Lab 5: Program Memory

Purpose/Scope: The purpose of this lab is to understand how program instructions are put into the processor address space, the relationship with the program counter, and how programs instructions and data coexist in the address space. For that purpose, we will manually encode, program, and execute machine instructions.

We will also practice using objdump and gdb to locate label addresses and the memory and data they reference.

Concepts: Binary encoding of assembly instructions into machine code
Relationship of program counter and machine code
Learn gdb commands to write and verify values in memory and registers
Use objdump to to examine memory and data values and decode machine code

The items specified in the red boxes are required for inclusion in your journal and need to be clearly marked by section and item. Please feel free to include any other information that you think would be helpful in the future, but it is not required.

Lab 5 Tasks

To have total control of what happens in the processor, we will erase the flash so that nothing remains in memory – no stray instructions to cause anything to happen even by accident ("nothing up my sleeve!"). Then we will manually encode, program, and execute instructions, using the processor RAM.

Part 0: Clear the flash memory for the F4

From a terminal window issue the following:

```
$ openocd -f board/stm32f4discovery.cfg
```

From another terminal window issue the following gdb commands:

```
$ arm-none-eabi-gdb -q
(gdb) target remote localhost:3333
(gdb) monitor reset halt
(gdb) monitor flash erase_address 0x08000000 0x00100000
(gdb) monitor flash erase_check 0
```

This erases the flash (program) memory on the F4 board. It takes a while (about 30 seconds) so cool your jets and wait for the erase_address command to return before proceeding.

Note: the commands used above are the same commands that we used in Lab01. The "monitor" command passes the rest of the command line to the gdb server that is the target of the session. This is required so that gdb knows not to try to process the command since the command is meant for the on-chip debugger.

The following will introduce (or re-introduce) a few more gdb commands that you will need for this lab. NOTE: These are only examples – modify for your specific use!

Useful gdb command examples (same as in pre-lab):

1. Show the current values assigned to registers:

```
(gdb) info reg
. . .

(gdb) info reg r2 r3 pc
. . .

(gdb) print $r2
. . .
```

2. Assign new values to registers:

```
(gdb) set $r4 = 123

(gdb) set $pc = 0x20000010
```

3. Write a byte to memory 0x20000000:

```
(gdb) set *(unsigned char *)0x20000000 = 0xab
```

   Verify:

```
(gdb) x /1xb 0x20000000
```

4. Write a 16-bit value to memory 0x20000002:

```
(gdb) set *(unsigned short *)0x20000002 = 0xcdef
```

   Verify:

```
(gdb) x /1xh 0x20000002
0x20000002:     0xcdef

(gdb) x /2xb 0x20000002
0x20000002:     0xef    0xcd
```

5. Write a 32-bit value to memory 0x20000004:

```
(gdb) set *(unsigned int *)0x20000004 = 0x12345678
```

Verify:

```
(gdb) x /1xw 0x20000004
0x20000004:     0x12345678

(gdb) x /4xb 0x20000004
0x20000004:     0x78    0x56    0x34    0x12
```

6. Examine machine instruction (set casting for 16 or 32 bit!):

```
(gdb) set *(unsigned short *)0x20000008 = 0x0a23

(gdb) x /i 0x20000008
0x20000008:  lsrs    r3, r4, #8

(gdb) set $pc = 0x20000008

(gdb) x /i $pc
0x20000008:  lsrs    r3, r4, #8
```

Part 1: Manually Program the F4 Processor

This section will require that you use gdb to place the machine code into the processor's SRAM address space at the specified addresses. You will also need to make sure that the data specified is available at the appropriate locations (registers and/or memory) to match the requirements of the instruction. Finally, you will need to make sure that the program counter is set to execute the proper instruction in memory (your pre-lab is useful here).

The gdb tool usually works with program files that have debugging information supplied in the executable file, and coordinates the execution from the perspective of the source code that the program is written in. Since we will manually enter the machine code into the processor, gdb has no source code to reference (there is no "make"). For this situation, instead of using the step command (which is stepping for a source instruction) we need to instruct gdb to step at the machine level. This is done using the "stepi" instruction.

Use the gdb commands supplied in Part 0 above to complete Parts 1a-d.

Part 1a: Execute the add instruction

Fill in the blanks

Machine code (Arm instruction):	0x0306EB05
Decoded machine instruction (assembly):	_____
Address for instruction:	0x20000000
Values for r5 and r6:	$r5: 0x55555555 $r6: 0x77777777
Value assigned to r3 (after stepping):	_____

For your journal:

Screenshot (not list) of gdb commands to set up and execute instruction for Part 1a – show registers at the end!

Part 1b: Another add instruction

Fill in the blanks

Machine code:	`0x0791F107`
Decoded machine instruction (assembly):	_____
Address for instruction:	`0x20000010`
Original value for r7:	`50`
New value for r7:	_____

For your journal:

Screenshot (not list) of `gdb` commands to set up and execute instruction for Part 1b – show registers at the end!

Part 1c: The ldr instruction

Fill in the blanks

Machine code:	0x69A2
Decoded machine instruction (assembly):	_____
Address for instruction:	0x20000020
Value for r4:	0x20000030
Effective address of data:	_____
Data placed at effective address:	0x12345678
New value for r2 (after execution):	_____

For your journal:

Screenshot (not list) of gdb commands to set up and execute instruction for Part 1c – show registers at the end!

Part 1d: Mystery instruction

Machine code: `0x41cb`

Fill in the blanks

Address for instruction: `0x20000050`

Decoded machine instruction (assembly): _____

Input register values: `0x2f5d08dd and 7`

Result register and its value: _____

Note: Once you've decoded the machine instruction, you should be able to choose the appropriate registers to put the supplied data values into. The resulting value after execution should be a recognizable hexword.

For your journal:

Screenshot (not list) of `gdb` commands to set up and execute instruction for Part 1d – show registers at the end!

Part 2: Accessing data in program memory

In Lab 4 we worked with data in the data memory (SRAM – read/write access). Program memory can also be SRAM (read/write), but many times it is flash (read only - program code). However, if some of our data are constants or need to start with certain values, it can be beneficial to have data in the program region of memory.

The cortex-m4 has 128 ki of data memory (SRAM) starting at address 0x20000000 and ending at 0x2001ffff. The program memory, 1 MByte (flash), starts at 0x08000000 and ends at 0x080fffff.

Part 2a: Review of accessing data in SRAM

Before working with data in the .text section (program memory), we will review the process of accessing data in the .data section. Below is code that was covered in Lab 4.

Use "create-project p2a" command and modify the main_asm.s file to look like the following. Save it as "lab05p2a.s" and modify the makefile with that same file name.

```
1  # file: lab05p2a.s
2
3  .include "macros.inc"
4
5    SET_TARGET
6
7    .text
8
9    FUNCTION main,global
10
11     push {r4,r5,r6,r7,lr}
12
13     ldr  r4,=A
14     ldrh r5,[r4]
15
16     all_done:
17
18     pop {r4,r5,r6,r7,lr}
19
20     bx lr
21
22   ENDFUNC main
23
24   .data
25
26 A: .short 0xabcd
27
28   .end
29
```

data in .data section

The following disassembly of the elf file shows that the ldr r4,=A instruction stores the address referenced by the label A in the program space (in a region called the literal pool). NOTE: The disassembly option below is "-dS"!!!!!!!!!

```
$ make all
$ arm-none-eabi-objdump -dS bin/p2a.elf
. . .
```

```
   ldr  r4,=A
 80001e6:       4c02            ldr     r4, [pc, #8]    ; (80001f0 <all_done+0x6>)

. . .

   ldr  r4,=A
 80001f0:       20000000        .word   0x20000000
```

objdump of the data section

This shows that the data referenced by the =A parameter is at location 0x080001f0 (in the .text section) and the value is 0x20000000.

Since the data being reference is in the .data section we can dump the memory image using the objdump utility to display the SYMBOL table and view the memory referenced by the label A.

```
$ make all
$ arm-none-eabi-objdump -s -t --section=.data bin/p2a.elf

bin/p2a.elf:     file format elf32-littlearm

SYMBOL TABLE:
. . .
20000000 l       .data  00000000 A
. . .

Contents of section .data:
 20000000 cdab0000 00000000 20030000 40030000  ........ ...@...
 20000010 60030000 00000000 00000000 00000000  `...............
```

objdump of the data section

Using gdb, we can locate the address and data associated with the label A as follows:

```
$ make debug

(gdb) x /2xb &A
0x20000000:     0xcd    0xab
(gdb) x /1xh &A
0x20000000:     0xabcd
```

Using gdb to locate and display data reference by a label

Using your own copy of objdump and gdb "make debug" session, show a TA how you find the answers to the following questions:

1. What is the address of the data referenced by the label A?
2. What is the data referenced by the label A?
3. What is the memory referenced by the label A?

Part 2b. Accessing data in the .text section (program memory)

Using the same basic program, but moving the data to the .text section, results in the following code.

Use "create-project p2b" command and modify the main_asm.s file to look like the following. Save it as "lab05p2b.s" and modify the makefile with that same file name.

As instructed in Part 2a, do a "make all" and run objdump to first find the address location and data for labels DS, A, and B.

```
1  # file: lab05p2b.s
2
3  .include "macros.inc"
4
5    SET_TARGET
6
7    .text
8
9    FUNCTION main,global
10
11     push {r4,r5,r6,r7,lr}
12
13     ldr  r4,=DS
14     ldrh r5,[r4,A-DS]
15     ldr  r6,[r4,B-DS]
16
17     all_done:
18
19     pop {r4,r5,r6,r7,lr}
20
21     bx lr
22
23     .align 2,0xaa
24 DS:
25     .align 1,0xaa
26 A:   .short 0xabcd
27
28     .align 2,0xaa
29 B:   .word 0x12345678
30
31   ENDFUNC main
32
33   .end
34
```

data in .text section

For your journal:

1. What is the address and data referenced by the =DS parameter (for the ldr r4,=DS instruction). Provide summarized capture of the objdump results (include command used)

2. Provide a summarized capture of the objdump showing the memory referenced by the DS label. (This should include the data referenced by labels A and B). Hint: make sure that you are referencing the correct section - .text instead of .data this time!

3. Provide a screenshot of the gdb commands to display the memory at the A and B label addresses.

Part 2c. Another way of establishing addressability to data in the .text section.

Another way to establish addressability to data in the .text section is via the `adr` instruction.

1. Use the Cortex-M4 Generic User Guide entry on the ADR command and explain why this command only works to establish addressability for labels found in the .text section.

Use "create-project p2c" command and modify the main_asm.s file to look like the following. Save it as "lab05p2c.s" and modify the makefile with that same file name.

Just as in Parts 2a and b, do a "make all" and run objdump to first find the address location and data for labels DS, A, and B.

```
1  # file: lab05p2c.s
2
3  .include "macros.inc"
4
5    SET_TARGET
6
7    .text
8
9    FUNCTION main,global
10
11     push {r4,r5,r6,r7,lr}
12
13     adr  r4,DS
14     ldrh r5,[r4,A-DS]
15     ldr  r6,[r4,B-DS]
16
17     all_done:
18
19     pop {r4,r5,r6,r7,lr}
20
21     bx lr
22
23     .align 2,0xaa
24 DS:
25     .align 1,0xaa
26 A:  .short 0xabcd
27
28     .align 2,0xaa
29 B:  .word 0x12345678
30
31   ENDFUNC main
32
33   .end
34
```

Figure 7. Establish addressability to data in .text section

For your journal:

2. What is the address and data referenced by the DS parameter (for the "adr r4,DS" instruction). Provide summarized capture of the objdump results (include command used)

3. Provide a summarized capture of the objdump showing the memory referenced by the DS label. (This should include the data referenced by labels A and B). Hint: make sure that you are referencing the correct section.

4. Use the disassembly option with objdump find what the adr instruction is translated to by the ARM assembler program.

5. Provide a screenshot of the gdb commands to display the memory at the A and B label addresses.

Lab 5 Deliverables

- Journal with items as specified
  - Required files in submission: Contents of Lab05 in `readme.pdf`

Submit Lab05 via Blackboard Lab Assignments – you have 3 attempts.