COEN 311
Lab 2: Working with memory variables and linker scripts
Summer 2023

T. Obuchowicz

OBJECTIVES

• To learn the use of a linker script to partition a program and it's data into RAM memory.

• To become acquainted with fundamental ARM assembly language programming (assembler directives,basic addressing modes, the `mul` and `add` instructions).

• To learn how to examine memory contents from within the gdb environment.

• To write a complete ARM assembly language program which computes the vector dot product of two arrays stored in memory.

INTRODUCTION

The program in Lab 1 did not make use of any memory variables. The only data reference by the program in Lab 1 was **immediate data** - data which is contained within the instruction itself. For example, the assembly language instruction:

```
mov r0, #4
```

will move the immediate data 4 (the source operand) into register R0 (the destination operand). The fact that this data is stored within certain bits of the instruction is made evident upon examination of the corresponding line in the assembler listing file:

```
00ec 4FF00400              mov r0, #4
```

The hexadecimal digits indicated in bold font are the immediate data. The ARM assembly language uses 12 of the 32 bits within an instruction to hold immediate data, this imposes some restrictions on the size of the immediate data. The assembler will report as an error when an immediate value is out of range. For example, consider the following instruction:

```
 mov r0, #1452316  @ see what error message the assembler
                   @ reports when an immediate is out of range
```

Assembling this code will result in the error message:

```
big_immediate.s: Assembler messages:
big_immediate.s:19: Error: invalid constant (16291c) after fixup
```

This lab will explore the use of data stored in memory (SRAM) and the methods by which an instruction can access data stored in memory. These methods are termed *addressing modes*.

Consider the following ARM assembly language program:

```
@ Ted Obuchowicz
@ Oct. 12, 2022
@ add_from_mem.s

.syntax unified
.cpu cortex-m4
.thumb

.word 0x20000400
.word 0x800000ed
.space 0xe4

.data
mick:   .byte 0x01  @ reserve 1 byte of RAM and initialize it to 01
keith:  .byte 0x02  @ reserve 1 byte of RAM and initialize it to 02
result: .space 0x01 @ reserve 1 byte of RAM  without any
                    @initialization

.text
start:
       ldr r0, =mick  @ load address of mick into r0
       ldrb r1, [r0]   @ load r1 with memory byte contents of mick
       ldr r0, =keith @ load address of keith into r0
       ldrb r2, [r0]   @ load r2 with memory byte contents of keith
       add r3, r2, r1 @ r3 = r2 + r1
       ldr r0, =result @ load address of result into r0
       strb r3, [r0]   @ store sum into memory at location result
stop:  b stop
```

This program makes use of a `.data` section to define a region of memory which will be used to hold the program's data. The assembler directive **`.byte`** is used to define 1 byte of data. There are other directives to define a halfword (**`.hword`**) consisting of 2 bytes, and a 4 byte word (**`.word`**). The **`.space`** directive is used to reserve the specified amount of memory without initialization to any specified value. Typically, a .space directive is used to reserve memory into which the program will save save value into, so the initial vlaue of this reserved memory is immaterial. The general form of an assembler directive is:

```
label:  directive list of data
```

Multiple data values may be entered one one line separated by commas. For example:

```
list: .byte 0x01, 0x02, 0x03, 0x04, 0x05
```

The prefix `0x` is used to specify a hexadecimal value. Decimal values may be represented without any prefix.

Upon examining the corresponding .lst file, we see that the assembler has defined the following:

```
  13                      .data
  14 0000 01              mick:   .byte 0x01
  15 0001 02              keith:  .byte 0x02
  16 0002 00              result: .space 0x01
```

The numbers 0000, 0001, and 0002 represent offsets from some portion in main memory in which the data will be stored when the program is loaded into the microcontrollers memory with the load command from within the gdb environment. The first data item (0x01) will be stored at offset 0 , the second data item will be stored 1 byte after, etc. The labels `mick`, `keith`, and `result` will be used within the program to refer to these memory locations. Note how the memory byte at location `result` has been set `00`.

Register indirect addressing uses a register (for instance register `r0`) to hold the memory address of some data. The register is said to be a a pointer to the data item. To load another register (`r1`) with the contents of main memory pointed to the by the first register, we would make use of the following instruction:

```
ldrb r1, [r0] @ load r1 with the data pointed to by r0
```

The `[ ]` surrounding a register name is the syntax used by the ARM assembler to denote register indirect addressing. It is said that a picture is worth a thousand words, Figure 1 illustrates the concept of register indirect addressing. Suppose that the label `mick` is associated with memory address 1000, and suppose that somehow register r0 has been loaded with this address.

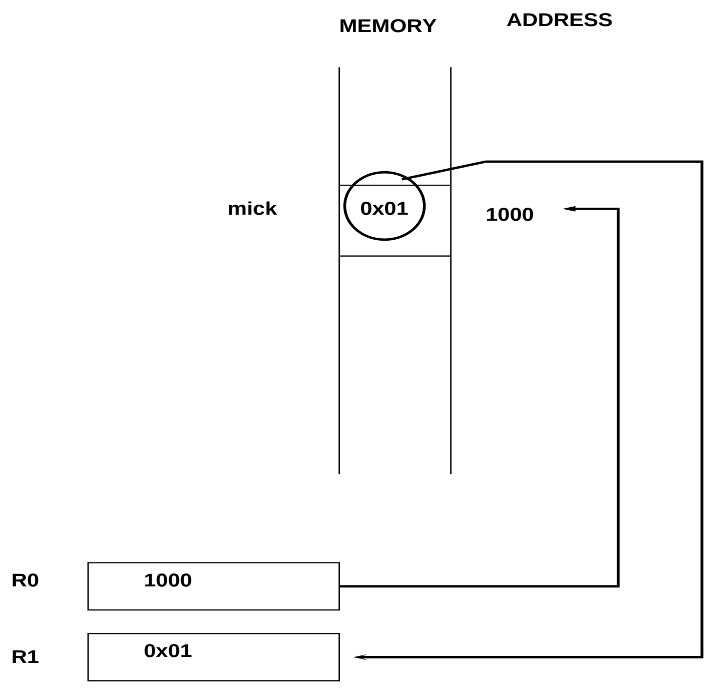

Figure 1: Register indirect addressing.

A register may be loaded with either a byte (8 bits), halfword (16 bits), or a word (32 bits). A type specifier of either b (`byte`), h (`halfword`). In the absence of a type specifier, the default is word. For example,

```
ldrb r1, [r0] @ load r1 with a byte of memory pointed to by r0
ldrh r1, [r0] @ load r1 with two bytes of memory pointed to by r0
ldr r1, [r0]  @ load r1 with  four bytes of memory pointed to by r0
```

In the case of the byte and halfword data transfers, the remaining high order bits of the specified destination register are set to zero.

The question which remains to be answered is how to initialize a particular register with a 32 bit main memory address given the limitations on the size of immediate data ? The ARM assembler uses of a clever technique which involves a *pseudo-instruction* to load a register with the address of some data item (as specified by it's label):

```
ldr r0, =mick  @ load address of mick into r0
```

The general form of this instruction is:
`ldr Rn, =some_32_bit_large_contant_value` if we want to load a register with some large immediate data (which exceeds the limitations of the 12 bit immediate data field) or `ldr Rn, =some_label`.

The assembler will **translate** the pseudoinstruction `ldr r0, =mick` into a variant of register indirect addressing with the program counter (PC) register where an offset value is added to the contents of the PC to obtain the address of memory location where the data (in this example, the 32 bit address corresponding to label mick) is stored. Typically, a portion of main memory at the end of the code is used to hold such data. This region of main memory is called the *literal pool*. [1] Figure 2 illustrates the translation mechanism.

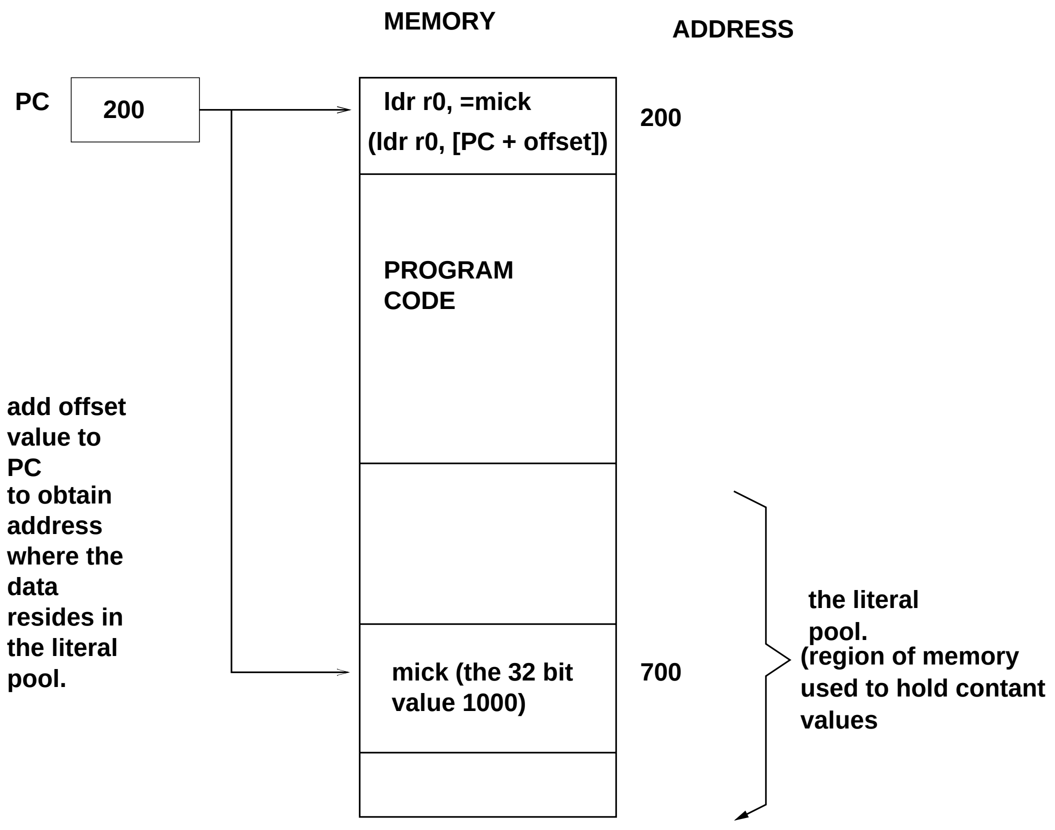
MEMORY
ADDRESS
PC
200
ldr r0, =mick
(ldr r0, [PC + offset])
200
PROGRAM CODE
add offset value to PC to obtain address where the data resides in the literal pool.
mick (the 32 bit value 1000)
700
the literal pool.
(region of memory used to hold contant values

Figure 2: Accessing the literal pool using register indirect with PC and some offset.

The assembler has translated the pseudoinstruction `ldr r0, =mick` into `ldr r0, [PC + offset]`. Suppose this instruction is stored at address 200. Hence the actual offset value is 500. When the instruction is executed, the contents of the program counter register (200) is added to the offset value (500) to yield the address of 700. Found at this address is the contant data corresponding to label `mick` (address 1000). Thus, the value 1000 will be loaded into register `r0`. This is somewhat of a simplification of the actual mechanism performed (due to the pipelined nature of the ARM microprocessor), nonetheless it suffices for explanatory purposes. It should be emphasized that the translation is performed by the assembler. As far as the assembly language programmer is concerned, one can proceed making use of the original pseudo-instruction. The assembler will take care of all the low level details of the translation. The assembler is quite sophisicated in that it is even possible to specify instructions of the form:

```
ldr r0, =mick+1
```

which in the given `.data` section would correspond to the address of label `keith`.

LINKER SCRIPTS

A linker script is an ASCII text file written in a specific format which controls how the linker maps the various portions of a assembly language program (the `.data` and `.text` sections) to the main memory of the target microcontroller. By convention, linker scripts are named after the intended microcontroller, so create a text file with name “stm32f334r8_ALL_IN_RAM.ld” containing:

```
MEMORY {
    FLASH : ORIGIN = 0x8000000,  LENGTH = 64K
    SRAM  : ORIGIN = 0x20000000, LENGTH = 16K
}

SECTIONS {
    .text : {
        *(.text)
   } >SRAM

   .data  : {
       *(.data)
   } >SRAM
}
```

The above is a ‘bare-bones’ linker script for the STM32F334 microcontroller. The first part of the script (the `MEMORY` section) simply defines the various portions of the microcontroller’s main memory. The starting addresses and their sizes are specified. The STM32F334 microcontroller has 64 Kbytes of flash (non-volatile) memory starting at address `0x8000000` and 16 Kbytes of SRAM (static RAM volatile) beginning at address `0x20000000.`

The second part (`SECTIONS`) direct the linker to add the .text section of the program at the starting address of the SRAM followed by the .data section [2]. In this script, both the machine code and data will be stored in the SRAM. This linker script shall be used for all the remaining labs.

The astute reader may have noticed that the starting address of flash (`0x8000000`) is the same address which was specified with the `-T` option for the `arm-none-eabi-ld` command used in Lab 1.

PROCEDURE

1. Use a text editor to create a file called “add_from_mem.s” containing the ARM assembly language program given in page 2.

2. Assemble the source code in the usual manner:

```
arm-none-eabi-as -g add_from_mem.s -o add_from_mem.o -al=add_from_mem.lst
```

3. Link the program using the name of the linker script as the -T option to the loader:

```
arm-none-eabi-ld add_from_mem.o -o add_from_mem.elf -T stm32f334r8_ALL_IN_RAM.ld
```

Note: the linker script should be in the same directory as the .o file.

4. Connect the microcontroller board to the USB port of the host PC and in a terminal window start the ‘openocd’ monitor program:

```
openocd -f board/st_nucleo_f3.cfg
```

5. Single step through the program with gdb:

```
5a. arm-none-eabi-gdb add_from_mem.elf
GNU gdb (GNU Toolchain for the Arm Architecture 11.2-2022.02 (arm-
11.14)) 11.2.90.20220202-git
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licens-
es/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu --tar-
get=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.linaro.org/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from add_from_mem.elf...

5b.(gdb) target extended-remote localhost:3333
Remote debugging using localhost:3333
start () at add_from_mem.s:20
20              ldr r0, =mick  @ load address of mick into r0
5c.(gdb) monitor reset halt
Unable to match requested speed 1000 kHz, using 950 kHz
Unable to match requested speed 1000 kHz, using 950 kHz
target halted due to debug-request, current mode: Thread
```

```
xPSR: 0x01000000 pc: 0x800000ec msp: 0x20000400
```
5d.
```
(gdb) load
Loading section .text, size 0x10c lma 0x20000000
Loading section .data, size 0x3 lma 0x2000010c
Start address 0x20000000, load size 271
Transfer rate: 44 KB/sec, 135 bytes/write.
```

Note the starting addresses of the .text and the .data sections. The machine code was loaded into SRAM starting at address `0x20000000` and is 0x10C bytes long. The .data section was loaded into SRAM following the machine code starting at address `0x2000010c`.

5e.
```
(gdb) break start
Breakpoint 1 at 0x200000ec: file add_from_mem.s, line 20.
(gdb) continue
Continuing.

Breakpoint 1, start () at add_from_mem.s:20
20              ldr r0, =mick  @ load address of mick into r0
```

Note how gdb has stopped at the first instruction of the program (which is the pseudo-instruction which as it appears in the source file).

We will use the `x` (examine) command to view the contents of main memory corresponding the label mick:

5f.
```
(gdb) x/3xb &mick
0x2000010c:       0x01    0x02    0x00
```

or alternatively:

```
(gdb) x/3xb 0x2000010c
0x2000010c:       0x01    0x02    0x00
(gdb)
```

The /3xb is used to specify that 3 bytes are to be examined and displayed in hexadecimal. Note that the address of mick is `0x2000010c`.

5g. Single step thorough the program and examine the various register contents as the program proceeds in its exectuion:

```
 (gdb) stepi
halted: PC: 0x200000ee
21             ldrb r1, [r0]   @ load r1 with memory byte contents
of mick
(gdb) disassemble
Dump of assembler code for function start:
   0x200000ec <+0>:        ldr     r0, [pc, #16]    ; (0x20000100
<stop+4>)
=> 0x200000ee <+2>:        ldrb    r1, [r0, #0]
   0x200000f0 <+4>:        ldr     r0, [pc, #16]    ; (0x20000104
<stop+8>)
   0x200000f2 <+6>:        ldrb    r2, [r0, #0]
   0x200000f4 <+8>:        add.w   r3, r2, r1
   0x200000f8 <+12>:       ldr     r0, [pc, #12]    ; (0x20000108
<stop+12>)
   0x200000fa <+14>:       strb    r3, [r0, #0]
End of assembler dump.
(gdb) print/x $r0
$1 = 0x2000010c
```

Note the disassembly output, the original pseudo-instruction has been replaced with:

```
ldr     r0, [pc, #16]    ; (0x20000100 <stop+4>)
```

The significance of the `(0x20000100 <stop+4>)` is that the data in the literal pool begins at address `0x20000100`. Another way to interpret is the when we add the contents of PC with the offset 16 (decimal value) the result is `0x20000100.`

5h. Let us examine the 4 bytes found starting at this address:

```
(gdb) x/4xb 0x20000100
0x20000100 <stop+4>:        0x0c    0x01    0x00    0x20
```

The 4 bytes of data found are the 4 bytes of the label `mick` (stored in reverse order - so called *"little endian"* byte ordering). Since the label `keith` appears 1 byte after `mick`, the address of `keith` should be `0x2000010d`. This is easily verified with:

(gdb) x/1xb &keith
0x2000010d: 0x02

Label `result` is 1 byte after `keith`:

(gdb) x/1xb &result
0x2000010e: 0x00

All three of the addresses are contained within the literal pool starting at address `0x20000100:`

```
(gdb) x/15xb 0x20000100
0x20000100 <stop+4>:       0x0c    0x01    0x00    0x20    0x0d
0x01    0x00    0x20
0x20000108 <stop+12>:      0x0e    0x01    0x00    0x20    0x01
0x02    0x00
```

```
(gdb)
```

The 4 bytes in black color correspond to the 4 bytes of the address of label mick, the 4 bytes in red correspond to the 4 bytes of label keith, and the 4 bytes in blue are the 4 bytes of address result. We also see the actual data of 0x01 and 0x02 and the uninitilialized space reserved for the result at addresses `0x2000010c`, `0x2000010d`, and `0x2000010e` respectively.

5i. Single stepping through the program, we can verify that the program obtains the memory operands and loads the two operands into registers, computes the sum, and saves the result back into memory.

```
(gdb) stepi
halted: PC: 0x200000f0
22              ldr r0, =keith @ load address of keith into r0
(gdb) stepi
halted: PC: 0x200000f2
23             ldrb r2, [r0]   @ load r2 with memory byte contents
of keith
(gdb) stepi
halted: PC: 0x200000f4
24              add r3, r2, r1 @ r3 = r2 + r1
(gdb) stepi
halted: PC: 0x200000f8
25              ldr r0, =result @ load address of result into r0
(gdb) info register
r0             0x2000010d          536871181
r1             0x1                 1
r2             0x2                 2
r3             0x3                 3
r4             0x0                 0
r5             0x0                 0
```

(rest of the registers omitted for brevity)

```
(gdb) stepi
halted: PC: 0x200000fa
26             strb r3, [r0]   @ store sum into memory at location
result
(gdb) stepi
halted: PC: 0x200000fc
stop () at add_from_mem.s:27
27      stop:   b stop
(gdb) x/1xb &result
0x2000010e:     0x03
(gdb) quit
A debugging session is active.
```

```
        Inferior 1 [Remote target] will be detached.

Quit anyway? (y or n) y
Detaching from program: /nfs/home/t/ted/COEN311/ARM_LABS/Code/
ADD_FROM_MEM/add_from_mem.elf, Remote target
[Inferior 1 (Remote target) detached]
ted@deadflowers ADD_FROM_MEM 7:09pm >
```

QUESTIONS

1. Write a complete ARM assembly language program to compute the vector dot product of two arrays stored in main memory. For simplicity, each array will consist of three bytes. Recall that the vector dot product of two arrays is given by:

dot product = (a[0] * b[0]) + (a[1] * b[1]) + (a[2] * b[2]) .

As a starting point, here is a C++ program which employs a simplistic "straightline" approach to compute the dot product of two arrays:

```
// Ted Obuchowicz
// March 17, 2023
// dot_product.C
// simple straightline execution to
// compute dot product of two array

#include <iostream>
using namespace std;

int main()
{
 int mick[3] = {2,3,4} ;   // the first array
 int keith[3] = {5,6,7} ;  // the second array
 int dot ;                 // will hold the answer

 int sum = 0 ;  // will hold the running sum of
 int mult ;     // the product of mick[i] * keith[i]

 mult = mick[0] * keith[0] ;
 cout << mult << endl ;
 sum  = sum + mult;
 cout << mult << " " << sum << endl ; // for testing purposes only

 mult = mick[1] * keith[1] ;
 sum  = sum + mult;
 cout << mult << " " << sum << endl ; // for testing purposes only
mult = mick[2] * keith[2] ;
 sum  = sum + mult;
 cout << mult << “ “ << sum << endl ; // for testing purposes only

 // we are done

 return 0;
}
```

Use a similar straightline approach in your assembly language program. The ARM instruction set contains a multiply instruction and an add instruction, both of which expect the input operands to be in registers and writes the result to a register. The general form of these arithmetic instructions are:

```
add Rdest, Rsrc1, Rsrc2  @ Rdest = Rsrc1 + Rsrc2
                         @ Rsrc2 can also be an immediate value
                         @ instead of a register

mul Rdest, Rsrc1, Rsrc2  @ Rdest = Rsrc1 x Rsrc2
                         @ all three operands MUST be registers
```

Assemble, link and single step through your code with gdb. Examine the memory locations where the data and result are stored both before and after the execution of the program. Use the examine memory command to view the contents of the literal pool showing the bytes which compose the addresses of the labels used in your assembly code. Make use of the following .data section:

```
mick:   .byte  2,3,4 @ declare 3 bytes of data starting at
                     @ address mick
keith:  .byte  5,6,7 @ declare 3 bytes of data starting at
                     @ address keith
dot:    .space 0x01  @ reserve 1 byte of RAM  without any
                     @ initialization
```

Include the .lst file in your lab report along with the relevant portions of the various gdb produced outputs.

REFERENCES
1. ARM Cortex-A Series Version: 4.0 Programmer’s guide, ARM DEN0013D, 2013, p.5-2.
2. ARM-ASM-Tutorial, Niklas Gurtler, https://www.mikrocontroller.net/articles/ARM-ASM-Tutorial, 2022, p. 32.

T. Obuchowicz
March 17, 2023