Lab session 3 & 4

Bare-metal ARM assembly programming

In the previous two lab sessions, we investigated the execution of Bare-metal C code on a Beaglebone ARM processor and the JTAG debugging tool. For this lab we will learn bare-metal ARM assembly programming using Beaglebone, JTAG and the Code Composer Studio (CCS). We will introduce the basics of ARM assembly programming. You will use what we introduced to accomplish some programming tasks. Please note that the instruction set we introduced in this lab is supported by ARMv7 processors. It is different from LEGv8 discussed in the lectures, which is part of the ARMv8 instruction set.

Step 1: Power up the Beaglebone board in Bare-metal mode and then connect the SEGGER JTAG debugger unit.

Step 2: Create a new CSS Project as we did at the start of Lab Session 1. Ensure that the target, debugger connection, and compiler version are the same as in Lab Session 1. Name the Project "EEEN301_Lab3_pt1", choose the Basic Examples -> Hello World template. Click finish to create the CCS Project.

Step 3: For this laboratory, we are going to insert an inline assembly function, which is stored in a separate file, into the main.c file. Edit main.c to contain the following:

```
#include <stdio.h>
extern void test();
int main(void)
{
        test();
        printf("Hello World!\n");
        return 0;
}
```

Step 4: Now we will create a new file test.S that contains the assembly code for the function test(). In the project explorer, right click on the project and select *New -> File*. Name the file test.S and click finish.

Step 5: Now populate the test.S file with the following assembly code.

```
.globl test
test:
        MOV r1, #1
        MOV r2, #2
        BX lr
```

Note: CCS may ask you to re-configure the debugger connection setting. If so, please follow step 4 of Lab session 1 to configure the connection properly. If CCS continues to generate error message regarding the connection setting, please repeat steps 1 to 4 of Lab session 1 to create a clean new project (in step 1 of Lab session 1, choose "Empty project (with main.c)"). Then copy *main.c* and *test.S* in EEEN301_lab3_pt1.zip to the newly created project to overwrite the existing *main.c* file in the new project. Afterwards, try to debug the code again.

Step 6: Expand the "Registers" window as per below.

Variables | Expressions | Registers

Name	Value	Description
Core Registers		Core Registers
PC	0x403008C4	Program Counter [Core]
SP	0x40300800	General Purpose Register 13 [Core]
LR	0x40300894	General Purpose Register 14 [Core]
CPSR	0x60000190	Stores the status of interrupt enables a
N	0	Stores bit 31 of the result of the instruc
Z	1	Is set to 1 if the result of the operation
C	1	Stores the value of the carry bit if it occ
V	0	Set to 1 if an overflow occurred
Q	0	Indicates whether an overflow or a satu
IT_1_0	00	IT state bits.
J	0	Java State Bit.
Reserved	0000	Reserved.
GE	0000	Greater than or equal bits
IT_7_2	000000	IT state bits
E	0	If set, data memory is interpreted as bi
A	1	If set, any asynchronous abort is held p
I	1	If set, IRQ is disabled. If cleared IRQ is a
F	0	If set, FIQ is disabled. If cleared FIQ is a
T	0	If set ARM is in Thumb mode
M	10000	Mode of ARM
R0	0x00000000	General Purpose Register 0 [Core]
R1	0x00000000	General Purpose Register 1 [Core]
R2	0x000059EB	General Purpose Register 2 [Core]
R3	0x00001999	General Purpose Register 3 [Core]
R4	0x00000000	General Purpose Register 4 [Core]

Check the **current program status register** (**CPSR**). The format of this register is shown in the figure below:

31	30	29	28	27	7	6	5	4	3	2	1	0
N	**Z**	**C**	**V**	**Unused**	**I**	**F**	**T**	**M4**	**M3**	**M2**	**M1**	**M0**

Let us focus now on the mode bits, M0, M1, M2, M3 and M4. The five mode bits together determine the ARM operation mode. An ARM processor can operate in one of the following operation modes:

- User mode: normal operation
- IRQ mode: mode for handling interrupt operations
- Supervisory mode: mode for the operating system
- FIQ mode: fast interrupt mode
- Undefined mode: when an undefined instruction is executed
- Abort mode: this mode indicates that the current memory access cannot be completed

Q1: What are the current values of the five mode bits?
Q2: What do you think is the current mode of your ARM processor when debugging just starts from the main() function and why?

Step 7: Open the *test.S* assembly file. Replace the code in *test.S* by the assembly code given below

```
.globl test
test:
        MOV r1, #1
        MOV r2, #2
        …
        BX lr
```

Please fill the … part in the above code respectively by each of the following instructions (fill the … part by each of the three instructions below one at a time):

```
ADD r1, r1, #10
AND r1, r2, #10
EOR r1, r2, #10
```

Add a breakpoint on the BX lr instruction, to prevent the code from executing the printf statement. Ensure that a breakpoint is placed before the printf statement on all steps of this laboratory script. Debug the code using the JTAG debugger.

Q3: For each of the instructions above, when we add it to test.S, which register will change its value as a result of executing test.S? What is the changed value and why?

Step 8: Some data processing instructions can change the flag bits in CPSR. Replace the … part in test.S (see step 7) by each of the following instructions (fill the … part by each of the two instructions below one at a time):

```
ADDS r1, r1, r2
SUBS r1, r2, r1
```

Q4: For each of the instructions above, when we add it to test.S, which flag bit will be set or unset as a result of executing the instruction? Explain your finding.

Step 9: The MOV instruction is very flexible in ARMv7. Particularly, we can perform logic shift, such as logic shift left (lsl), on the Rm register before copying its value to the Rn register in the instruction template below:

```
MOV Rn, Rm, lsl #n // Shift Rm n times to the left and store the result in Rn
```

Q5: Replace the … part in test.S (see step 7) by each of the following instructions (fill the … part by each of the four instructions below one at a time). As a result of executing each added instruction, what will be the new value in register r1 and why?

```
MOV r1, r2, lsl #2
MOV r1, r2, lsr #1 // lsr stands for logic shift right
MOV r1, r2, asr #1 // asr stands for arithmetic shift right
ADD r1, r2, r2, lsl #2
```

Q6: We can also set up conditions for the MOV instruction. Replace the … part in test.S (see step 7) by the following two instructions (fill the … part with both of the two instructions below), what will happen as a result of executing the added instructions and why?

```
CMP r1, r2
MOVMI r1, r2
```

Step 10: Similar to the MOV instruction, arithmetic instructions can include extra conditions. Please refer to the table below for different types of conditions that can be included in arithmetic instructions.

Condition Code	Condition	
0000	EQ	Equal
0001	NE	Not equal
0010	CS	Carry set
0111	CC	Carry is clear
0100	MI	Negative (N flag is set)
0101	PL	Positive (N flag is zero)
0110	VS	Overflow set
0111	VC	Overflow is clear
1000	HI	Higher for unsigned number
1001	LS	Less than for unsigned number
1010	GT	Greater for signed number
1011	LT	Signed less than
1100	GT	Greater Than
1101	LE	Less than or equal
1110	AL	Unconditional instructions
1111	Unused code	

For the example assembly code below

CMP r1, r2

ADDEQ r3, r4, r5

Register r3 will have its value updated to r4+r5 only when r1=r2.

*Q7: Convert the following pseudo-code into **THREE lines of ARM assembly code**. Insert the converted code into the ... part of test.S (see step 7). What will happen as a result of executing test.S and why? Please include your assembly code in test.S in your answer to this question. Please also include a screenshot that clearly shows the result of executing your assembly code.*

If r1>r2 Then r3=r1-r2

Else if r1<r2 Then r3=r1+r2

Step 11: ARMv7 provides dedicated instructions such as BL and BX to support function calls. Particularly, **BL subroutine_name** is used to call a subroutine (or function), where the return address will be saved in the link register lr (r14). On the other hand, we can use **BX lr** to return the control back to the function caller.

Q8: In test.S, add new assembly code to define a new function named func to add the integer values in registers r1 and r2 and save the addition result back to r1. Make necessary changes to the test function in test.S so the test function can call func() in the ... part (see step 7). You may need to preserve some registers in the test function before calling func(). After func() returns, you need to restore the original value of some registers in the test function too. Please include your assembly code in test.S in your answer to this

question. Please also include a screenshot that clearly shows the result of executing your assembly code.

Step 12: ARMv7 supports a range of conditional branch instructions as summarized in the table below:

Instruction	
B	Branch always
BAL	Branch Always
BEQ	Branch if Equal
BNE	Branch if Not equal
BPL	Branch on positive
BMI	Branch on negative
BCC	Branch if carry flag is clear
BLO	Branch below for unsigned number
BCS	Branch carry flag is set
BHS	Branch if higher for unsigned number
BVC	Branch if Over flow flag is clear
BVS	Branch if Over flow flag is clear
BGT	Branch greater for signed number
BGE	Branch greater or equal for signed number
BLT	Branch Less than for signed number
BLE	Branch Less than for signed number
BLS	Branch less than or equal for unsigned number

Q9: Using some of the conditional branch instructions above as well as instructions introduced earlier in this lab, convert the following pseudo-code to ARM assembly code. Assuming that variable a (signed number) is assigned to register r1 and variable b (signed number) is assigned to register r2, replace the … part of test.S (see step 7) by the newly converted assembly code. Please include your assembly code in test.S in your answer to this question. Please also include a screenshot that clearly shows the result of executing your assembly code.

```
a=  10;
b=45;
while ( a! =b ) {
       if ( a <  b)
              a   =   a  +5;
       else
              b= b+5;
 }
```

Step 13: With ARMv7 processors, we can access memory using several data transfer instructions, including LDR and STR. LDR is used to load a word into a register from memory. STR is used to transfer a word from a register to memory.

Q10: Replace the existing code in test.S completely by the assembly code given below. Execute test.S and identify any changes to register r1, after executing the instruction highlighted in blue. Explain what the blue instruction does.

```
.globl test
test:
        MOV r1, #1
        MOV r2, #2
        LDR r3,=numbers
        LDR r1, [r3,#8]
        BX lr

.data
numbers:
        .word   0
        .word   1
        .word   2
        .word   3
```

Q11: Replace the blue instruction in Q10 with the instruction given below. Identify the changes in memory and in register r3 because of executing the replaced instruction. Discuss what the replaced instruction does.

```
STR r2, [r3, #4]!
```

Step 14: The LDR and STR instructions enable us to handle arrays of arbitrary length in memory.

Q12. Following the assembly code given in Q10 as an example, re-define the test function in test.S to calculate the sum across an array of words. The sum result must be placed in register r0 before the test function returns. In test.S, you need to define the array to be stored in memory and associate the array with the label "numbers". We assume that the number of elements/words in the array is unknown upon writing the assembly code.

However, all elements of the array, except the last element, are non-zero. The last element is 0.

Please include your assembly code in test.S in your answer to this question. Please also include a screenshot that clearly shows the result of executing your assembly code.

Submission instruction

Please pay attention to the following while submitting your report for lab 3&4.

- Please provide a single report for all questions included in this lab. While this lab covers week 4 and week 5, you only need to write a single report for both weeks.
- Your report MUST be submitted in PDF format. Reports submitted in other formats will not be marked.
- In your report, please clearly separate and indicate your answer to each question. Please make sure that you include assembly code and screenshot requested in your answers to questions Q7, Q8, Q9, and Q12. For other questions, no source code and screenshot are required for your answer.