *ldw* directive. In turn, the *ldr* instruction with pseudo-direct addressing must then be translated into a real *ldr* instruction with PC-relative ARM "base plus offset" addressing.

Details, from official documentation but with my edits, are shown on the next 2 pages:

Loading with LDR Rd, =const

The `LDR Rd,=const` pseudo-instruction can construct any 32-bit numeric constant in a single instruction. Use this pseudo-instruction to generate constants that are out of range of the `MOV` and `MVN` instructions.

The `LDR` pseudo-instruction generates the most efficient code for a specific constant:

- If the constant can be constructed with a `MOV` or `MVN` instruction, the assembler generates the appropriate instruction.
- If the constant cannot be constructed with a `MOV` or `MVN` instruction, the assembler:
  - places the value in a *literal pool* (a portion of memory embedded in the code to hold constant values)
  - generates an `LDR` instruction with a program-relative address that reads the constant from the literal pool.

  For example:

```
LDR           rn [pc, #offset to constant]
                                  ; load register n with one word
                                  ; from the address [pc, + offset]
```

The pseudo-instruction loads a register with either:

- a 32-bit constant value
- an address.

Note

This section describes the `LDR` *pseudo*-instruction only. Refer to the *ARM Architectural Reference Manual* for information on the `LDR` *instruction*.

Syntax

The syntax of `LDR` is:

```
LDR{condition} register,=[expression | label-expression]
```

where:

condition

is an optional condition code.

register

is the register to be loaded.

expression

evaluates to a numeric constant:

- If the value of *expression* is within range of a MOV or MVN instruction, the assembler generates the appropriate instruction.
- If the value of *expression* is *not* within range of a MOV or MVN instruction, the assembler places the constant in a literal pool and generates a program-relative LDR instruction that reads the constant from the literal pool.
  The offset from the pc to the constant must be less than 4KB. You are responsible for ensuring that there is a literal pool within range.
  See *LTORG directive* for more information [the gnu assembler uses its .LTORG directive].

label-expression

is a program-relative or external expression. The assembler places the value of *label-expression* in a literal pool and generates a program-relative LDR instruction that loads the value from the literal pool.
The offset from the pc to the value in the literal pool must be less than 4KB. You are responsible for ensuring that there is a literal pool within range.
See *LTORG directive* for more information [the gnu assembler uses its .LTORG directive].
If *label-expression* is an external expression, or is not contained in the current area, the assembler places a linker relocation directive in the object file. The linker ensures that the correct address is generated at link time.

Usage

The LDR pseudo-instruction is used for two main purposes:

- to generate literal constants when an *immediate* value cannot be moved into a register because it is out of range of the MOV and MVN instructions.
- to load a program-relative or external address into a register. The address remains valid regardless of where the linker places the AOF area containing the LDR.

Refer to Chapter 5 *Basic Assembly Language Programming* in the *ARM Software Development Toolkit User Guide* for a more detailed explanation of how to use LDR, and for more information on MOV and MVN.

Example

```
LDR             r1,=0xfff                ; loads 0xfff into r1
                                         ;
LDR             r2,=place                ; loads the address
                                         ; place into r2
```

Based on the **ARM Software Development Toolkit User Guide** at
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0040d/Babcfejg.html

4 The ADR pseudo instruction

If a literal pool needs to be involved in loading a constant, it slows the processor down, as a load from memory is needed at the time of execution of the instruction, and such loads are relatively slow. If the value to be loaded into a register is an address in the same "part" of the program, a choice of two pseudo instructions can be used. The simplest is the ADR pseudo instruction, which gets translated into either an ADD or a SUB (subtract) instruction, as described on the next page.

So, for example, the last instruction in the program that loops back to the instruction at address 8 of the program could be written:

```
            adr   pc, Loop
```

Because *Loop* evaluates to a lower-numbered memory address than where the ADR pseudo instruction is located, it is clear that some value should be subtracted from the current PC. In this case, the pseudo-instruction is translated to the following.

```
            sub   pc, pc, #0x??
```

The ADR pseudo instruction always ends up adding or subtracting an immediate value from the PC.

We are leaving it to you, below in this lab, to work out the proper immediate value to use and how to translate this *sub* instruction to machine code. The machine code for this *sub* instruction is different from the machine code for the *add* instructions that we are using. It is different in two ways:

1) the opcode in the instruction is switched from *add* to *sub*
2) we are using immediate data instead of a register for the 2nd operand of the operation (so invert the I bit and supply immediate data instead of indicating a register).

As described on the next page, the ADR pseudo-instruction can only get so far from the current program counter. In order to get further, we could use two *add* or two *sub* instructions, as described in the next section.

ADR ARM pseudo-instruction

The ADR pseudo-instruction loads a program-relative or register-relative address into a register.

Syntax

The syntax of ADR is:

ADR{*condition*} *register*,*expression*

where:

register

is the register to load.

expression

is a program-relative or register-relative expressionthat evaluates to:

- a non word-aligned address within 255 bytes
- a word-aligned address within 1020 bytes.

The address can be either before or after the address of the instruction or the base register.

See *Register-relative and program-relative expressions*.

Usage

ADR always assembles to one instruction. The assembler attempts to produce a single ADD or SUB instruction to load the address. If the address cannot be constructed in a single instruction, an error is generated and the assembly fails.

Use the ADRL pseudo-instruction to assemble a wider range of effective addresses.

If *expression* is program-relative, it must evaluate to an address in the same code area [section for the GNU toolchain] as the ADR pseudo-instruction. Otherwise the address may be out of range after linking.

Example

```
start           MOV             r0,#10
                ADR             r4,start                        ; => SUB r4,pc,#0xc
```

5 The ADRL pseudo-instruction

In our program, we could exchange the instruction

```
        strb    r1, output  ;@ Store r1 at address output
```

with the pseudo-instruction and instruction:

```
        adr     r4, output      ;@ get address of output into r4
        strb    r1, [r4, #0]    ;@ Store r1 at address in r4
```

or

```
        adrl    r4, output      ;@ get the address into r4 using adrl
        strb    r1, [r4, #0]    ;@ Store r1 at address in r4
```

These two sequences first load the address of the variable to store to into r4, and then store the result located in register r1 into memory using r4 as holding the base address and using an offset of 0. If the *adr* or *adrl* pseudo-instruction was moved up in the program, the same r4 base address could potentially be used for the 2 *ldrb* instructions earlier in the program. That would require the earlier offsets to be subtracted. Or r4 could point to constant *in1* and the offset in the code lines above could be positive instead of 0.

As indicated above, whereas ADR is translated into one *add* or *sub* instruction, ADRL is translated into two instructions (usually two *add* or two *sub* instructions), and allows one to calculate an address further away from the PC. Any *add* or *sub* instructions involved in this use immediate data. The calculated address can potentially be much further away from the PC but that requires a more-complicated form of immediate data in at least one of the arithmetic instructions. We will learn about this more-complicated form of immediate data in the class soon. The ADR pseudo instruction is a very fast way to load an address in the same section of the program. The ADRL pseudo instruction typically takes twice as long as the ADR pseudo instruction but can reach much further. For data in different sections of the program, the pseudo-immediate addressing mode can be used with the *ldr* instruction, but that is the least efficient way to load an address compared with these pseudo instructions.

In Section 6.2 of the lab, we ask you to consider what *add* instructions would be needed to get the required address into r4 when translating the ADRL pseudo-instruction if *output* was located at address 0x129.

Details about the ADRL pseudo-instruction from the same official documentation are shown on the next page.

ADRL ARM pseudo-instruction

The ADRL pseudo-instruction loads a program-relative or register-relative address into a register. It is similar to the ADR pseudo-instruction. ADRL can load a wider range of addresses than ADR because it generates two data processing instructions.

Syntax

The syntax of ADRL is:

ADRL{*condition*} *register*,*expression*

where:

register

is the register to load.

expression

is a register-relative or program-relative expression that evaluates to:

- a non word-aligned address within 64KB
- a word-aligned address within 256KB.

The address can be either before or after the address of the instruction or the base register. See *Register-relative and program-relative expressions*.

Usage

ADRL always assembles to two instructions. Even if the address can be reached in a single instruction, a second, redundant instruction is produced.

If the assembler cannot construct the address in two instructions, it generates an error message and the assembly fails. See *LDR ARM pseudo-instruction* for information on loading a wider range of addresses. See also Chapter 5 *Basic Assembly Language Programming* in the *ARM Software Development Toolkit User Guide*.

If *expression* is program-relative, it must evaluate to an address in the same code area [section for the GNU toolchain] as the ADRL pseudo-instruction. Otherwise the address may be out of range after linking.

[…]

Example

```
start           MOV             r0,#10
                ADRL            r4,start + 60000        ; => ADD r4,pc,#0xe800
                                                        ;  ADD r4,r4,#0x254
```

6 ADRL exercise

In a previous section of this document we had arrived at something like the following for our program:

```
wl       equ   4   ;@ bytes (Word Length is 4)
;@ start at address 0
         mov   r2, #1
         ldrb  r0, in2        ;@ Load from 0x21 into r0
Loop:    ldrb  r1, in1        ;@ Loop is a label for address 8
         mvn   r1, r1         ;@ take 1's comp
         add   r1, r1, r2     ;@ add 1 for 2's comp
         add   r0, r0, r1

         strb  r0, [pc,#3]    ;@ Store r1 at address 0x23

         mov   pc, #2*wl      ;@ take us back to 'ldrb r1, …'

in1:     dcb 3
in2:     dcb -1

         space 1              ;@ bring us to the needed address
output:  space 1              ;@ was at address 0x23

         end
```

As described in section 5, let's put the address *output* into *r4* using the ADRL pseudo instruction and use *r4* as the base address for the STRB instruction, giving us the following program:

```
wl       equ   4   ;@ bytes (Word Length is 4)
;@ start at address 0
         mov   r2, #1
         ldrb  r0, in2        ;@ Load from 0x21 into r0
Loop:    ldrb  r1, in1        ;@ Loop is a label for address 8
         mvn   r1, r1         ;@ take 1's comp
         add   r1, r1, r2     ;@ add 1 for 2's comp
         add   r0, r0, r1

         adrl  r4, output     ;@ 6.2 uses a variable output129 at 0x129
         strb  r0, [r4, #0]   ;@ Store r0 at address output

         mov   pc, #2*wl      ;@ take us back to 'ldrb r1, …'

in1:     dcb 3
in2:     dcb -1

         space 1              ;@ bring us to the needed address
output:  space 1              ;@ was at address 0x23

         space 4
;@ Put literal pool here for this lab. - address 0x30

         end
```

6.1 ADRL with an address close to the program counter.

1. If the second arithmetic instruction generated for the ADRL pseudo-instruction uses an immediate operand value of zero (0), what symbolic arithmetic instructions does the ADRL pseudo-instruction on the previous page translate into?

2. Translate this into machine code and test it in the simulator.

6.2 ADRL with an address further away from program counter.

Keep the variable output in memory, but now consider there is another variable called output129 at address 0x129 and load the address of output129 into r4 using the ADRL pseudo-instruction.

1. If the second arithmetic instruction generated for the ADRL pseudo-instruction uses an immediate operand value of 0xFF, what symbolic arithmetic instructions does the ADRL pseudo-instruction translate into?

2. Translate this into machine code and test it in the simulator.

7 Alternatives for instruction: `mov pc, #2*wl`

At the end of the above program we move the value 0 into the program counter to loop to the top of the program. However, there are many other ways of doing this, many involving pseudo addressing modes and pseudo instructions. For each of the following, replace the "`mov pc, #0`" instruction with the instruction or pseudo-instruction indicated. You may have to determine what instructions should actually be inserted, determine relative offsets to data from the program counter, and write any literal pool values (i.e. literals). Put the literal pool at address 0x30. In any case, please write the machine code.

*** For our simulation the label Loop should end up at address 8 ***

7.1 mov pc, r2, lsl #3

1. What value is in r2? "r2, lsl #3" takes r2 and logically shifts it left in a temporary fashion by 3 bit positions. So if r2 is 0b…00000001, then "r2, lsl #3" is 0b…00001000.
2. Translate this into machine code and test it in the simulator. The addressing mode is described in section A5.1.5, "Data-processing operands - Logical shift left by immediate"

on page A5-9 of the Reference Manual.

3. What value ends up in PC?

7.2 adr pc, Loop

1. What symbolic instruction(s) does this translate into?

2. Translate this into machine code and test it in the simulator

3. Given we have the `adr` pseudo instruction, would it ever make sense to use the `adrl` pseudo instruction with PC as the destination? So a pseudo instruction of the form:

```
adrl  pc, <expression>
```

Why or why not?

7.3 ldr pc, =8

1. Translate this using a 'mov' instruction.

2. Translate this into machine code and test it in the simulator.

7.4 ldr pc, =Loop

1. Translate this from pseudo-immediate addressing to an 'ldr' instruction with pseudo-direct addressing. Introduce a literal at label *literal0*

2a. What value gets put into the literal in the literal pool?

2b. What is the complete symbolic '`dcw`' directive to put the value in memory?

2c. Since this is currently the only literal in the literal pool, at what address is the literal located? Recall I requested above that the literal pool be put at address 0x30.

2d. Translate the `ldr` instruction with pseudo-direct addressing into a real `ldr` instruction with PC-relative Base-plus-offset addressing.

3. Translate this into machine code and test it in the simulator.

7.5 b Loop

b is the “Branch” instruction. You can look it up in the Architectural Reference Manual that we used for lab 1. Translate this instruction into machine code. In our case, the 24-bit signed immediate value needs to be negative, and so the most-significant bit of the immediate value will be 1. Sign-extending the 24-bit signed immediate value to 30 bits means taking that most-significant bit and pre-pending it 6 times. Test the machine code in the simulator.