

BLG212E Microprocessor Systems Project – 1

1. INTRODUCTION

In this project, you will design an emulator that emulates a subset of ARM Cortex M0 processor architecture. The emulator will emulate a preliminary CPU architecture, the memory system, and a subset of instruction set. You are expected to write the emulator in C/C++ language.

The emulator will take an ARM Cortex M0 assembly program and will emulate the results.

You will be given a library written in C that includes an ARM assembler and a virtual LCD implementation for a graphics interface. The library provides Application Programming Interface (API) functions to assemble the given assembly program, load it into memory and write/read LCD peripheral registers (see Section 2.5). You will also be provided a test assembly program to test your emulator.

2. EMULATOR COMPONENTS

The emulator will include the following components.

2.1 CPU Registers

The CPU registers will be represented by the following C struct:

```
typedef struct {  
    int32_t reg[16]; //reg[0-12] are general purpose registers, reg[13] is SP, reg[14] is LR, and reg[15] is PC  
    int32_t cpsr;    //status register  
}tCPU;
```

You need to initialize all registers to zero.

2.2 RAM

RAM will be represented by an unsigned char array as follows:

```
uint8_t RAM[0x200000];
```

2.3 ROM

ROM will be represented by an unsigned short array as follows:

```
uint8_t ROM[0x100000];
```

2.4 Emulator Library

You will be provided an emulator library which handles assembling your program as well as accessing the virtual LCD peripheral. The functions are explained below:

int32_t system_init();

Initializes the virtual LCD. You must call this function at the beginning of your main function in your emulator.

void system_deinit();

De-initializes the virtual LCD. You must call this function at the end of your main function before exiting the emulator.

int32_t load_program(char *path, uint8_t *rom);

It takes a string path to the input ARM assembly program, assembles it and loads it to the given ROM address. When calling this function, you should pass the path to the input assembly program to be executed and the rom array mentioned in Section 2.3. It returns 0 if the program could be loaded successfully or -1 if there was an error (e.g, the assembly program could not be found at the specified path).

int32_t peripheral_write(uint32_t addr, uint32_t value);

This function is used to write **value** to an LCD register at address **addr**.

int32_t peripheral_read(uint32_t addr, uint32_t *value);

This function is used to read a **value** from an LCD register at address **addr**.

The virtual LCD has the following registers at the given addresses:

0x40010000 LCD row register

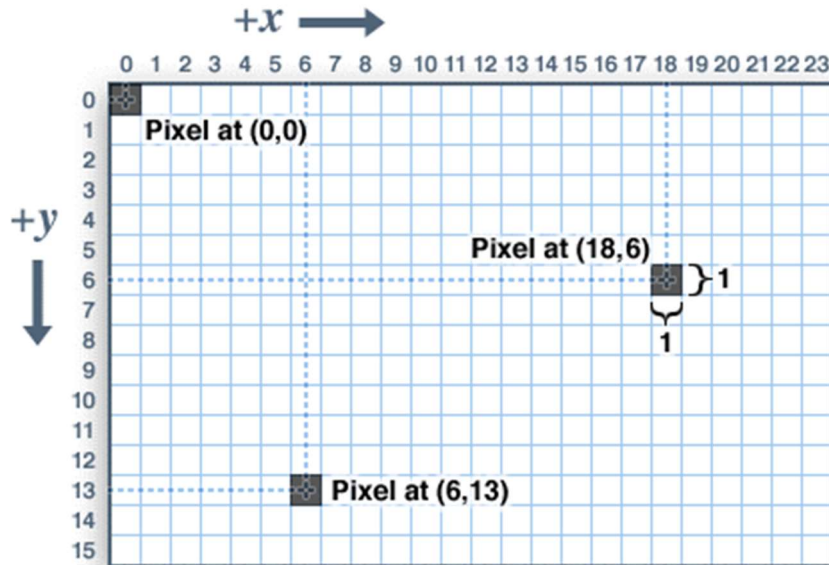
0x40010004 LCD column register

0x40010008 LCD color register

0x4001000C LCD refresh register

0x40010010 LCD clean register

LCD is made up of a pixel matrix display with rows and columns. It has a total size of 320x240 pixels (i.e., 240 rows and 320 columns). See below Figure for a sample 24x16 LCD screen representation.



The color information of an LCD pixel is encoded in ARGB format. **A** represents the opacity, **R** represents the red color amount, **G** represents the green color amount and **B** represents the blue color amount. Each component has a range between 0-255 which indicates the intensity of that color component. So, for instance, ARGB value of (255, 255, 255, 255) means a fully opaque white pixel, whereas ARGB value of (255, 255, 0, 0) represents a fully opaque red pixel (R component has the value 255 while G and B have values of 0). You should always set the opacity value to 255 in this project. In order to paint a pixel, say at position (6,13), to blue, one has to write 13 to the row register; 6 to the column register and ARGB value of (255, 0, 0, 255) or 0xFF0000FF to the color register. Then, in order to show the colored pixel, one has to refresh the screen by writing any value to the refresh register. Here are the assembly instructions for these steps:

```

movs r1, #13           //write 13 to the row register
ldr r2, =0x40010000
str r1, [r2]
movs r1, #6            //write 6 to the column register
ldr r2, =0x40010004
str r1, [r2]
ldr r1, =0xFF0000FF    //write blue color to the color register
ldr r2, =0x40010008
str r1, [r2]
ldr r2, =0x4001000C    //refresh the screen in order to show the painted pixel
str r1, [r2]

```

2.5 Emulator Implementation

The emulator you will write should take the assembly program to be executed from the command line. A skeleton C code (emulator.c) for this is provided to you in the project package. You can build up on this skeleton code.

When you call `load_program()` function from the emulator library, the program will be loaded to the beginning of ROM at address 0x0.

On ARM Cortex-M0, each instruction is 2 bytes. Therefore, when fetching an instruction from an address X from ROM, you need to read the byte at `ROM[X]` and the next byte at `ROM[X+1]` and combine these two bytes as follows to build up the fetched instruction's data.

```
inst = ROM[X] | ROM[X+1] << 8;
```

Furthermore, on ARM Cortex-M0, PC always points to 2 bytes after the next instruction to be fetched. For instance, if the next instruction to be fetched is at address 0x0, PC should be 0x2, or if the next instruction to be fetched is at address 0x10, PC should be 0x12, and so on. Therefore, when fetching an instruction from an address pointed by PC, you need to subtract 2 from current PC value and read the instruction at the resulting address as follows:

```
X = PC - 2;
```

```
inst = ROM[X] | ROM[X+1] << 8;
```

After fetching each instruction, you will need to decode it to determine:

1. The opcode of the instruction
2. The operand registers of the instruction
3. Any immediate value used
4. Any flags that the instruction may read or modify

Based on the opcode, you need to execute the instruction and update register file and/or RAM memory contents.

The information to be able to decode an instruction is provided in section **B2.2 of ARM-Thumb-instruction-encodings.pdf** file provided with the project bundle.

As an example, let's consider an instruction which has an encoded value 0x4329

This value has the binary equivalent:

0100 0011 0010 1001

From the **ARM-Thumb-instruction-encodings.pdf manual, table B2.5**, we can see that the first 8 bits of the instruction encoding, marked with red, tells us that this instruction is one of ORR | MUL | BIC | MVN instructions. Which one of these four instructions the encoding belongs to is determined by the “op” field in table B2.5. The “op” field is a two bits index value according to table B2.5 and is marked with blue above. It has the value 00. Therefore, the instruction we are looking for is the first one the possible 4 instructions above (since index value is 0), i.e., it is an ORR. If the op field was 1, it would be a MUL; if the op field was 2, it would be a BIC and if the op field was 3, it would be an MVN instruction.

Now let us look at the operands.

The source operand Lm has the value 101 (5 in decimal), which is marked with green, and the destination operand Ld has the value 001 (1 in decimal), which is marked with yellow above.

Therefore, this instruction is an ORR R1, R5 or ORR R1, R1, R5 in long form. Since ORR instruction can take only two operands in Cortex-M0, you can guess that one of the source operands must be same as the destination operand for Cortex-M0. This is a limitation of Thumb-16 instruction set (shorter instructions).

Important Note: To determine whether an instruction modifies any flags, and if so, which flags, you should refer to **CortexM0_QuickReferenceGuide.pdf** included with this project bundle.

For instance, according to **CortexM0_QuickReferenceGuide.pdf**, the ORR instruction modifies N and Z flags. Note that as we have seen in class, instructions that modify flags are normally suffixed with an “S”, For instance, ORR which modifies the flags should actually be ORRS. However, on Cortex-M0, with a few exceptions, every instruction by default modifies some flags. Therefore, S suffix is typically omitted in some manuals and even in some assemblers. To determine whether an instruction modifies any flags, you should always refer to the Quick Reference Guide. For instance, according to the Quick Reference Guide, this form of ADD does not modify any flags:

ADD Rd, SP, #<imm>

While this form modifies N, Z, C and V flags

ADDS Rd, Rd, #<imm>

According to the Quick Reference Guide, there is only one form of ORR which is

ORRS Rd, Rd, Rm

which modifies the N and Z flags by default. Therefore, the instruction we have just decoded must be modifying those flags even though “S” suffix is omitted.

As mentioned in Section 2.4, the LCD has some registers at certain memory mapped addresses such as 0x40010000, 0x40010004, etc. Therefore, whenever the assembly code writes to or reads from those addresses, your emulator should make calls to **peripheral_write()** and **peripheral_read()** functions from the library. For instance, assume we are emulating the assembly code given at the end of Section 2.4 and the emulator is about to execute the third instruction in the assembly program:

```
str r1, [r2]
```

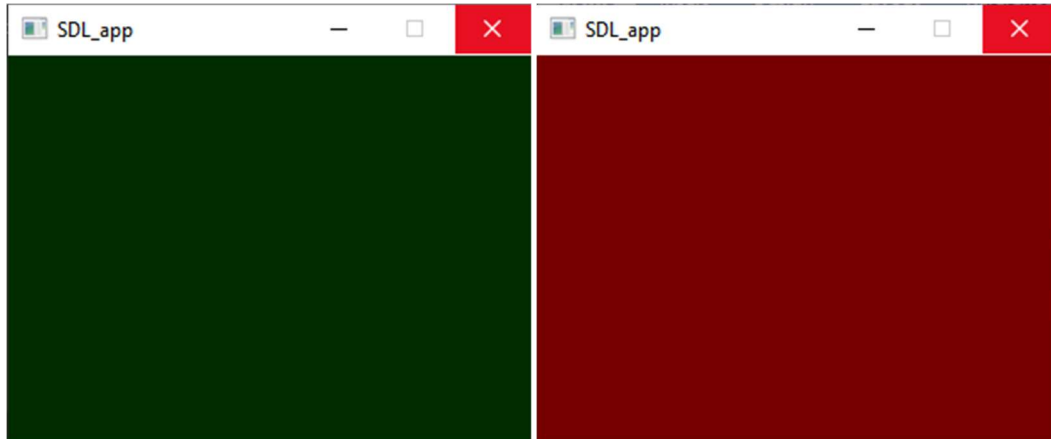
where r2 has the value of 0x40010000. This is a memory write to address 0x40010000, which is one of the peripheral registers of the virtual LCD. Therefore, your emulator should make the following call to the LCD library when emulating above str instruction:

```
peripheral_write(cpu.reg[2], cpu.reg[1]);
```

That is, perform a peripheral write of the value stored in CPU register R1 to the address stored in CPU register R2 as the str instruction above tells us to do.

3. Test Program

You will be given an ARM assembly test code to test your emulator functionality. The assembly code when run via your emulator should display color patterns changing from blue, green and red colors continuously. Example screenshots are given below.

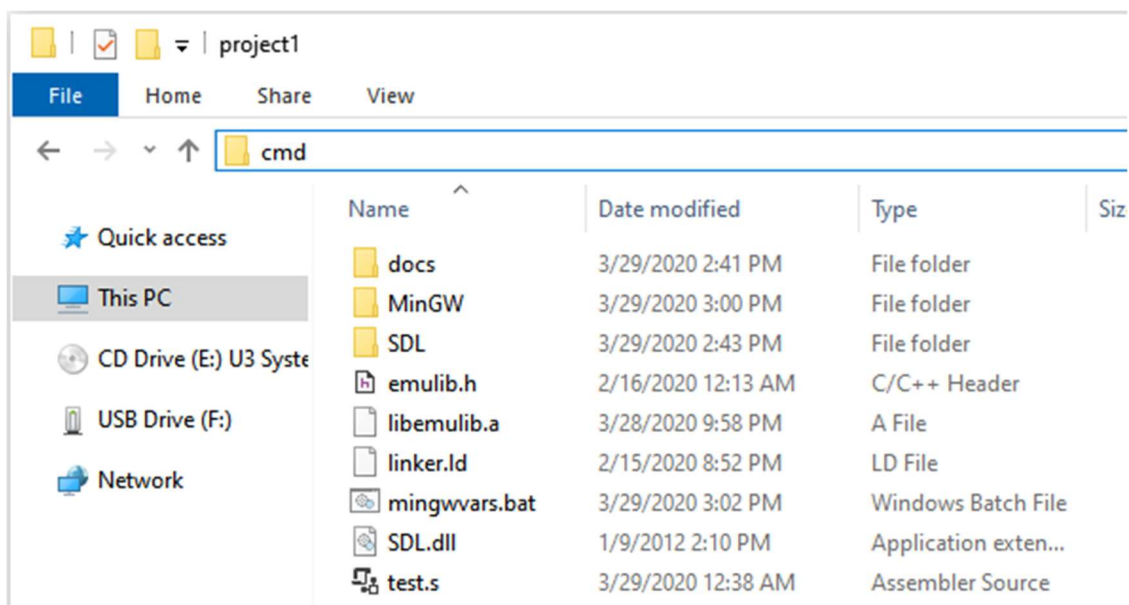


4. Compiling your Emulator

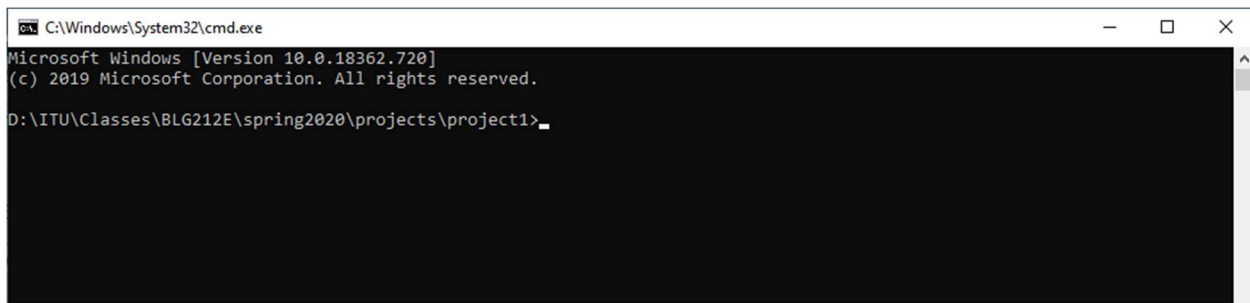
You can use the compiler of your choice but GNU gcc compiler is the recommended compiler. The project bundle has the MinGW compiler that you can readily use for compiling your emulator on a Windows operating system.

To run the compiler, follow these steps:

1. Unzip the project package to a folder of your choice.
2. Navigate to the extracted project package folder.
3. At the top address bar of the folder, type "cmd" and hit Enter (see below).



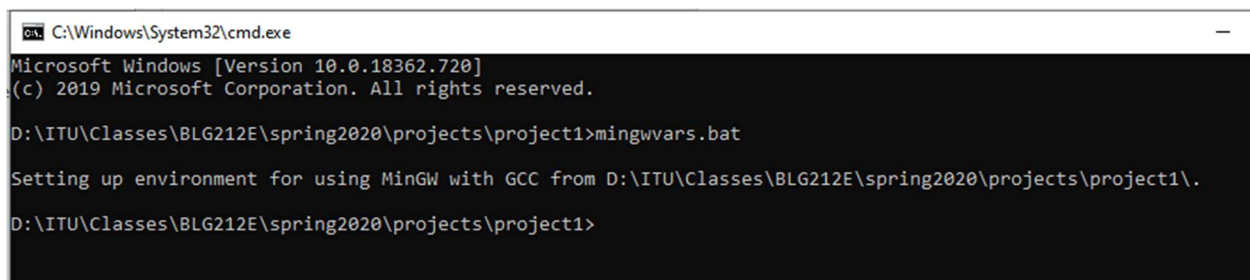
A DOS command prompt “DOS komut penceresi” will be opened as below.



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.18362.720]
(c) 2019 Microsoft Corporation. All rights reserved.
D:\ITU\Classes\BLG212E\spring2020\projects\project1>
```

*4. On the command window type the following command and hit Enter:

`mingwvars.bat`



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.18362.720]
(c) 2019 Microsoft Corporation. All rights reserved.
D:\ITU\Classes\BLG212E\spring2020\projects\project1>mingwvars.bat
Setting up environment for using MinGW with GCC from D:\ITU\Classes\BLG212E\spring2020\projects\project1\.
D:\ITU\Classes\BLG212E\spring2020\projects\project1>
```

You should see a message like “Setting up environment for using MinGW with GCC...”

This step is important. If you fail to run this step or if you get an error message, you won’t be able to run the compiler. Also, you must repeat this step if you close the DOS command window accidentally and re-open it.

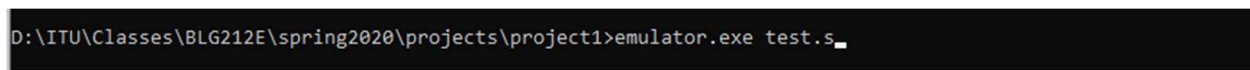
5. You can now compile your emulator source code. Assume you have given emulator.c to your source code. In order to compile it, type the following command into the DOS command window.

`gcc emulator.c libemulib.a -o emulator.exe -lSDL`

If your code compiles without errors, you should see emulator.exe generated in the same folder. Otherwise, you will see compilation errors which you need to fix in your code before re-compiling your code.

6. To run your code, type the name of the executable giving it the test.s assembly file as input:

`emulator.exe test.s`



```
D:\ITU\Classes\BLG212E\spring2020\projects\project1>emulator.exe test.s
```

If your emulator works correctly, you should see a window opening up and painting the colors as shown in Section 3.

5. Project Requirements

The emulator that you will develop should emulate the following instructions at least (refer to the Quick Reference Guide):

- All forms of MOV and MOVS instructions
- All forms of ADD and ADDS instructions
- All forms of SUB and SUBS instructions
- All forms of LDR instruction (excluding LDRH, LDRB, LDRSH, LDRSB, LDRM)
- All forms of STR instruction (excluding STRH, STRB, STRM)
- PUSH <loreglist> instruction
- POP <loreglist> instruction
- All conditional and unconditional branch instructions
- All forms of CMP instruction
- ORRS and ANDS instructions
- All forms of LSL, LSR and ASR instructions
- MUL instruction

6. Project Deliverables

Your submission should include a zip file containing a project report and the source code of your emulator. DO NOT INCLUDE ANY OTHER FILE.

1. You need to add comments to your code. Uncommented code will get partial credit. Be reasonable with the number of comments you add. Do not try to comment every line.
2. DO NOT submit files individually. Put them into a compressed zip archive and submit the zip archive. Name your zip archive with full name of your team members such as ahmet_bilir_and_veli_yapar.zip
3. Submit your homework zip file as an attachment to Ninova.

Below is the rubric for the project report.

Introduction

Briefly describe the project goals here

Team Members

Name the project team members and specify their roles in the implementation of the project.

Implementation

Here, you should describe the functional blocks of your emulator, how you implemented the ARM Cortex-M0 instructions, how you have interfaced with the virtual LCD.

Discussion

Briefly describe the problems you faced in the implementation of your project and how you could improve it.