Experiment 26- Introduction to the ARM Microprocessor

Submission Template of Experimental Results

Name: RUIMIN ZHAO
ID: 201139030
Group: 44

Experimental Results and Comments (Total 90 marks):

Please provide the required screenshots, photos, code, values highlighted in the script according to the following requirements (screenshots should be clear and readable):

1 Code of the Hello World programme (Section 4) after editing (put as text NOT as a screenshot)[5 marks]

• Code
  – The code after changing “LED1” to “LED2” is presented below.

```c
#define _include "mbed.h" //library for mbed board

DigitalOut myled(LED2); //Declaration of one LED

int main()
{
    while(1) //Repeat these four line //in the loop
    {
        myled = 1;
        wait (0.2);
        myled = 0;
        wait (0.2);
    }
}
```
The modified code which enables two LEDs flash alternately is presented below.

```c
#include "mbed.h"

DigitalOut myled1(LED1); // Declaration of two LED
DigitalOut myled2(LED2);

int main()
{
  while (1)
  {
    myled1 = 1;
    myled2 = 0;

    wait(1); // The interval time between
    // the alternation is 1 second
    myled1 = 0;
    myled2 = 1;

    wait(1)
  }
}
```

The code with the smallest delays (wait) that allows user to see the LED flashing is presented below.

```c
#include "mbed.h"

DigitalOut myled1(LED1);
DigitalOut myled2(LED2);

int main()
{
  while (1)
  {
    myled1 = 1;
    myled2 = 0;

    wait(0.015); // The smallest delays that allows
    // seeing flash is 0.15s
    myled1 = 0;
    myled2 = 1;

    wait(0.015)
  }
}
```
Explaination

– For the first version of the code, the “LED1” is line 3 was changed to be “LED2”, which changes the declaration of the single LED. Before modification, the LED that can flash every 0.2 second is green LED while after changing the code the single LED that flashes is red LED instead.

– For the second version of the code, one more declaration of LED was added which means both two LEDs are declared. Also, to enable the user to visualize the alternate time clearly the delays were changed to be 1 second.

The screenshot of two seconds (0.2s and 0.3s) of the video of the alternately flashing LEDs are presented in Figure 1.

![Figure 1: LED flash alternately by 1 second](image)

It can be seen clearly from the above Figure that the enlightened LED is the green LED at 2 second while the enlightened LED is the red LED at 3 second of the video which verifies that using this modified code, two LEDs can indeed flash alternately and the delays are 1 second.

– For the third version of the code, the delays was changed to be 0.015s which was tested to be the smallest delay that allows user to see the LED flashing. This is related to the answer to Question 1 later.
2 Answer to Q1 [5 marks]

- **Answer**
  
  - The smallest delays that allows the user to see the flash is **0.15 second**, and the frequency it responds to is approximately **3.333Hz**.

- **Explanation**
  
  - For the smallest delays, this value was obtained by trying filling many values into the “wait()” and the correct value is 0.015 after trails and errors. This result can have a slight degree of error since this time is quite small and users may have different ability of recognizing flash alternating delays.

  - For the frequency, this result is obtained by calculation

    \[ f = \frac{1}{T} = \frac{1}{0.15 \times 2} \approx 3.333\text{Hz} \] (1)

3 Screenshot of the result of Section 5 part VIII (the output screen) [5 marks]

- **Screenshots**
  
  - The messages displayed on the screen using Putty based on the code with and without the added “\r” are presented respectively in Figure 2.

![Figure 2: Message based on the code with/without a carriage return](image)

- **Comment/Explanation**
  
  - For the first figure in Figure 2, it can be seen clearly that the numbers printed with a time delay one by one. Noticeably, in this situation, each one occupies a new line, which agree with the code “\n” for “printf” sentence. This is why the messages appear to “walk across the screen shown in the first figure in Figure 2.

  - For the second figure in Figure 2 however, each one occupies a new line and align with each other vertically, which agree with the code “\n\r” for “printf” sentence where a carriage return is added compared to the original code.
4 The modified code of section 6 (put as text NOT as a screen-shot)[5 marks]

- Code

```c
#include "mbed.h"
#include "SLCD.h"
#include "tsi_sensor.h"

// Very simple program to read the analog slider and print its value
// on the LCD. Also flashes the RED led.
// -- Al Williams
/* This defines will be replaced by PinNames soon */
#if defined (TARGET_KL25Z) || defined (TARGET_KL46Z)
#define ELEC0 9
#define ELEC1 10
#elif defined (TARGET_KL05Z)
#define ELEC0 9
#define ELEC1 8
#else
#error TARGET NOT DEFINED
#endif

TSIAnalogSlidertsi(ELEC0, ELEC1, 40);
DigitalOutpo(D0);
DigitalOutled1(LED_RED);
DigitalOutled2(LED_GREEN);
SLCD slcd;

int main()
{
    while (true)
    {
        float f = tsi.readPercentage();
        slcd.printf("%1.3f", f); // The percentage value has 3 decimal
                        // number after the decimal point

        if (f == 0) // When there is no finger touch
            { // no LED is illuminated
                led1 = 1;
                led2 = 1;
            }
        else if (f < 0.5) // When the finger touch area between
                        // left end and midpoint, green LED
                        // is illuminated
            { // green LED is illuminated
                led1 = 0;
                led2 = 1;
            }
        else // When the finger touch area between
            // right end and midpoint, green LED
            // is illuminated
            { // green LED is illuminated
                led1 = 1;
                led2 = 0;
            }
    }
}
```
• Explaination

– The there different types of displays on board are presented in Figure 3.

![Figure 3: There scenarios of the LED display](image)

– In the code, “f” represents the position where the finger touches the slider, a percentage from left (0) to right (1). Also, based practice, it was found that when LED is assigned value 1, this LED is off, if assigned 0, the corresponding LED is illuminated. The code was modified based on these core knowledge.

– If there is no finger touching the slider, the “float” type number “f” is equal to 0, both two LEDs remain off. While if the finger touch the area between the right end and the midpoint of the slider, “f” is smaller than 0.5 but bigger than 0, the right LED lights; If the finger touch the area between the left end and the midpoint of the slider, “f” is bigger than 0.5, the green LED lights.
5 Screenshot of the result of Section 7 part VIII[3 marks]

- Screenshots
  - The obtained Keil debugger debugging windows are presented in Figure 4.

![Keil debugger windows](image)

**Figure 4: Keil debugger windows**

- Comment/Explanation
  - The Registers window mainly display the values in every register and the statement of flags. It allows user to notice and easily follow any change of the values in registers.

  - The Disassembly window shows the machine code to carry out in this programme.
6 Answer to Q2[4 marks]

- **Answer**
  - The second digit.

- **Explanation**
  - As required by the script, the first two MOVS instructions are looked at and their related contents are presented in Figure 5.

![Figure 5: Keil debugger windows](image-url)
The related contents of the first two MOVS instructions with the second numbers of machine codes underlined were listed in Table 1.

<table>
<thead>
<tr>
<th>Program Counter</th>
<th>Machine Code</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000002E2</td>
<td>2700</td>
<td>MOVS r7, #0x00</td>
</tr>
<tr>
<td>0x0000002E4</td>
<td>2601</td>
<td>MOVS r6, #0x01</td>
</tr>
</tbody>
</table>

Table 1: The two MOVS instruction

It can be seen clearly that for both two instructions, the second number in the machine code is the same as the index of the register (2700 and r7; 2601 and r6).

In detail, we take the machine code “0x2700” as an example: The first digit “2” represents the type of machine code which is MOVS, and the second digit represents which register will load the value, which is register 7.

7 Answer to Q3[4 marks]

• Answer
  - The third and forth digits (the last byte of the machine code).

• Explanation
  - The related contents of the first two MOVS instructions with the third and forth numbers of machine codes underlined were listed in Table 2.

<table>
<thead>
<tr>
<th>Program Counter</th>
<th>Machine Code</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000002E2</td>
<td>2700</td>
<td>MOVS r7, #0x00</td>
</tr>
<tr>
<td>0x0000002E4</td>
<td>2601</td>
<td>MOVS r6, #0x01</td>
</tr>
</tbody>
</table>

Table 2: The two MOVS instruction

It can be seen clearly that for both two instructions, the third and forth numbers in the machine code are the same as the value of the immediate value (2700 and #0x00; 2601 and #0x21).

In detail, we take the machine code “0x2700” as an example: The last byte “00” represents the value of the number to be moved into the register 7.

Overall, for Q2 and Q3, a comprehensive explanation of machine code is like this: We take machine code “2601” for example, whose mnemonic is “MOCS r6, #0x01”. Here, the “01” corresponds to “#0x01”, “6” corresponds to “r6”, and “2” corresponds to “MOVS”. Therefore, it is apparent that the second number of machine code represents the register used while the last byte of the machine code represents the value to be moved.
8 Answer to Q4[4 marks]

- **Answer**
  
  The original content of R4 is **0x00000314**, and the resulting content of R4 after executing the instruction “ADDS r4, r2, #7” is **0x00000015**.

- **Explanation**
  
  The contents of R4 before and after executing the related instruction are presented in Figure 6.

---

Figure 6: The content change of R4
The mnemonic of the instruction concerning the R4 is “ADDS r4, r2, #7”, which means adding the value 7 to the value stored in R2 and then load this result into R4. It can be clearly seen in Figure 6 that before executing this statement, the content of R4 is 0x00000314, and the content in R2 is 0 000000E.

However, after executing this statement, the value of R4 is the loaded result of the addition of value in R2 and the value of immediate #7, which makes the content of R4 be 0 0000015.

9 Answer to Q5[4 marks]

• Answer
  – The order of the registers in the mnemonics is Important.

• Explanation
  – This is because when the mnemonic statements are converted into binary format, they represent execution type, used register or value etc. in a set order, which means that different order of mnemonic corresponds to different executions.

  – In detail, if we take the instruction “SUBS r2, r3, r2” as an example, this means the value in register 2 is subtracted from the value in register 3 and then the subtraction result is placed in register 2. However if the order of r3 and r2 is exchanged, it means the value in register 3 is subtracted from the value in register 2 and the subtraction result is placed in register 2. They are two different instructions.

  – However, for some other instructions such as ADDS or MULS, the order of registers in mnemonics is not important since A+/× B is the same as B+/× A. Therefore it can be concluded that for some instructions, the result might not be effected if some of the mnemonics are exchanged however for many other instructions, the order cannot be changed at all. It is reasonable to say that the order of mnemonics is important in a large scale.

10 Answer to Q6[4 marks]

• Answer
  – The values are as expected.

  – When the program is stepped to address 0x000001EA, the contents of register R2 and R3 are 0x00000064 and 0x0000000A respectively. The result content of R2 when the program is stepped to address 0x000001F6 is 0xFFFFFA6, which is the 2s complement of 0x0000005A. This is the same as expected since for the mnemonic “SUBS r2, r3, r2” the result obtained is -90 (10-100=-90) in decimal form (0x0000005A in hexadecimal).
• **Explanation**

  – The obtained result of R2 at that step is presented in Figure 7.

![Figure 7: The result content of R2](image)

  – As the program steps from address 0x0000001EA to address 0x000001F6, the related information and contents are presented in Table 3.

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Mnemonics</th>
<th>Original Content</th>
<th>Result Content</th>
<th>Decimal Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000001EA</td>
<td>ADDS r4, r2, #7</td>
<td>R4 0x00000314</td>
<td>R4 0x0000006B</td>
<td>107</td>
</tr>
<tr>
<td>0x0000001EC</td>
<td>ADDS r5, r2, r3</td>
<td>R5 0x1FFFE000</td>
<td>R5 0x0000006E</td>
<td>110</td>
</tr>
<tr>
<td>0x0000001EE</td>
<td>ADDS r6, r3, r2</td>
<td>R6 0x00000000</td>
<td>R6 0x0000006E</td>
<td>110</td>
</tr>
<tr>
<td>0x0000001F0</td>
<td>SUBS r0, r2, #1</td>
<td>R0 0x1FFFE060</td>
<td>R0 0x00000063</td>
<td>99</td>
</tr>
<tr>
<td>0x0000001F2</td>
<td>SUBS r1, r2, r3</td>
<td>R1 0x1FFFE060</td>
<td>R1 0x0000005A</td>
<td>90</td>
</tr>
<tr>
<td>0x0000001F4</td>
<td>SUBS r2, r3, r2</td>
<td>R2 0x00000064</td>
<td>R2 0xFFFFFFFFA6</td>
<td>-90</td>
</tr>
</tbody>
</table>

**Table 3: The step by step execution**

  – It can be found that although there are five more steps executed before the “SUBS r2, r3, r2” is executed, the five steps do not influence the values in r2 and r3. Therefore, we only focus on the substraction step, which is value in r3 minus the value in r2.

  – The obtained substraction result is -90 in decimal form since the calculation is A-64 whose decimal equivalent equation is 10-100. The absolute value of -90 has a hexadecimal form 0x0000005A. The 1s complement is then obtained by converting each digit which is 0xFFFFFFFFA5. Then we add 1 to the 1’s complement so that we obtain the 2s complement format for -90 is 0xFFFFFFFFA6, which is same as the result presented in Figure 7 above.
11 Answer to Q7 [4 marks]

- **Answer**
  
  - The values are as expected.
  
  - When the program is stepped to address 0x000001EA, the contents of register R2 and R3 are 0x00000020 and 0x0000000E respectively. The result content of R2 at step 0x000001F6 is 0xFFFFFFFEE, which is the 2s complement of 0x00000012. This is the same as expected since in decimal form the result is 14-32=-18 (hexadecimal form of 18 is 0x00000012).

- **Explanation**
  
  - The obtained result of R2 at that step is presented in Figure 7.
As discussed in previous section, there is no need to focus on the previous five steps, therefore there is no detailed step by step execution Table listed.

After the step for address 0x0000001F4, the obtained result is -18 in decimal form since in decimal, the calculation of E-20 is equivalent to 14-32=-18.

The absolute value of -18 has a hexadecimal form 0x00000012. The 1s complement is then obtained by converting each digit which is 0xFFFFFFED. Then we add 1 to the 1’s complement so that we obtain the 2s complement format for -18 is 0xFFFFFFFEE, which is same as the result presented in Figure 9 above.

12 Answer to Q8 [4 marks]

• Answer

– The performed logic function is **AND**.

• Explanation

– It can be seen from Figure 10 below that the mnemonic when reaches address 0x000001FE, “ANDS r4, r4, r3”, and since the previous step is “MOV r4, r2” therefore the original r4 (the second r4 in ANDS statement) has the same content as r2. The contents of R2 and R3 is 0x00000011 and 0x00000101 before ANDS statement, while after the statement is executed, the content of R4 at address is 0x00000001, which is the result for the execute R2 AND R3. The truth table for the inputs R4(R2) and R3 is presented in Table 4.

![Figure 10: Related contents after “ANDS r4, r3” is executed](image)

<table>
<thead>
<tr>
<th>Register</th>
<th>4th digit</th>
<th>3rd digit</th>
<th>2nd digit</th>
<th>1st digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>R4(R2)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>R3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>R4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4: The truth table for “ANDS r4, r3”
The truth table of AND logic function is shown in Table 5. \( Z = AB \), where \( Z \) can output HIGH only when both of \( A \) and \( B \) are HIGH. This AND logic is the same as the truth table for the statement as shown in Table 4. Therefore the performed logic is AND.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5: The truth table for AND

13 Answer to Q9[4 marks]

- **Answer**
  - For the two registers on the two side of the logic symbol, it is **not important**. This is because apart from the BICS logic function, there are OR logic function, AND logic function and EOR logic function. Based on the logic principles, it is apparent that for these logic functions, the order will not influence the result.
  - However, if we consider the register for storing the result as well, this register should not be messed up with the two registers on the two sides of the logic symbol due to the fact as we explained for Q5.

- **Explanation**
  - AND logic function, OR logic function and XOR logic function conduct the logic function based on the combination of two value for each digit regardless of their order. Therefore, the order does not affect the operation.
14 **Answer to Q10[4 marks]**

- **Answer**
  - The program counter will **return to address 0 00000212**.

- **Explanation**
  - After the code was executed at address of 0x00000214 in register 15, the branch will return to 0x00000212 as can be seen in figure 11.

![Figure 11: Related contents after “ANDS r4, r3” is executed](image)

15 **Answer to Q11[5 marks]**

- **Answer**
  - The value of register 2 became **0x00000005**. It actually perform as a counter.

- **Explanation**
  - Register R2 is working as a counter: Once the instruction at 0x00000214 is executed, the value will be added 1. If the instructions at address 0x00000214 is executed for 5 times and the original content of register R2 is 0x00000000, the final content of register R2 would be 0x00000005, since each time the content of R2 will increase 1.
16 Answer to Q12[4 marks]

• Answer
  – The count system has a period. The counted time has a ratio to the period. This is why after 40s, the value held by R2 is approximately twice the value after the initial 20 seconds.

• Explanation
  – The first 20 second is 0x1075C47C, and the content of R2 after the another more 20 second is 0x211E8343. The twice value of 0x1075C47C is 0x20EB88F8, which is approximately equal to the 0x211E8343, as can be seen in Figure 12.

  Figure 12: result after 20s and 40s

– The loop is executed in a certain time cycle. The code in loop is repeat. For the first 20 seconds, the loop is executed for several times with the r2 serving as a counter, while for the later 20 seconds (that is 20s to 40s), the loop will be executed for the same number of times. Therefore, the value stored in r2 should be approximately twice the value after the initial 20 seconds.
17 The graph of Section 11 (using MS Excel or MATLAB) [6 marks]

• Screenshots

![Figure 13: The time versus number of instructions](image.png)

• Comment/Explanation

– For the three loops, these three points are roughly in a straight line. As the number of instruction increases, the time taken increases proportionally as well with a certain direct ratio.

– As for the calculation method of these values used to draw points, the principle is that since the counting time is so short that we cannot directly measure it using timer, we record the counting time in a certain period which is long enough to record using a timer. Therefore, we can obtain the single time interval by dividing the total time by the number counted.
18 Answer to Q13[4 marks]

- **Answer**
  - 22ns \((2.2 \times 10^{-8}\text{s})\)

- **Explanation**
  - The add instruction takes one clock cycle to execute whereas the branch instruction takes more than this. Therefore, we firstly take the decimal form of the value stored in r2; then we use this value to subtract it from the measured time.

<table>
<thead>
<tr>
<th>Number of instructions</th>
<th>Time Taken (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>72</td>
</tr>
<tr>
<td>3</td>
<td>94</td>
</tr>
<tr>
<td>4</td>
<td>120</td>
</tr>
</tbody>
</table>

Table 6: The values used for Figure 13

- The time taken by one clock cycle is worked out using the previously listed numbers as shown in Table 6 using equation below:

\[
94 - 72 = 22\text{ns}.
\]  \hspace{1cm} (2)

19 Answer to Q14[3 marks]

- **Answer**
  - 3 clock cycles are required for the branch instruction.

- **Explanation**
  - The clock cycle can be worked using equation as below:

\[
\frac{72 - 22}{22} \approx 2.273
\]  \hspace{1cm} (3)

The extra 0.273 clock circle should be considered as one clock circle practically.
20 Screenshot of the result of Section 12 [3 marks]

- Screenshots

Figure 14: A1:0x0000022C

Figure 15: A2:0x0000022E
Figure 18: A5:0x00000234

Figure 19: A6:0x00000234
Figure 20: B1:0x000022C

Figure 21: B2:0x000022E
<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core: R0</td>
<td>0x00000000</td>
</tr>
<tr>
<td>R1</td>
<td>0x00000000</td>
</tr>
<tr>
<td>R2</td>
<td>0x00000000</td>
</tr>
<tr>
<td>R3</td>
<td>0x00000000</td>
</tr>
<tr>
<td>R4</td>
<td>0x00000000</td>
</tr>
<tr>
<td>R5</td>
<td>0x00000000</td>
</tr>
<tr>
<td>R6</td>
<td>0x00000000</td>
</tr>
<tr>
<td>R7</td>
<td>0x00000000</td>
</tr>
<tr>
<td>R8</td>
<td>0x00000000</td>
</tr>
<tr>
<td>R9</td>
<td>0x00000000</td>
</tr>
<tr>
<td>R10</td>
<td>0x00000000</td>
</tr>
<tr>
<td>R11</td>
<td>0x00000000</td>
</tr>
<tr>
<td>R12</td>
<td>0x00000000</td>
</tr>
<tr>
<td>R13</td>
<td>0x00000000</td>
</tr>
<tr>
<td>R14</td>
<td>0x00000000</td>
</tr>
<tr>
<td>R15</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

Disassembly:
```assembly
0x00000026 E7FB B 0x00000020
; clear r3 and
0x00000028 2200 MOV r2, #0
0x00000028 2200 MOV r2, #0
; set value in
65: MOV r5, #1
66: MOV r5, #1
67: MOV r4, r2
68: MOV r4, r2
; same as previous instruction
70: ADDS r5, r2, r3
71: ADDS r5, r2, r3
72: SUBS r6, r2, r3
73: SUBS r6, r2, r3
74: ; Conditional branching
```
Figure 24: B5:0x00000234

Figure 25: C1:0x0000022C
Figure 26: C2:0x0000022E

Figure 27: C3:0x00000230
Figure 28: C4:0x00000232

Figure 29: C5:0x00000234
Comment/Explanation

- As described by the script materials, this processor we used has four “flags” which provides information about the action state of the previous instruction. “Z” is zero flag which is set only when the result is zero. “N” is negative flag. “C” is carry flag, which is set when addition calculation result has a carry. “V” is overflow flag. Only those instruction with a mnemonic ending with an S (which means that the registers are restricted to the low registers in range r0 to r7) actually affects the flags.

- As can be seen in Figures, these figures can be separated into three sections (A, B and C). The first section is basically to first reset the program counter to 0x00000228 and then run through the program to 0x00000234 step by step. When the result is a negative number the flag “N” is set. When the most significant binary digit is 1, which means “N” is set. In detail, Figure A1 to Figure A6 presents the results when R2 is 0x0000022E. It can be seen that for address 0x00000228, MOVS r2, #0 00, the result is zero so that the zero flag “Z” is set. For address 0x0000022A, MOVS r3, #0 01, no flags is set since the result is not zero, carry, or negative. For the address 0x0000022C, MOV r4, r2, since the instruction is MOV without an “S”, no flags have changes. For address 0x0000022E, MOVS r4, r2, since the value in R2 is 0x00, so move the 0x00 into R4 makes the result to be zero, therefore the zero flag “Z” is set. For the address 0x00000230, ADDS r5, r2, r3, the result stored into R5 is #0x01, no flags is set. For the address 0x00000232, SUBS r6, r2, r3, the result is negative number -110, so the flag “N” is set, and there has a borrow so the flag C is clear.

- The second section is to change the value held in r2 to 0xFFFFFFFF and run from address 0x0000022A step by step as shown in Figure B1 to Figure B5. When the address is 0x00000230, the result is a negative number therefore the flag “N” is set. When the address is 0x00000232, the result is 0 so the “N” is clear, “Z” is set and “C” is clear. When the address is 0x00000234, the result is negative value so “N” is set and Z is clear.

- The final section is to change the value held in R2 to 0x7FFFFFFF and run the program from address 0x0000022A step by step. For the address 0x00000232, the results is negative and overflow therefore the flag “N” is set and “V” is set. For address 0x00000234, the result is positive with a carry so the “N” is clear and “C” is set. In detail, Figure C1 to Figure C5 presents the results when set R2 as 0x7FFFFFFF, flags “C” remains unchanged for MOV operations. For address 0x0000022E, MOV r4, r2, since the content of R2 is 0x7FFFFFFF, the 0x7FFFFFFF is moved into R4, but without changing the flags. For the address 0x00000230, ADDS r5, r2, r3, the result stored into R5 is negative #0x80000000, so flag “N” and flag “V” are set.

21 Answer to Q15[3 marks]

- Answer

- BEQ, BCS, BPL and BVC condition branch instructions are executed. This is because Z was set; C was set; N was clear; and V is clear, and when it jump to the next stage it means the condition is satisfied.
- It can be seen in Figure 30 that for BEQ next1, it is branch to next1 when the zero flag “Z” is set; When Z=1, the condition branch instruction BEQ is executed and therefore it jumps to next1.

Figure 30: BEQ branch instruction

- It can be seen in Figure 31 that for BNE next2, it is branch to next2 when the zero flag “Z” is clear; When Z=1, the condition branch is not executed and therefore it sequentially goes to the next step as shown in Figure 32.

Figure 31: BNE branch instruction is not executed
It can be seen in Figure 32 and Figure 33 that for mnemonic BCS next3, it is branch to next3 if carry flag “C” is set; When C=1, so the condition branch instruction BCS is executed, and it jump to next3 as shown in Figure 33.

Similarly, following this method to observe the program step by step in detail, it can be found that overall, the branch instruction will be executed correspondingly when the state of flags is set or clear.
Answer to Q16 [3 marks]

- Answer
  - The conditional branch instructions have executed as expected. This is because by judging the conditions, it follows the principles that when condition is true then jump to the corresponding next stage, and while when condition is false then goes to next step.

- Explanation
  - By observing the program conduction step by step, it can be found that BNE, BCC, BMI, BVS were executed while BEQ, BCS, BPL, BVC were not executed. When the BNE function was executed since the zero flag “Z” is clear. BCC function was executed since carry flag “C” is clear. BMI function was executed when negative flag “N” is set. BVS function was executed since V is set.

  - In contract, BEQ, BCS, BPL, BVC was not executed since the zero flag “Z” is clear, “C” is set, “N” is clear and “V” is set. Initially, when there is no branch executed yet, the values is shown in Figure 20 that “N” is set, “V” is set, “Z” is clear and “C” is clear, while after running the program step by step, it was observed that the actual conductance method agrees with what we expected as discussed above. Some typical steps are presented below:

![Figure 34: BEQ branch instruction](image-url)
Figure 35: BNE branch instruction

Figure 36: BCS branch instruction
Figure 37: BCC branch instruction

Figure 38: BMI branch instruction
Figure 39: BPL branch instruction

Figure 40: BVS branch instruction
Figure 41: BVC branch instruction