M1 Embedded systems

Lab 3: ARM architecture and GPIO

Mohammed Akib Iftakher (20215282)

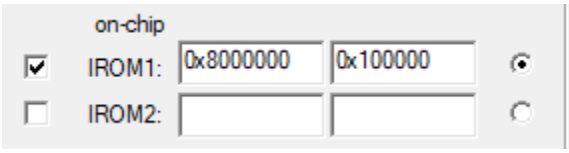
Mazhar Iqbal (20216524)

Lecturer: Professor Lamri Nehaoua

Immediate addressing mode

1. Show the reference of the µC as Target and report the following component-specific information.

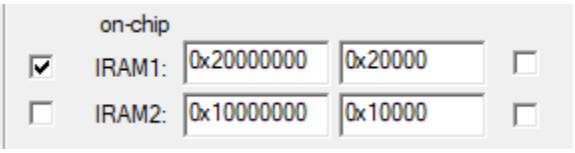
start address of the program memory	0x8000000
size of the program memory (in MByte)	1
start address of the data memory	0x20000000
size of the data memory (KByte)	128



0x00100000

So, $2^{20} = 2^{10} + 2^{10}$ Hence 1 MB

Figure 1: Program memory



0x00020000

So, $2^{17} = 2^{7} + 2^{10}$. Hence 128 kB

Figure 2: Data memory

2. What is the role of the xPSR register (expand its contents '+')

The Program State Register (PSR) is a vital register that continually updates the status of a running programme. It has condition code flags that may be modified when an ALU operation is performed. The xPSR is automatically updated when instructions are compared. The ALU is directly attached to the xPSR and may immediately change the xPSR registers based on the outcome of a computation (or comparison). The condition code flags are stored in the Application PSR(APSR).

bit[31] N Negative – If the outcome of a data processing instruction was negative, this bit is set.

bit[30] Z Zero – If the result was zero, this bit is set.

bit[29] C Carry – If the result of an operation is bigger than 32 bits, this bit is set.

bit[28] V Overflow - If the result of an operation was bigger than 31 bits, this bit is set, suggesting that the signed bit in signed numbers may have been corrupted.

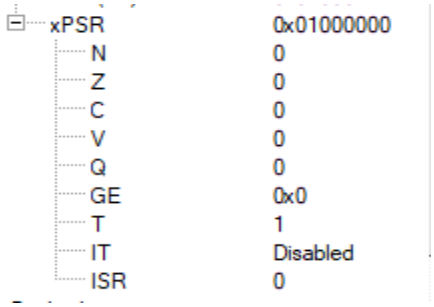
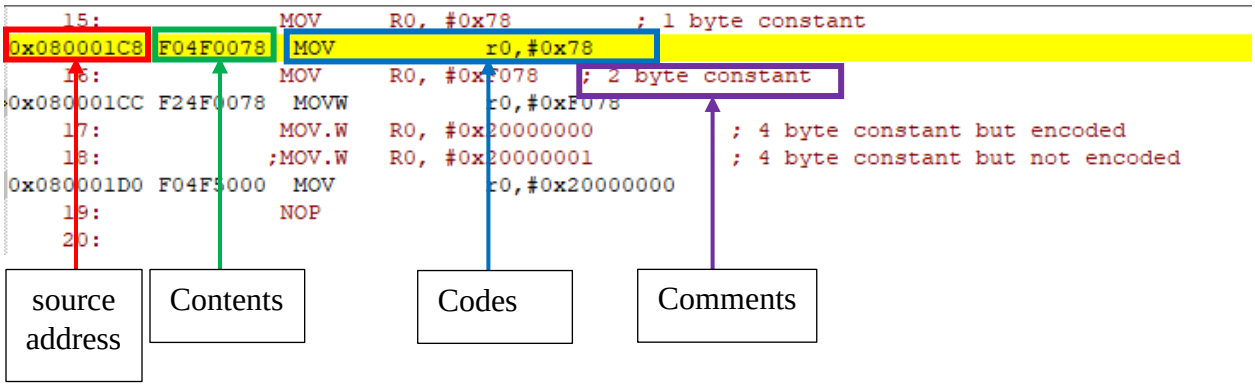


Figure 3: xPSR

3. Explain briefly the content of the Disassembly window.



Source Address: This is the automatically generated source code.

Contents: The contents associated with the source address which is little endian format.

Codes: The executed instruction codes which control the program.

Comments: Comments doesn't have any impact on the codes.

4. Locate the first NOP statement of the __main program and give the instruction address and the instruction code.

Instruction address	Instruction code	Contents	Register Values	Code Explanation
0x080001C8	MOV R0, #0x78	F04F0078	0x00000078	Immediate addressing is being applied. Here MOV is responsible for transfer of data and R0 is the destination register and #0x78 is the immediate value. So, R0 copies the value of #0x78.
0x080001CC	MOV R0, #0xF078	F24F0078	0x0000F078	Here, again immediate addressing has been used. But this time the immediate value has been 2-byte constant instead of one. So,
0x080001D0	MOV.W R0, #0x20000000	F04F5000	0x20000000	Here, R0 is loaded with 32-bit value.

5. Report in the displayed order, the 2 bytes of the instruction. Is the arrangement of the bytes is in Big endian or Little endian?

Instruction Address: 0x080001D4,

The displayed order: BF00,

The arrangement of the bytes: Little endian.

ARM cores support both modes, but are most commonly used in, and typically default to little-endian mode.

6. Run the program step by step and check the evolution of the contents of the R0 register. After the execution of the 4th instruction of the "generic" block, answer the followings:

the value R0	0x20000001
Is this the expected value?	Yes
Note the instruction used for `LDR R0, =0x20000001`	LDR r0,[pc,#48] the value of R0 is uploaded to Program counter+#48 initializing the system which is 0x080001E2 + #48. The contents are CONST data in Flash or ROM.
What the addressing mode used?	PC relative addressing mode has been used. Otherwise, it can't be encoded on 16 bits.

7. From which memory (RAM or ROM) does the value 0x20000001 come? What is its address? Check your answer using the "memory view" window (screen copy).

It came from the RAM and its address is 0x080001E2.

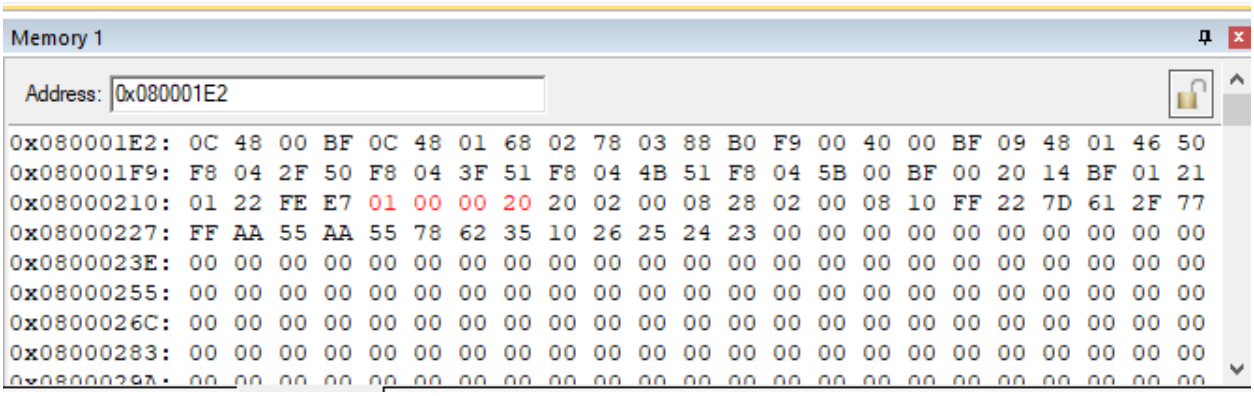


Figure 4: The screen copy of memory view.

Black - memory area or uninitialized RAM.

Red - CONST data in Flash or ROM that has been accessed at least once**.**

Gold - memory location that has been initialized, but not accessed yet.

Green - memory location has been accessed at least once.

8. Did the compilation go correctly? What do you think happened?

It gave an error at "Immediate 0x20000001 It can't be represented by 0-255.

Reset

9. Using the two lines; Vector Table Mapped to Address 0 at Reset, AREA RESET, DATA, READONLY, in which memory this zone is located? What is its starting address (zero is not the correct answer)?

It is located in ROM and its address is 0x080001AC. Here indirect addressing has been used. So, After executing the program, it can be seen that the value of R0 has been changed to 0x080001C9 which is same value pointed by 0x080001AC.

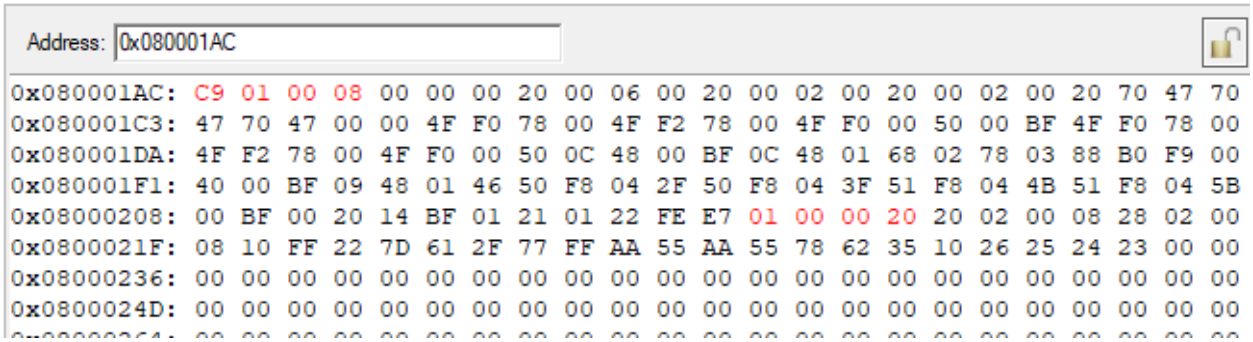


Figure 5: The Memory view which is red color explaining its content is in ROM.

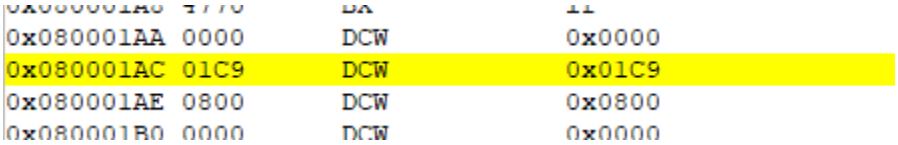


Figure 6: The value pointed by 0x080001AC.

10. Note the first two 32-bit words located at the beginning of this zone and give the value of PC and SP. What can you conclude about the contents of word1 and word2?

word1	C9010088
word2	00000020
PC	0x08000188
SP	0x20000600

word1 is in ROM and word2 is in RAM.

11. Report the content of R0 after running LDR (compare with what you found in question 4)? What type of addressing is used? What does the BX R0 instruction do? What is the usefulness of these first two instructions for?

The value of R0 register is 0x080001C9 and the contents are F078004FF278004F. Here, PC relative addressing is used.

0x08000188 4808 LDR r0[pc, #32]	This line is referring the pre-index. Here Load data from the memory unit pc+offset to r0 and

	offset is 32. In other word, it can be said that Constant loaded from literal pool address.
0x0800018A 4700 BX r0	The BX instruction causes a branch to the address contained in R0 and exchanges the instruction set, if required. So, in a word BX R0 basically derived the target instruction set from R0 means first ever instruction to be executed.

Indirect addressing mode without Offset

13. Indicate the address of the table named desoctets. Check its contents in memory.

The address: 0x08000218

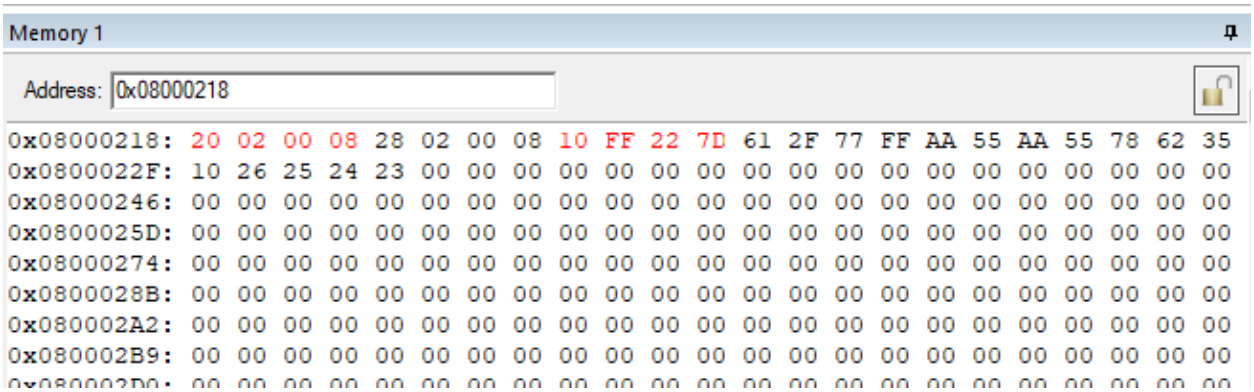
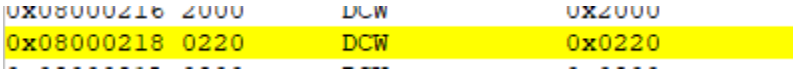


Figure 7: The contents in the desoctets.



The value 0220 has been loaded to R0 after the execution of first instruction.

14. Report the content of registers R0, R1, R2, R3, R4 after the execution of their respective loading instructions and explain what each instruction does.

Instructions	Register Value	Register Contents	Explication
`LDR R0, =desoctets`	R0= 0x08000220	0C480168…	R0 takes the pointer to the address of the array desoctets.
`LDR R1, [R0]`	R1= 0x7D22FF10	01680278…..	LDR R1 it loads it to R1.

`LDRB R2, [R0]`	R2= 0x00000010	02780388…	LDRB R2 it only loads the one byte of data from R1.
`LDRB R3, [R0]`	R3= 0x0000FF10	3388B0F9...	LDRH R3 loads the half word.
`LDRSH R4, [R0]`	R4= 0xFFFFFF10	B0F90040…	LDRSH R4 loads the Signed half word.

15. Why the content of R3 is different from R4?

The content of R3 and R4 are Different because we are loading the half word and in R4 we are loading the signed halfword.

Indirect pre-index addressing mode

16. Indicate the address of the table named des32bits. Check its content using the Memory window and indicate the storage method in memory Little/Big endian.

Address= 0x0800021C,

Contents = 28020008,

Method = Little endian.

17. Report the content of registers R0, R1, R2, R3, R4, R5 after the execution of their respective loading instructions and explain what each instruction does.

Instructions	**Register Value**	**Register content**	**Source Base address**	**calculated address before/after**	**Code Instruction**
`LDR R0,=des32bits`	R0 = 0x08000228	0948014650	0x08000228		
`MOV R1, R0`	R1 = 0x08000228	014650F804	0x08000228		
`LDR R2, [R0, #4]!`	R2 = 0x10356278	50F8042F50	0x080001F4	Before: 0x08000228 After: 0x0800022C	Here, R2 is loaded with Base memory Address of R0 + offset 4. At the same time the R0 address has been incremented by 4. So, R0 is new value is 0x0800022C.

`LDR R3, [R0, #4]!`	R3 = 0x23242526	50F8043F51	0x0800022C	Before: 0x0800022C After: 0x08000230	Here, R3 is loaded with contents of R0 + offset 4. Again, R0 base address has been incremented by 4. So, R0 is new value is 0x08000230.
`LDR R4, [R1], #4`	R4 = 0x55AA55AA	51F8044B51	0x08000228	Before: 0x08000228 After: 0x0800022C	Here, the R4 has been loaded with the same contents of R1 and then R1 register has been incremented by 4. So, new R1's value is 0x0800022C.
`LDR R5, [R1], #-4`	R5 = 0x10356278	51F8045900	0x0800022C	Before: 0x0800022C After: 0x08000228	Here, the R5 has been loaded with the same contents of R1 and then R1 register has been decremented by 4. So, new R1's value is 0x08000228.

Conditional execution

18. Report the values of R0, R1 and R2 before and after the execution of the instructions. Why R1 contains a different value from R2?

Before	R0=0x08000230	R1=0x08000230	R2=0x10356278
After	R0=0x00000000	R1=0x08000230	R2=0x00000001

The instructions are different that's why R1 and R2 contains different values. The zero flag is also changed according to the instruction. Here R0 is 0 so, it is activated zero flag which corresponds to MOVEQ. So, only R2 has been changed.

19. Modify the MOVS R0, # 0 instruction with a non-zero value. Report the values of R0, R1 and R2 before and after the execution of the instructions. Why R1 contains a different value from R2.

Before	R0=0x08000230	R1=0x08000230	R2=0x10356278
After	R0=0x00000002	R0=0x00000001	R2=0x10356278

Again, the instructions are different in this case. R0 is not zero anymore. So, MOVNE has been activated and changed R1 values.

Part 2: subprogram calls

1. Recall the 2 types of possible subroutines in an application.

The 2 types of possible subroutines in an application are 1. Leaf Routine and 2. Sub Routine

2. How many subroutines are used in this application? What is the hierarchy of calls?

two subroutines. Both of them used nested subroutines.

3. Note the value of the stack pointer SP register and explain briefly its role.

The value of SP is 0x20000600.

Stack memory is a memory usage mechanism that allows the system memory to be used as temporary data storage that behaves as a first-in-last-out buffer. One of the essential elements of stack memory operation is a register called the Stack Pointer. The stack pointer indicates where the current stack memory location is and is adjusted automatically each time a stack operation is carried out. Register R13 is used as a pointer to the active stack.

4. Report the generated code for the subprogram call instruction and explain its role. After executing the instruction, read and discuss the new values of the PC and LR register.

Here, R15 (PC) holds the address of the instruction being fetched from the memory. PC always returns the current instruction address + 4. On the other hand, R14 (LR) holds the return the address when the subroutine is called.

So, in BL (Branch Link) LR is assigned to $PC_{current}$ + 4 and PC is assigned to label (Subroutine's name).

That's why LR = 0x0800020A+4 = 0x0800020E. But Bit 0 of the LR is always 1 during a function call. The least significant bit is used to indicated whether the processor will run in Thumb state or ARM state when function returns. Since Cortex M always support Thumb state, the least significant bit of the LR is always 1 during a function call. So, the final LR is 0x0800020F.

The PC values is 0x080001E8.

5. Report the generated code for the subprogram call instruction and explain its role. After executing the instruction, read and discuss the new values of the PC and LR register.

Similarly, to the previous answer, here again, R15(PC) and R14(LR) is being used to holding the instruction address and returning address. After entering to the subroutine, **factorielle** some of the registers are initialized with corresponding values. Then a conditional loop **factofin** has been initiated. After the next instruction, the second subroutine has been called. Here, LR = 0x080001F6 + 4 =0x080001FA. But similar to previous case, Cortex M always support Thumb state, so LR = 0x080001FA+1 = 0x080001FB.

Since PC is storing the 2nd subroutines address, so PC = 0x080001C8.

6. Continue to step through the program until the end. What do you see? Why do you think that the program does not run as expected? (If you don't know how to answer this question, go to the next one).

As the instruction continues, couple of issues can be noticed.

After the completing the instruction of the second subroutines **multiplie_a_b**, the program returns to the first subroutines, using the saved address in LR. Here, the LR's saved address is 0x080001FB and PC will load this address and subtract 1 (0x080001FA) which will bring the previous value after reversing Bit 0 operation. But in the first subroutines, the saved address of main (___main) routine (or function) was removed before going to the second subroutines. As a results, the program can't go back to main routine (or function) and fall in the continuous loop or infinity loop.

In the Lab2.s file, uncomment the two stack management statements by removing the semicolons (Remember that any modification of the code requires compilation which cannot be executed in Debug mode).

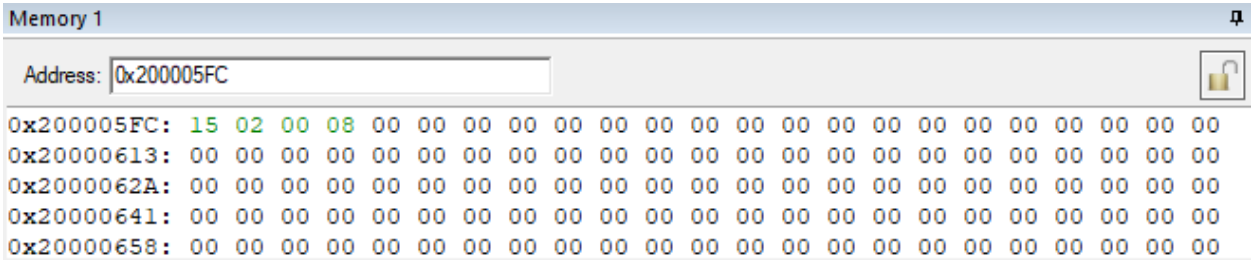
7. Briefly explain the role of these two instructions.

Before we can understand the instructions, we must first understand what a stack pointer is and how it relates to PUSH and POP. Pushing data to the stack (using the PUSH instruction) is known as pushing, while pulling data from the stack is known as popping (using the POP instruction). Depending on the CPU architecture, some processors utilise incremental address indexing to store new data to stack memory, while others use decrement address indexing. The stack operation in Cortex-M processors is based on a "full-descending" stack paradigm. This implies that the stack pointer always links to the most recently filled data in the stack memory, and it pre-decrements for each new data storage (PUSH).

PUSH and POP are widely used at the start and end of a function or subroutine, respectively. The current contents of the calling program's registers are placed onto the stack memory using PUSH operations at the start of a function, and the data on the stack memory is returned to the registers using POP operations at the end of the function.

8. Run the program in steps until the first stack management instruction. When finished, note the contents of the SP register. What does this value correspond to? Check your answer using a Memory window.

The first stack management instruction is PUSH {LR}. This instruction is storing the value of LR through Push operation in Stack Pointer register. The SP value is 0x200005FC which is corresponds to LR value 0x08000215. In the Memory window, the value is presented in little endian format.



9. Continue executing the program until the second stack handling instruction. At the end of its execution, record and discuss on the contents of the SP and LR registers.

Unlike the last program, the program continued until the second stack handling instruction. The second stack handling instruction is POP {LR}. Here the data stored in the SP register is restored back to LR register.

So, before the execution of the second stack handling instruction, the value of SP is 0x200005FC and LR is 0x080001FD. Then when the execution was completed the SP's value returns back to its original value 0x20000600 and LR's got back its first value 0x08000215 which it passed to PC later and PC took the program to the original main routine (or function).

Check that the made changes allow the program to run correctly.

At the end, the program successfully completed.

The final values are

SP = 0x20000600, LR = 0x08000215, PC = 0x08000214.

Reference

Girjau, Gheorghe. (2016, October 16). *Cortex M special purpose registers*. ggirjau. Retrieved December 12, 2021, from http://ggirjau.com/cortex-m-special-purpose-registers/.

Gopal. (2020, December 29). *ARM cortex-M startup code*. IoTality. Retrieved December 12, 2021, from https://www.iotality.com/arm-cortex-m4-startup-code/.

Yiu, J. (2015, June 26). *Architecture*. The Definitive Guide to Arm® Cortex®-M0 and Cortex-M0+ Processors (Second Edition). Retrieved December 12, 2021, from https://www.sciencedirect.com/science/article/pii/B9780128032770000047.