마이크로프로세서응용실험 LAB04 결과보고서

Data Processing

20181536 엄석훈

1. 목적

● 논리(logical), 연산(arithmetic), 곱셈/나눗셈, bits 단위의 데이터처리 명령어들의 종류와 동작에 대해 이해한다.

● Flag들의 종류와 각각이 set 되었을 경우의 의미를 이해한다.

● Bits단위 데이터처리의 예를 일부 명령어들의 동작을 통해 확인한다.

● Sign extension의 의미와 필요성에 대해 구체적으로 이해한다.

2. 이론

1) Number system

컴퓨터에서는 0과 1이라는 2진수만을 사용할 수 있다. 하지만 컴퓨터를 사용하는 사용자가 2진법만으로 표현된 숫자를 보기는 매우 힘들고 그 숫자가 나열된 길이 또한 매우 길 것이다. 따라서 우리가 사용하는 10진수으로 나타내도 좋지만 10진수과 2진수 사이에는 한눈에 들어오는 상관관계가 없다. 따라서 보통 2진수을 3bit를 하나의 숫자로 표현하는 8진수이나 4bit를 하나의 숫자로 표현하는 16진수로 변환하여 사용한다.

또한 음수를 나타내기 위해서 가장 간단하게 sign bit만 사용하거나 모든 비트를 토글하는 1's complement를 사용할 수 있지만 연산의 편의성을 위해서 2's complement라는 방법을 사용한다. 원하는 숫자의 2's complement를 구하기 위해서는 원래의 값에서 모든 비트를 토글하고 1을 더해주면 된다. 예를 들어 6은 0000_0110인데 이의 2's complement는 1111_1001로 바꾸고 1을 더한 1111_1010이 된다.

마지막으로 텍스트를 표현하기 위해서 ASCII코드를 표준으로 사용한다. 물론 최근에는 다양한 나라의 언어까지 모두 표현하기 위해서 UTF-8과 같은 코드를 사용한다. 아스키코드는 표준으로 정해져 있기 때문에 사용할 때 마다 알맞은 숫자로 바꿔 사용해주어야 한다. 예를 들어 0을 메모리에 저장하면 0x00이 저장되는데 비해 문자 '0'을 저장하면 아스키 코드표에 따라서 0x30이 저장된다.

2) Flags

일부 데이터를 처리하는 명령어를 수행의 결과로 flag값이 변경된다. 이러한 flag가 필요한 이유는 이어지는 명령어에서 flag값을 이용해서 명령어가 수행되는 일이 많기 때문이다. 이러한 flag는 PSR이라는 특수 레지스터에 저장된다. PSR 레지스터의 비트 할당은 아래 [그림 1]에서 볼 수 있다.

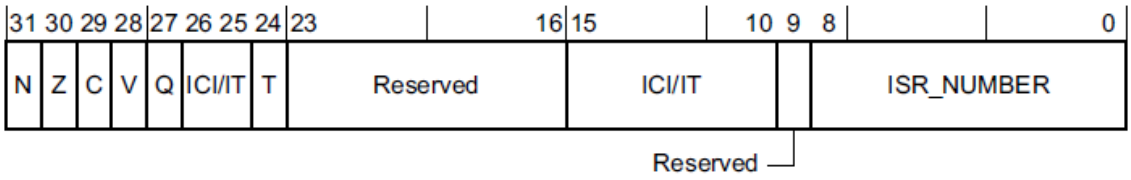

그림 1 PSR 레지스터의 비트 할당

각 flag의 의미는 다음과 같다. 먼저 N flag는 결과가 음수인지 확인해주는 flag이다. MSB비트에 따라서 N값이 정해진다. 다음으로 C flag는 덧셈 또는 shift 등의 연산에서 캐리가 발생하는 경우에 flag가 1이 된다. 또한 Cortex-m3에서는 borrow flag도 carry flag와 공간을 같이 공유하기 때문에 뺄셈에서 borrow가 발생하지 않았을 때 또한 flag가 1이 된다. 다음으로 V flag는 연산에서 overflow가 발생하였을 때 flag가 1이 된다. 예를 들어 두 음수를 더해서 양수가 나오면 overflow가 발생하고 두 양수를 더해서 음수가 나오면 overflow가 발생한다. 다음으로 Z flag는 레지스터에 저장된 결과가 0일 때 1로 설정된다. 마지막으로 Q flag는 명령어 수행 시 saturation 여부를 확인해주는 flag이다. 단 Q flag는 한번 1로 설정되면 MRS명령어만 읽을 수 있고MSR을 통해서만 변경해줄 수 있다. Saturation이 발생하는 경우는 큰 bit를 가진 숫자를 작은 bit의 숫자로 변화하는 과정에서 표현할 수 있는 범위를 벗어나는 경우 saturation이 일어나 Q flag를 1로 만들어 준다.

3) Logical 명령어

op{S}{cond} {Rd,} Rn, Operand2

의 꼴로 명령어를 작성할 수 있다. 위의 op에는 and 연산을 해주는 AND와 or 연산을 해주는 ORR, exclusive or을 수행해주는 EOR, operand2의 not과 and 연산을 해주는 BIC, oprand2의 not과 or 연산을 해주는 ORN 총 5가지가 존재한다. 그리고 {S}가 포함되면 연산 후에 flag값을 갱신해 준다. 그리고 {cond}는 flag에 따라 명령어를 수행할지 말지를 결정하는 condition체크 부분이다. 그리고 Rd는 명령어 수행 결과를 저장할 레지스터이고 Rn은 첫번째 데이터를 가지고 있는 레지스터이고 operand2는 2번째 데이터를 가지고 있다.

4) Shift and rotation 명령어

op{S} {cond} Rd, Rm, Rs 또는 op{S} {cond} Rd, Rm, #n

RRX{S} {cond} Rd, Rm

의 꼴로 명령어를 작성할 수 있다. 위의 op에는 ASR, LSL, LSR, ROR이 들어갈 수 있다. ASR은 arithmetic shift right로 MSB가 1이면 shift하면서 MSB를 1로 채우고 0이면 0을 채우는 shift 명령어이다. 다음으로 LSL과 LSR은 logical shift left와 right로 각각 좌, 우로 shift하는 명령어이다. 마지막으로 ROR은 rotate right로 오른쪽으로 밀면서 밀린 데이터가 왼쪽 끝에서 들어오는 명령어이다. {S}는 logical 명령어와 같이 연산 후에 flag값을 갱신할지 결정하는 부분이고 {cond}는 flag에 따라 명령어를 수행할지 말지를 결정하는 condition체크 부분이다. Rd는 연산 결과를 저장할 레지스터, Rm은 연산을 수행할 레지스터, Rs또는 #n은 shift할 길이이다. 마지막으로 RRX명령어는 Rm의 데이터를 오른쪽으로 1칸 shift해서 Rd에 저장하는 명령어이다.

5) Arithmetic 명령어

op{S} {cond} {Rd, } Rn, Operand2

의 꼴로 명령어를 작성할 수 있다. 첫번째 명령어의 op에는 ADD, ADC, SUB, SBC, RSB가 들어갈 수 있다. ADD는 말 그대로 더해주고, SUB는 빼주고, ADC와 SBC는 캐리를 포함해서 더하고 빼주는 명령어이다. 그리고 RSB는 거꾸로 Operand2에서 Rn을 빼는 명령어이다. {S}와 {cond}는 위의

명령어 설명과 동일한 역할을 하고 Rd은 연산 결과를 저장할 레지스터, Rn과 operand2는 연산을 수행할 데이터들이 포함된다.

6) Multiply and divide 명령어

MUL{S} {cond} {Rd, } Rn, Rm : Rn과 Rm을 곱해 Rd에 저장한다.

MLA{cond} Rd, Rn, Rm, Ra : Rn과 Rm을 곱하고 Ra를 더해 Rd에 저장한다.

MLS{cond} Rd, Rn, Rm, Ra : Rn과 Rm을 곱하고 Ra를 빼서 Rd에 저장한다.

UMULL{cond} RdLo, RdHi, Rn, Rm : Rn과 Rm을 곱하고 RdLo에 하위 비트를 RdHi에 상위 비트를 저장한다. (부호 고려 X)

SMULL{cond} RdLo, RdHi, Rn, Rm : Rn과 Rm을 곱하고 RdLo에 하위 비트를 RdHi에 상위 비트를 저장한다. (부호 고려)

UMLAL{cond} RdLo, RdHi, Rn, Rm : Rn과 Rm을 곱하고 원래 레지스터 값에 더해서 RdLo에 하위 비트를 RdHi에 상위 비트를 저장한다. (부호 고려 X)

SMLAL{cond} RdLo, RdHi, Rn, Rm : Rn과 Rm을 곱하고 원래 레지스터 값에 더해서 RdLo에 하위 비트를 RdHi에 상위 비트를 저장한다. (부호 고려)

SDIV{cond} {Rd, } Rn, Rm : Rn 나누기 Rm의 몫을 Rd에 저장한다. (부호 고려 X)

UDIV{cond} {Rd, } Rn, Rm : Rn 나누기 Rm의 몫을 Rd에 저장한다. (부호고려)

7) Bitfield 명령어

BFC{cond} Rd, #lsb, #width : Rd에서 lsb번째 bit부터 width만큼 0으로 reset한다.

BFI{cond} Rd, Rn, #lsb, #width : Rd에서 lsb번째 bit부터 width만큼 공간에 Rn의 bit 0부터 width 만큼 복사해 붙여넣는다.

8) Sign extension

SBFX{cond} Rd, Rn, #lsb, #width : Rn의 lsb째 bit부터 width만큼 부호를 고려하여 Rd에 저장한다.

UBFX{cond} Rd, Rn, #lsb, #width : Rn의 lsb째 bit부터 width만큼 Rd에 저장한다.

SXTB{cond} {Rd, } Rm {, ROR #n}, SXTH{cond} {Rd, } Rm {, ROR #n} : Signed 8-bit또는 16-bit를 Rm에서 #n만큼 오른쪽으로 rotate해서 Rd에 저장한다.

UXTB{cond} {Rd, } Rm {, ROR #n}, **UXTH{cond} {Rd, } Rm {, ROR #n}** : Unsigned 8-bit 또는 16-bit를 Rm에서 #n만큼 오른쪽으로 rotate해서 Rd에 저장한다.

3. 실험 과정

1) 실험 1

STEP 1: Program 4.1을 이용하여 프로젝트를 생성한다.

```
1        area lab4_1,code
2        entry
3   __main proc
4        export __main [weak]
5   strat ldrb r0,data1
6        ldrb r1,data2
7        ldrb r3,mask1
8        ldrb r4,mask2
9        ands r5,r0,r3
10  ;    tst r0,r3
11       orr r0,r0,#0x08
12       and r5,r0,r4
13  ;
14       eor r6,r0,r3
15  ;    eors r6,r0,#2_10101111
16       teq r0,#2_10101111
17       eor r6,r0,r0
18  ;
19       ldr r7,=ramdata
20       strb r0,[r7,#0x6]
21  ;
22  ;
23       ldr r0,data3
24       bic r6,r0,#0x0f0
25  ;
26       ldr r5,data3
27       ssat r6,#12,r5
28       b .
29       endp
30       align
31  data1 dcb 2_10110011
32  data2 dcb 2_10100101
33       align
34  data3 dcd 0x00011234
35  mask1 dcb 2_00010000
36  mask2 dcb 2_11110111
37       align
38       area lab4data,data
39  ramdata space 16
40       end
```

그림 2 직접 작성한 lab4_1 코드

STEP 2: µVision를 이용해 simulation을 수행하기 전에 Register window에서 xPSR(program status register)을 통해 flag들의 변화를 살펴볼 수 있도록 +로 표시된 부분을 click해서 그림 4.7과 같이 표시되도록 한다. 그림에서 N, Z, C, V, Q로 표시된 부분이 4.3절에서 언급하는 flag들을 의미한다.

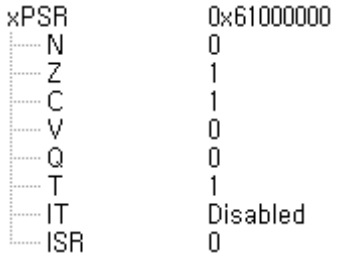

그림 3 Line 5 실행 전 PSR 레지스터 값

STEP 3: Line 9 명령어까지 수행한 후 r5에 저장된 내용과 flag들이 표시하는 상태가 일치하는지 확인한다. 명령어를 ands에서 and로 변경한 후 이 과정을 반복하여 명령어에 추가된 suffix s가 미치는 영향을 확인한다. 또한 line 9를 주석처리하고 line 10의 추석처리를 삭제한 후 이 과정(즉, 처음부터 line 9까지 수행)을 반복한다. A-4 페이지와 A-35 페이지를 통해 두 명령어의 차이점을 확인해 본다.

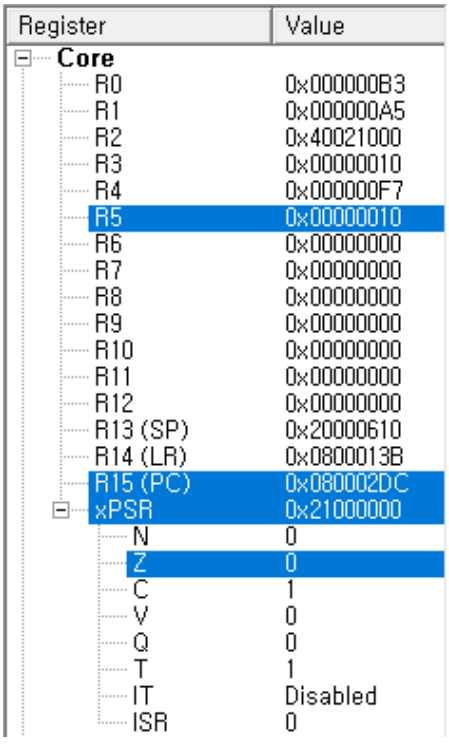

그림 4 Line 9 수행 후 레지스터 값

Line 5부터 line 8까지의 수행을 통해 data1, data2, mask1, mask2를 레지스터 r0, r1, r2, r3에 저장하였다. 그리고 line 9는 r0와 r3의 값을 and해서 나온 2_00010000 즉 16진수로 0x10을 r5에 저장해주었다. 그리고 ands의 명령어의 결과가 0이 아니기 때문에 zero flag는 0이 되었고 나머지 flag의 결과는 바꾸지 않았다. 그리고 명령어를 ands에서 and로 바꾼 결과 flag가 변동하지 않은 것을 아래 [그림 5]를 통해서 확인할 수 있다. 이를 통해서 명령어 뒤에 붙은 suffix s는 flag를 업데이트 할지 말지 결정하는 중요한 요소임을 확인할 수 있었다.

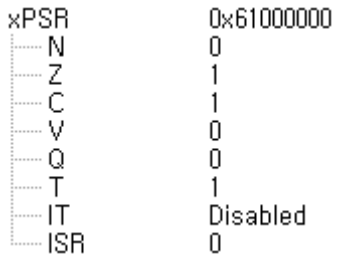

그림 5 Line 9 수정 후 레지스터 값

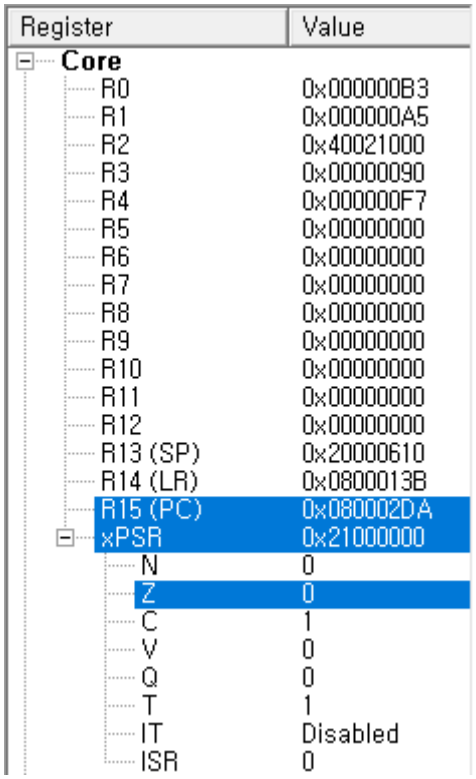

그림 6 Line 10으로 변경 후 레지스터 값

그리고 line 9의 명령어를 주석처리 하고 line 10을 수행한 결과 저장된 결과 [그림 6]과 [그림 4]에 저장된 PSR의 flag값의 결과는 동일하게 저장되었고 ands 명령어와 다르게 tst 명령어는 다른 일반 레지스터의 값에는 변화를 주지 않았다. 즉 tst 명령어는 두 레지스터의 값을 and해서 연산

결과는 버리고 그 결과에 의해 만들어진 flag값만 업데이트 해주는 것을 확인할 수 있었다.

STEP 4: AND 명령어를 통해 데이터의 특정 bit(s)를 0으로 만드는 방법을 line 12 명령어의 수행을 통해 확인한다.

```
    11:        orr r0,r0,#0x08
0x080002DC F0400008  ORR       r0,r0,#0x08
    12:        and r5,r0,r4
    13: ;
0x080002E0 EA000504  AND       r5,r0,r4
```

그림 7 Line 11, line 12 명령어

R0	0x000000BB
R1	0x000000A5
R2	0x40021000
R3	0x00000090
R4	0x000000F7
R5	0x000000B3

그림 8 Line 12 수행 후 레지스터 값

Line 11 명령 수행 전 r0에는 0xB3가 저장되어 있는데 0x08과 or을 해서 다시 r0에 저장을 하면 r0에는 0xBB가 저장된다. 그리고 r4에는 mask2값인 1111 0111 즉 0xF7이 들어있다. 4번째 비트만 0인 mask이다. 그리고 r0와 r4를 and하게 되면 r0는 1011 1011였음으로 4번째 bit만 0과 and를 해서 0이 되고 나머지 bit는 1과 and를 하였음으로 원래의 값이 유지된다. 따라서 r0에는 1011 0011 즉 0xB3가 저장되고 [그림 8]에서도 r5에 0xB3가 저장된 것을 확인할 수 있다. 이를 통해 데이터의 특정 bit를 0으로 만드는 masking기법을 확인할 수 있었다.

STEP 5: 그림 4.4(d-f)에서 소개된 exclusive OR의 동작과 활용을 lines 14-17을 수행하며 확인한다. 이 과정에서 line 15와 16은 STEP 3과 같이 주석처리를 바꾸어 수행하면서 두 명령어의 차이를 확인한다.

```
    14:        eor r6,r0,r3
    15: ;      eors r6,r0,#2_10101111
0x080002E4 EA800603  EOR       r6,r0,r3
```

그림 9 Line 14 명령어

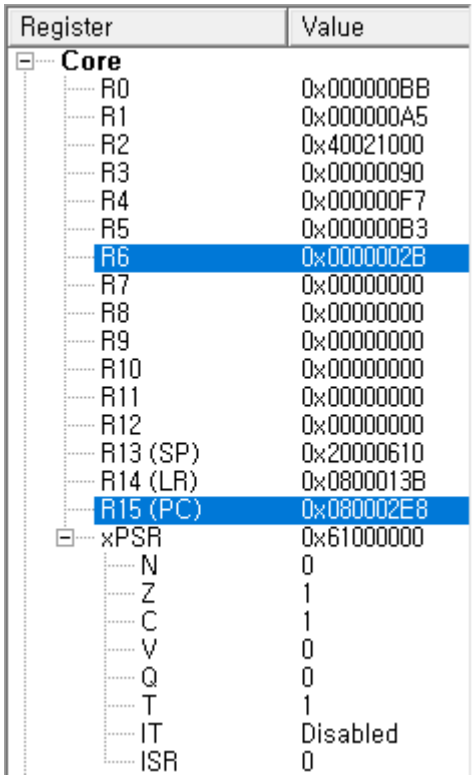

그림 10 Line 14 수행 후 레지스터 값

Line 14 명령어는 [그림 9]에서 보이듯 eor r6,r0,r3로 r0와 r3를 xor해서 r6에 저장하는 명령어이다. Line 14를 수행하기 전 레지스터 값은 위의 [그림 8]에서 확인 가능하다. 명령어 수행 시 r0의 1011 1011과 r3의 1001 0000이 xor하게 교재 그림 4.4의 (d)에서와 같이 5번째와 8번째 bit만 toggle되게 되어 0010 1011이 되어 r6에 2B가 저장된 것을 확인할 수 있다.

다음으로 Line 15 명령어는 아래 [그림 11]에서 보이듯 eros r6,r0,#2_10101111로 r0의 0xBB와 1010 1111을 xor하는데 이는 교재 그림 4.4의 (e)와 같이 두 데이터를 비교하기 위해 수행하였으며 그 결과 두 데이터에서 다른 부분만 1이 남아 0001 0100이 r6에 저장되고 명령어 뒤에 s가 붙어있어 zero flag또한 1로 업데이트 되는 것을 아래 [그림 12]에서 확인할 수 있다.

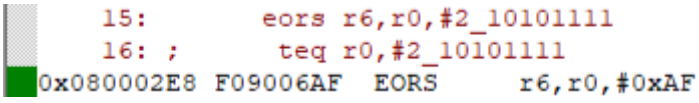

그림 11 Line 15 명령어

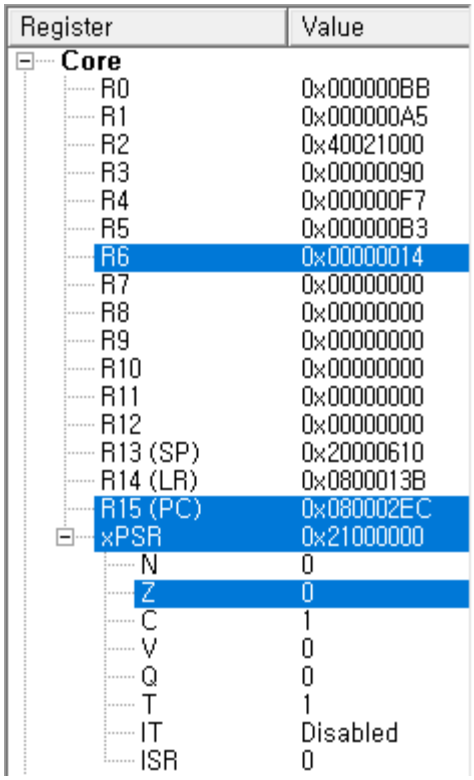

그림 12 Line 15 명령어 수행 후 레지스터 값

```
    16:         teq r0,#2_10101111
0x080002E8 F0900FAF  TEQ       r0,#0xAF
```

그림 13 Line 16 명령어

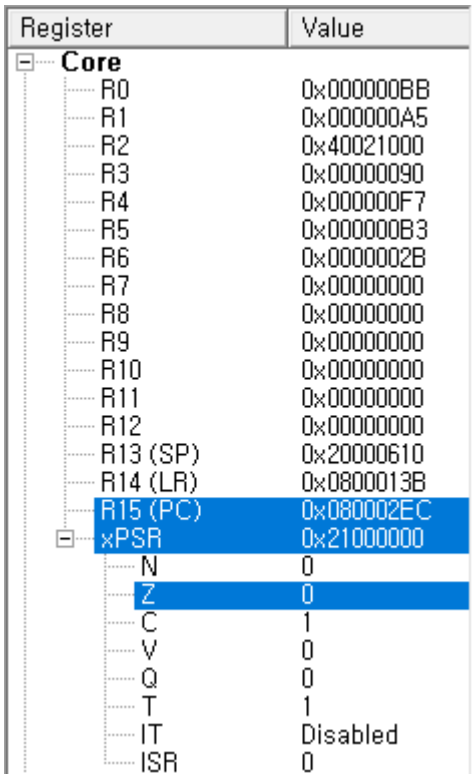

그림 14 Line 16 수행 후 레지스터 값

다음으로 Line 16 명령어는 [그림 13]에서 볼 수 있듯이 teq r0,#2_10101111로 이전의 Line 15명령어와 비슷한 역할을 수행하는 명령어이다. eors명령어와 같은 연산을 진행하지만 연산의 결과는 다른 레지스터에 저장하지 않고 flag에만 변화를 주는 명령어이다. 따라서 위의 [그림 14]에서 볼 수 있듯이 명령어 수행 전 레지스터 값인 [그림 10]과 비교하였을 때 flag값과 당연한 이야기지만 pc의 값만 다른 것을 확인할 수 있다.

```
    17:         eor r6,r0,r0
    18: ;
0x080002EC EA800600  EOR      r6,r0,r0
```

그림 15 Line 17 명령어

Register	Value
Core	
R0	0x000000BB
R1	0x000000A5
R2	0x40021000
R3	0x00000090
R4	0x000000F7
R5	0x000000B3
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000610
R14 (LR)	0x0800013B
R15 (PC)	0x080002F0
xPSR	0x21000000
N	0
Z	0
C	1
V	0
Q	0
T	1
IT	Disabled
ISR	0

그림 16 Line 17 명령어 수행 후 레지스터 값

마지막으로 line 17 명령어는 위의 [그림 15]에서 볼 수 있듯이 eor r6,r0,r0이다. 이는 교재 그림 4.4의 (f)와 같은 역할을 하는 동작이다. 같은 값을 xor하면 항상 결과는 0이 나오는데 line 17에서는 r0와 r0의 값을 xor해서 r6에 저장해 주었음으로 r6에는 0x00이 저장된 것을 확인할 수 있다. 이를 통해서 eor명령어를 통해서 수행할 수 있는 다양한 기능을 살펴보았다.

STEP 6: Lines 19-20의 수행을 통해 r0의 내용을 RAM에 저장한다. 메모리에 저장한 이 데이터에 대해 line 12와 동일한 동작(즉, mask와 AND 동작을 이용한 특정 bit의 reset)을 수행할 수 있도록 codes를 구성해서 lines 21-22 위치에 추가한다. 또한, 이 추가한 내용에 해당하는 동작을 앞장에서 설명한 bit-banding 동작을 통해 구현해 본다.

```
    19:         ldr r7,=ramdata
0x080002F0 4F07       LDR      r7,[pc,#28]   ; @0x08000310
    20:         strb r0,[r7,#0x6]
    21: ;
    22: ;
0x080002F2 71B8       STRB     r0,[r7,#0x06]
```

그림 17 Line 19, line 20 명령어

R0	0x000000BB
R1	0x000000A5
R2	0x40021000
R3	0x00000090
R4	0x000000F7
R5	0x000000B3
R6	0x00000000
R7	0x20000000

그림 18 Line 20 수행 후 레지스터 값

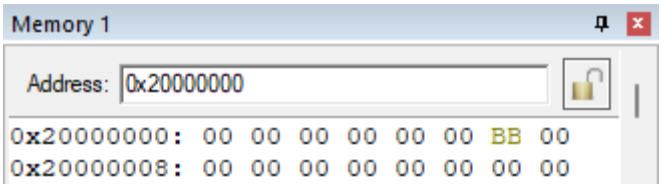

그림 19 Line 20 수행 후 메모리 값

먼저 line 19와 line 20의 명령어를 살펴보면 먼저 [그림 17]에서 볼 수 있듯이 code영역에 선언되어 있는 ramdata는 16바이트 공간을 할당 받고 r7에 주소를 저장하였다. 그리고 line 20에 의해 0x20000006번지에 r0의 데이터를 저장해주었고 그 결과는 [그림 19]에서 확인 가능하다.

다음으로 메모리에 저장한 데이터에 line 12에서와 같이 masking을 할 수 있도록 명령어를 작성해주었다. 메모리에서 데이터를 불러오고, masking하고, 저장하고 최소한 3번의 명령어가 필요하기 때문에 line 21-22 두 줄에는 넣지 못하고 아래 [그림 20]과 같이 3줄로 코드를 작성하였다.

```
    21:         ldr r2,[r7,#0x6]
0x080002F4 F8D72006   LDR      r2,[r7,#0x06]
    22:           and r6,r2,r4
0x080002F8 EA020604   AND      r6,r2,r4
    23:           strb r6,[r7,#0x6]
0x080002FC 71BE       STRB     r6,[r7,#0x06]
```

그림 20 직접 작성한 line 21 ~ line 23 명령어

R0	0x000000BB
R1	0x000000A5
R2	0x000000BB
R3	0x00000090
R4	0x000000F7
R5	0x000000B3
R6	0x000000B3
R7	0x20000000

그림 21 Line 23 수행 후 레지스터 값

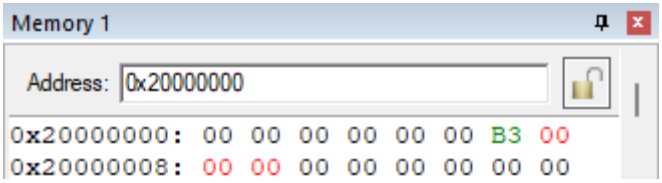

그림 22 Line 23 수행 후 메모리 값

Line 21을 통해서 0x20000006의 데이터 0xBB를 r2에 저장하고 line 22에서 mask가 저장되어 있는 r4와 and한 결과 0xB3를 저장해주었다. 그리고 line 23에서 다시 연산 된 결과를 0x20000006번지에 저장해주었고 그 결과는 [그림 22]에서 확인 가능하다.

추가적으로 masking이 아니라 bit-banding으로도 동일한 기능을 수행할 수 있기 때문에 한 번 더 코드를 작성해주었다.

```
    21:           ldr r8,=0x22000000
0x080002F4 F04F5808  MOV       r8,#0x22000000
    22:           mov r9,#0x0
0x080002F8 F04F0900  MOV       r9,#0x00
    23:           str r9,[r8,#0xCC]
0x080002FC F8C890CC  STR       r9,[r8,#0xCC]
```

그림 23 직접 작성한 line 21 ~ line 23 명령어

R0	0x000000BB
R1	0x000000A5
R2	0x40021000
R3	0x00000090
R4	0x000000F7
R5	0x000000B3
R6	0x00000000
R7	0x20000000
R8	0x22000000
R9	0x00000000

그림 24 Line 23 수행 후 레지스터 값

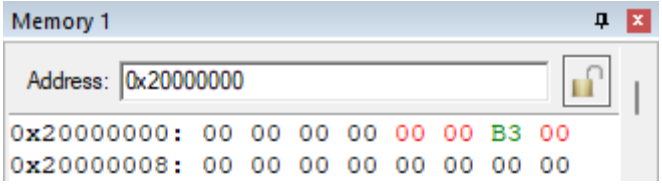

그림 25 Line 23 수행 후 메모리 값

Bit-banding으로 동일한 기능을 수행하기 위해서 먼저 line 21에서 bit-band alias 영역의 시작 주소 0x22000000을 r8에 저장해주었다. 그리고 우리가 수행하는 동작인 특정 bit를 0으로 바꾸는 것 이기 때문에 이를 위해 line 22에서 r9에 0을 저장해주었다. 사실 이번 프로그램에서는 r9을 사용한 적이 없기 때문에 필요 없는 명령어지만 좀 더 탄탄한 프로그램을 만들기 위해서 line 22도 추가하였다. 마지막으로 line 23에서 우리가 수정하려는 bit가 0x20000006번지의 4번째 bit를 0으로 바꾸려는 것 이기 때문에 32 * 6 + 4 * 3 = 204 즉 16진수로 0xCC이기 때문에 r7 + 0xCC를 한 주소에 0을 저장해주었다. 그 결과 [그림 25]에서 확인 가능한 것처럼 메모리의 값이 원하던 대로 0xB3이 된 것을 확인할 수 있다.

STEP 7: Lines 23-24의 수행과 A-4페이지의 설명을 통해 bic 명령어의 동작이 다른 논리 명령어들과 어떻게 구분되는지 확인한다.

```
    23:         ldr r0,data3
0x080002F4 4804      LDR      r0,[pc,#16]   ; @0x08000308
    24:         bic r6,r0,#0x0f0
    25: ;
0x080002F6 F02006F0  BIC      r6,r0,#0xF0
```

그림 26 Line 23, line 24 명령어

R0	0x00011234
R1	0x000000A5
R2	0x40021000
R3	0x00000090
R4	0x000000F7
R5	0x000000B3
R6	0x00011204

그림 27 Line 24 수행 후 레지스터 값

Line 23의 명령어는 간단히 data3의 값 0x11234를 r0에 저장하는 명령어이다. 그리고 line 24는 bic r6,r0#0x0f0라는 명령어로 먼저 bic는 operand2의 값을 not을 취해서 and를 해주는 명령어이다. 즉 operand2에서 1이 저장된 bit만 0으로 reset해주는 명령어이다. 따라서 우리가 line 24에서 수행한 명령을 해석하면 0xf0는 1111_0000임으로 r0에서 5~8번째 bit만 0으로 reset하고 나머지는 그대로 둔 채 r6에 저장을 하는 명령어이다. 그 결과를 보면 기존 r0의 데이터는 0x11234로 0001 0001 0010 0011 0100이고 여기서 5~8번째 bit만 reset해주면 0001 0001 0010 0000 0100즉 11204가 되고 이 결과는 [그림 27]에서 정확히 확인 가능하다. 따라서 이 bic 명령어는 다른 논리

명령어와 다르게 원래라면 2개의 명령어를 통해서 수행해야 할 명령을 하나의 명령어에 집약시켜 사용자로 하여금 더 편하게 프로그램을 만들 수 있게 해주었다.

STEP 8: Lines 26-27 명령어의 수행을 통해 saturation 명령어가 어떻게 동작하는지 확인한다. 이 과정에서 line 27 명령어의 #16을 #12로 변경해 수행해 본다. 또한 Q flag가 set 됨을 확인한다.

```
    26:         ldr r5,data3
0x080002FA 4D03       LDR      r5,[pc,#12]  ; @0x08000308
    27:         ssat r6,#16,r5
0x080002FC F305060F   SSAT     r6,#16,r5
```

그림 28 Line 26, line 27 명령어

Register	Value
Core	
R0	0x00011234
R1	0x000000A5
R2	0x40021000
R3	0x00000090
R4	0x000000F7
R5	0x00011234
R6	0x00007FFF
R7	0x20000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000610
R14 (LR)	0x0800013B
R15 (PC)	0x08000300
xPSR	0x29000000
N	0
Z	0
C	1
V	0
Q	1
T	1
IT	Disabled
ISR	0

그림 29 Line 27 명령어 수행 후 레지스터 값

Lind 26은 간단히 data3 0x11234를 r5에 불러오는 명령어이다. 그리고 line 27은 ssat r6,#16,r5라는 명령어로 ssat라는 명령어는 signed value가 저장된 레지스터의 값을 원하는 bit로 잘라서 목적지 레지스터에 넣는 명령어이다. Line 27 명령어를 이용해 설명하면 r5에 저장된 데이터를 signed 16-bit로 표현해 r6에 저장하는 명령어이다. 하지만 이때 0x11234는 signed 16-bit로 표현 가능한 범위를 벗어나기 때문에 saturation이 발생하고 표현할 수 있는 가장 큰 숫자인 7FFF로 r6에 저장해 주고 saturation이 발생했다는 것을 Q flag를 1로 set해서 알려준다.

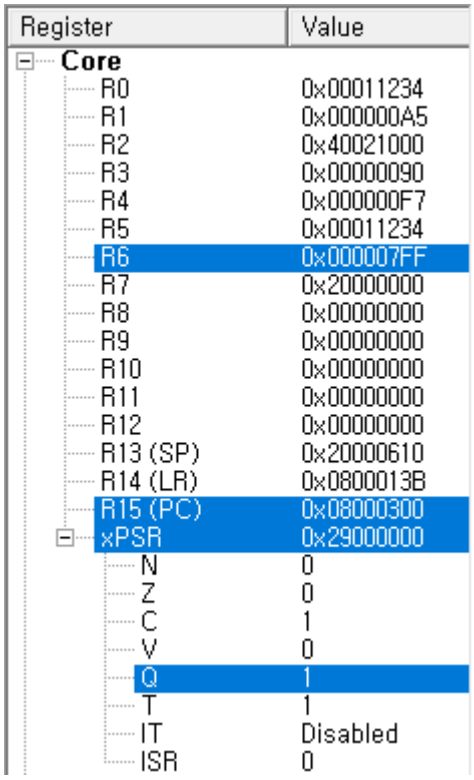

그림 30 Line 27 수정 후 레지스터 값

그리고 #16대신 #12로 해서 12-bit로 표현해 저장하면 위의 [그림 28]과 같은 결과가 나온다. 0x11234를 signed 12-bit로 나타낸 결과 최대한 나타낼 수 있는 0x7FF가 r6에 저장되었고 PSR의 Q flag를 보면 역시나 saturation이 발생했기 때문에 1로 set된 것을 확인할 수 있다.

STEP 9: MRS와 MSR 명령어를 이용하여 Q flag를 0으로 reset 시키는 codes를 앞서 배운 논리 명령어를 이용하여 작성하고 수행해본다.

```
    28:             mrs r7,apsr
0x08000300 F3EF8700  MRS       r7,APSR
    29:             bic r7,#0x08000000
0x08000304 F0276700  BIC       r7,r7,#0x8000000
    30:             msr apsr,r7
0x08000308 F3878800  MSR       APSR,r7
```

그림 31 직접 작성한 line 28 ~ line 30 코드

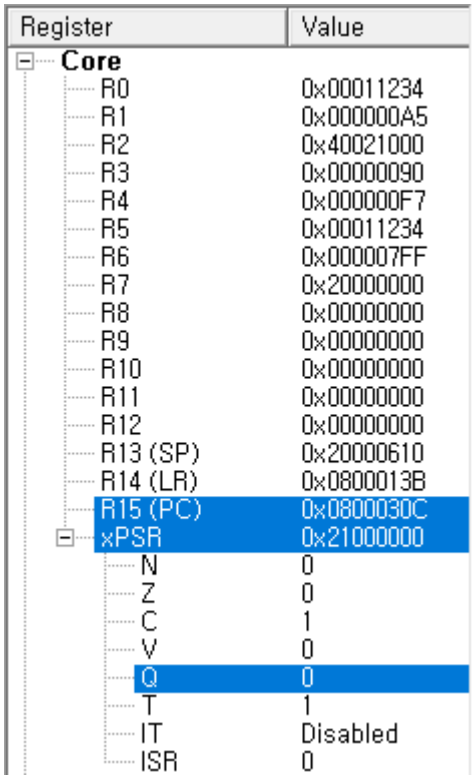

그림 32 Line 30 수행 후 레지스터 값

[그림 31]에서 볼 수 있듯이 mrs명령어로 Q flag값이 들어있는 APSR의 값을 r7으로 불러온 뒤 Q flag는 28번째 bit임으로 0x08000000과 bic를 해서 28번째 bit만 0으로 reset해주었다. 그리고 다시 msr명령어를 통해서 수정된 값을 원래 APSR에 다시 넣어주었다. 그 결과 [그림 32]를 통해서 Q flag값이 다시 0으로 resest된 것을 확인할 수 있다.

STEP 10: Line 31을 data1 dcb '8'로 교체하고 line 9 명령어까지 수행 한 후 그 결과를 STEP 3의 전반부 결과와 비교한다. 그 차이점을 line 31의 변경과 연관지어 설명해본다.

```
     5: strat ldrb r0,data1
0x080002C8 F89F0038  LDRB     r0,[pc,#56]  ; @0x08000304
     6:       ldrb r1,data2
0x080002CC F89F1035  LDRB     r1,[pc,#53]  ; @0x08000305
     7:       ldrb r3,mask1
0x080002D0 F89F3038  LDRB     r3,[pc,#56]  ; @0x0800030C
     8:       ldrb r4,mask2
0x080002D4 F89F4035  LDRB     r4,[pc,#53]  ; @0x0800030D
     9:       ands r5,r0,r3
    10: ;     tst r0,r3
0x080002D8 EA100503  ANDS     r5,r0,r3
```

그림 33 코드 수정 후 line 5 ~ line 9 명령어

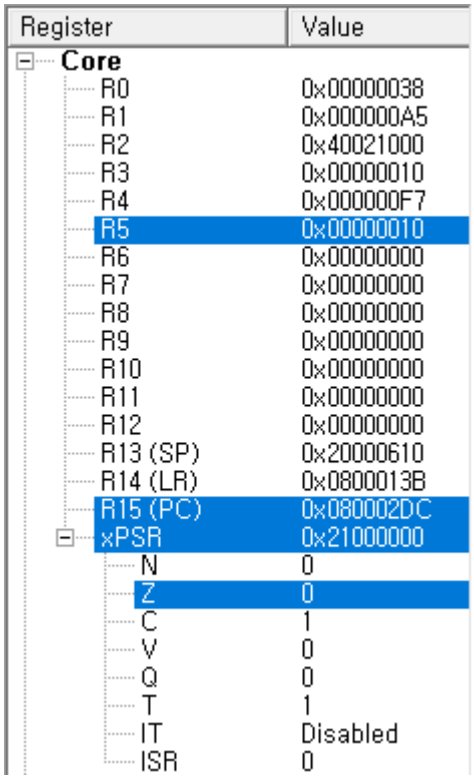

그림 34 코드 수정 후 line 9 실행 후 레지스터 값

data1 dcb 2_10110011에서 data1 dcb '8'로 바꾸고 line 9까지 명령어를 수행하였다. Line 9의 and에서 사용되는 r0와 r3레지스터에는 각각 r0에는 data1의 값이, r3에는 mask1이 들어있다. [그림 34]를 통해 보면 r0에는 0x38가 저장되어 있는데 이는 '8'의 ASCII코드 값이다. 즉 어셈블러가 '8'을 문자 8로 인식해서 자동으로 ASCII값으로 변환을 해준 것을 알 수 있다. 따라서 STEP 3와는 다른 data1의 값인 '8', 즉 0x38, 즉 0011 1000가 사용되어 r3의 1001 0000을 and한 결과 0001 0000이 되고 0x10이 r5에 저장된 것을 [그림 34]를 통해서 확인할 수 있다. 또한 이번에도 and의 결과값이 0이 아니기 때문에 Z flag는 0으로 reset되었다.

2) 실험 2

STEP 11: Program 4.2를 이용하여 프로젝트를 생성한다.

```
1        area lab4_2,code
2        entry
3   __main proc
4        export __main [weak]
5   strat
6        ldr r0,data3
7        asrs r1,r0,#0x3
8   ;
9   ;
10       ldr r0,data2
11       mov r1,#0x02
12       mul r2,r0,r1
13  ;
14       ldr r2,data1
15       umull r3,r4,r0,r2
16  ;
17
18       b .
19       endp
20       align
21  data1 dcd 0x67000005
22  data2 dcd 0x41000000
23  data3 dcd 0xb7000005
24  data4 dcd 0xb1234567
25  data5 dcd 0xa0000000
26       end
```

그림 35 직접 작성한 lab4_2.s 코드

STEP 12: Line 7 명령어를 수행하여 데이터에 대한 shift 동작을 이해한다. Line 7의 asrs를 lsrs, lsls, rors, rrxs로 각각 변경하면서 수행하여 각 명령어의 동작을 그림 4.5와 비교하며 이해한다. 이 과정에서 C flag의 변화를 함께 확인한다.

```
    6:        ldr r0,data3
0x080002C8 4807      LDR      r0,[pc,#28]  ; @0x080002E8
    7:        asrs r1,r0,#0x3
    8: ;
    9: ;
0x080002CA 10C1      ASRS     r1,r0,#3
```

그림 36 ASRS 명령어 사용 코드

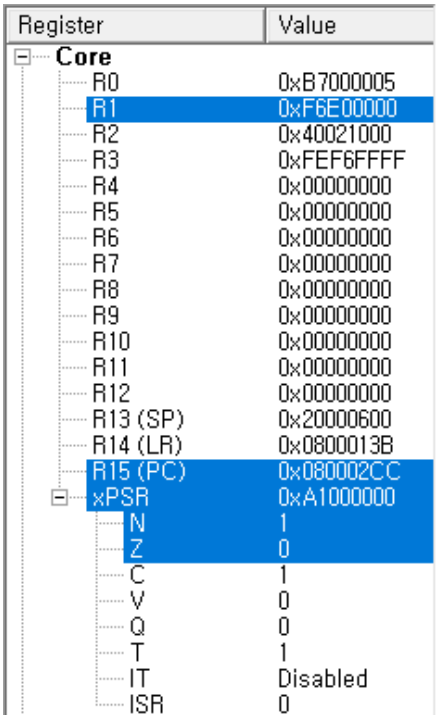

그림 37 ASRS 명령어 사용 후 레지스터 값

먼저 line 6의 경우는 data3인 0xB7000005를 r0로 가져오는 명령어이다. 이를 2진법으로 나타내면 1011 0111 0000 0000 0000 0000 0000 0101이 된다. 그리고 line 7 명령어는 r0를 signed로 취급해서 오른쪽으로 3칸 shift하고 r1에 저장한다. 따라서 그 결과는 1111 0110 1110 0000 0000 0000 0000 0000 즉0xF6E00000가 되고 원래 데이터의 3번째 bit가 캐리로 들어가 C flag는 1이 되고 수행 결과 MSB가 1이기 때문에 음수로 취급해 N flag는 1이 되고, 연산 결과 0이 아니기 때문에 Z flag는 0이 된다. 이는 교재 그림 4.5의 (a)의 동작이다.

다음으로 line 7에서 lsrs명령어를 사용한 경우 r0의 데이터를 unsiged로 취급하고 오른쪽으로 3칸 shift해서 r1에 저장한다. 따라서 0001 0110 1110 0000 0000 0000 0000 0000즉 0x16E00000가 되고 이를 [그림 39]에서 확인할 수 있다. 그리고 원래 데이터의 위의 3번째 bit가 캐리로 들어가 C flag는 1이되고 MSB가 0이기 때문에 N flag는 0이되고 또한 연산 결과가 0이 아니기 때문에 Z flag도 0이 된다. 그리고 이는 교재 그림 4.5의 (b)의 동작이다.

```
      6:        ldr r0,data3
0x080002C8 4807      LDR      r0,[pc,#28]  ; @0x080002E8
      7:        lsrs r1,r0,#0x3
      8: ;
      9: ;
0x080002CA 08C1      LSRS     r1,r0,#3
```

그림 38 LSRS 명령어 사용 코드

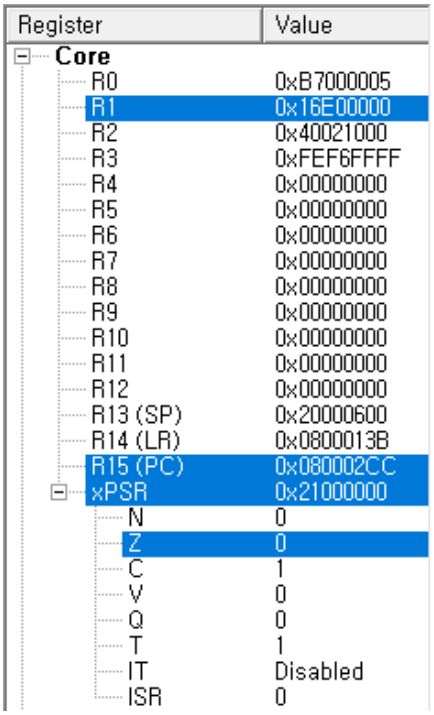

그림 39 LSRS 명령어 사용 후 레지스터 값

```
      6:        ldr r0,data3
0x080002C8 4807      LDR      r0,[pc,#28]  ; @0x080002E8
      7:        lsls r1,r0,#0x3
      8: ;
      9: ;
0x080002CA 00C1      LSLS     r1,r0,#3
```

그림 40 LSLS 명령어 사용 코드

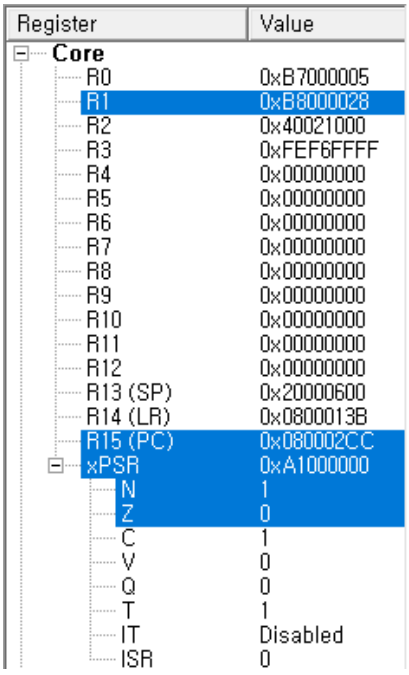

그림 41 LSLS 명령어 사용 후 레지스터 값

다음으로 lsls명령어를 사용한 경우 r0의 데이터를 unsiged로 취급하고 왼쪽으로 3칸 shift해서 r1에 저장한다. 따라서 1011 1000 0000 0000 0000 0000 0010 1000즉 0xB8000028이 되고 이를 [그림 41]에서 확인할 수 있다. 그리고 원래 데이터의 위의 30번째 bit가 캐리로 들어가 C flag는 1이 되고 MSB가 1이기 때문에 N flag는 1이되고 연산 결과가 0이 아니기 때문에 Z flag는 0이 된다. 그리고 이는 교재 그림 4.5의 (c) 동작이다.

```
      6:        ldr r0,data3
0x080002C8 4807      LDR      r0,[pc,#28]  ; @0x080002E8
      7:        rors r1,r0,#0x3
      8: ;
      9: ;
0x080002CA EA5F01F0  RORS     r1,r0,#3
```

그림 42 RORS 명령어 사용 코드

Register	Value
Core	
R0	0xB7000005
R1	0xB6E00000
R2	0x40021000
R3	0xFEF6FFFF
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000600
R14 (LR)	0x0800013B
R15 (PC)	0x080002CE
xPSR	0xA1000000
N	1
Z	0
C	1
V	0
Q	0
T	1
IT	Disabled
ISR	0

그림 43 RORS 명령어 사용 후 레지스터 값

다음으로 rors명령어를 사용한 경우 r0의 데이터를 오른쪽으로 3칸 shift하고 밀리 데이터를 왼쪽에서 집어넣는 rotation을 해서 r1에 저장한다. 따라서 1011 0110 1110 0000 0000 0000 0000 0000 즉 0xB6E00000이 되고 이를 [그림 44]에서 확인할 수 있다. C flag는 변화가 없고 MSB가 1이기 때문에 N flag는 1이되고 연산 결과가 0이 아니기 때문에 Z flag는 0이 된다. 그리고 이는 교재 그림 4.5의 (d) 동작이다.

```
      6:        ldr r0,data3
0x080002C8 4807      LDR      r0,[pc,#28]  ; @0x080002E8
      7:        rrxs r1,r0
      8: ;
      9: ;
0x080002CA EA5F0130  RRXS     r1,r0,
```

그림 44 RRXS 명령어 사용 코드

Register	Value
Core	
R0	0xB7000005
R1	0xDB800002
R2	0x40021000
R3	0xFEF6FFFF
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000600
R14 (LR)	0x0800013B
R15 (PC)	0x080002CE
xPSR	0xA1000000
N	1
Z	0
C	1
V	0
Q	0
T	1
IT	Disabled
ISR	0

그림 45 RRXS 명령어 사용 후 레지스터 값

마지막으로 rrxs명령어는 r0의 데이터를 C flag와 포함해서 오른쪽으로 한 칸 shift해서 r1에 저장한다. 원래 데이터는 1011 0111 0000 0000 0000 0000 0000 0101이고 C flag에는 1이 들어 있었다. 따라서 1101 1011 1000 0000 0000 0000 0000 0010, 즉 0xDB800002가 r2에 저장되고 C flag는 LSB인 1이 들어간다. [그림 45]에서 동일한 결과가 나온 것을 확인할 수 있다. 그리고 이는 교재 그림 4.5의 (e) 동작이다.

STEP 13: 그림 4.6에서 소개된 64-bit 수의 덧셈을 구현하는 codes를 lines 8-9 위치에 작성한다. 이를 위해 그림에서 사용된 레지스터에는 Program 4.2의 다음 데이터들이 각각 저장되도록 code를 작성한다.

r0 : data3

r1 : data 4

r3 : data 1

r4 : data2

```
0x080002C8 480A      LDR      r0,[pc,#40]  ; @0x080002F4
     7:        asrs r1,r0,#0x3
0x080002CA 10C1      ASRS     r1,r0,#3
     8:           ldr r0,data3
0x080002CC 4809      LDR      r0,[pc,#36]  ; @0x080002F4
     9:           ldr r1,data4
0x080002CE 490A      LDR      r1,[pc,#40]  ; @0x080002F8
    10:           ldr r3,data1
0x080002D0 4B06      LDR      r3,[pc,#24]  ; @0x080002EC
    11:           ldr r4,data2
0x080002D2 4C07      LDR      r4,[pc,#28]  ; @0x080002F0
    12:           adds r2,r0,r1
0x080002D4 1842      ADDS     r2,r0,r1
    13:           adc r5,r3,r4
0x080002D6 EB430504  ADC      r5,r3,r4
```

그림 46 64-bit 덧셈을 위해 직접 작성한 코드

Register	Value
Core	
R0	0xB7000005
R1	0xB1234567
R2	0x6823456C
R3	0x67000005
R4	0x41000000
R5	0xA8000006
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000600
R14 (LR)	0x0800013B
R15 (PC)	0x080002DA
xPSR	0x31000000
N	0
Z	0
C	1
V	1
Q	0
T	1
IT	Disabled
ISR	0

그림 47 Line 13 실행 후 레지스터 값

위에 주어진 대로 데이터를 저장하고 64-bit 덧셈을 구현하는 코드를 위 [그림 46]과 같이 작성하였다. 먼저 line 7부터 line 10에 걸쳐서 덧셈할 데이터를 가져왔다. 그리고 교재 그림 4.6에서 소

개된 64-bit 덧셈 방법을 활용해서 line 11과 line 12를 작성하였다. 먼저 낮은 비트를 포함한 r0와 r1을 더해서 flag까지 반영하도록 adds를 이용해서 더한 뒤 그 결과를 r2에 저장하였다. 그리고 높은 비트를 포함한 r3와 r4를 캐리를 포함해서 더하도록 adc 명령어를 사용해 더한 뒤 r5에 저장하였다. 따라서 덧셈의 결과는 r5에는 높은 비트를, r2에는 낮은 비트를 저장하였다. 그 덧셈 결과를 살펴보면 우리가 더하려 했던 첫번째 64-bit 데이터는 0x67000005B7000005이고, 다음 데이터는 0x41000000B1234567이다. 따라서 두 숫자를 더하면 0xA80000066823456C가 나오고 r5에는 상위 비트 0xA8000006이 r2에는 하위 비트 0x6823456C이 저장된 것을 [그림 47]을 통해서 확인할 수 있다.

STEP 14: Lines 10-12 명령어들을 통해 결과가 32-bit 레지스터에 수용 가능한 곱셈의 수행 방법을 확인한다.

```
    10:         ldr r0,data2
0x080002CC 4805      LDR      r0,[pc,#20]   ; @0x080002E4
    11:         mov r1,#0x02
0x080002CE F04F0102  MOV      r1,#0x02
    12:         mul r2,r0,r1
    13: ;
0x080002D2 FB00F201  MUL      r2,r0,r1
```

그림 48 Line 10 ~ line 12 명령어

R0	0x41000000
R1	0x00000002
R2	0x82000000

그림 49 Line 12 수행 후 레지스터 값

Line 10을 통해 data2인 0x41000000을 r0로 로드하고 line 11을 통해 r1에 0x2를 로드한다. 그리고 line 13의 명령어를 통해서 r0와 r1을 곱해서 r2에 저장한다. 그 결과를 보면 [그림 49]에서 확인 가능 하듯 0x41000000의 2배인 0x82000000이 r2에 저장되어 있다. 이를 통해 곱셈 명령어인 mul의 동작을 확인해보았다.

STEP 15: Lines 14-15 명령어들을 통해 결과가 32-bit를 초과하는 두 32-bit 수의 곱셈 방법을 확인한다.

```
    14:         ldr r2,data1
0x080002D6 4A02       LDR      r2,[pc,#8]   ; @0x080002E0
    15:         umull r3,r4,r0,r2
    16: ;
    17:
0x080002D8 FBA03402   UMULL    r3,r4,r0,r2
```

그림 50 Line 14 ~ line 15 명령어

R0	0x41000000
R1	0x00000002
R2	0x67000005
R3	0x45000000
R4	0x1A270001

그림 51 Line 15 수행 후 레지스터 값

Line 14를 통해서 data1인 0x67000005를 r2에 로드한다. 그리고 line 15의 명령어는 unsinged 32-bit 데이터 r0와 r2를 곱해서 나온 64-bit결과 중 상위 bit가 r4에 저장되고 하위 bit가 r3에 저장된다. 따라서 r0와 r2의 곱셈의 결과인 0x1A27000145000000가 r4와 r3에 저장된 것을 [그림 50]을 통해서 확인할 수 있다. 이를 통해서 32-bit 데이터 두개를 곱해서 64-bit 데이터가 나오는 곱셈에 대한 연산을 확인해 보았다.

STEP 16: A-30 페이지의 내용을 참조하여 나눗셈을 수행하기 위한 codes를 작성하고 그 동작을 확인해 본다.

```
    16:           ldr r0,data5
0x080002DC 4807       LDR      r0,[pc,#28]   ; @0x080002FC
    17:           mov r1,#0x02
0x080002DE F04F0102   MOV      r1,#0x02
    18:           sdiv r2,r0,r1
0x080002E2 FB90F2F1   SDIV     r2,r0,r1
    19:           udiv r2,r0,r1
0x080002E6 FBB0F2F1   UDIV     r2,r0,r1
```

그림 52 직접 작성한 line 16 ~ line 19 명령어

R0	0xA0000000
R1	0x00000002
R2	0xD0000000

그림 53 SDIV 수행 후 레지스터 값

R0	0xA0000000
R1	0x00000002
R2	0x50000000

그림 54 UDIV 수행 후 레지스터 값

마지막으로 나눗셈의 동작을 확인하기 위해서 r0에는 0xA0000000를 로드하고 r1에는 0x2를 로드

하였다. 그리고 SDIV로 signed 32-bit 데이터의 나눗셈을 수행한 결과는 다음과 같다. 먼저 0xA0000000은 2진법으로 1010 0000 0000 0000 0000 0000 0000 0000이고 10진법으로는 -1,610,612,736이다. MSB를 signed로 고려해서 2로 나눗셈을 한 결과는 10진법으로 -805,306,368, 2진법으로는 1101 0000 0000 0000 0000 0000 0000 0000, 즉 0xD0000000이 나오게 된다. 이는 [그림 53]에서 확인할 수 있다.

다음으로 UDIV로 unsinged 32-bit 데이터의 나눗셈을 수행한 결과는 다음과 같다. 먼저 0xA0000000은 2진법으로 1010 0000 0000 0000 0000 0000 0000 0000이고 10진법으로는 -2,684,354,560이다. 그리고 이를 2로 나눈 결과는 10진법으로 1,342,177,280, 2진법으로는 0101 0000 0000 0000 0000 0000 0000 0000, 즉 0x50000000이 나오게 된다. 이는 [그림 54]에서 확인할 수 있다.

이를 통해서 signed, unsiged 32-bit 데이터의 나눗셈에 대해서 확인해보았다.

3) 실험 3

STEP 17: Program 4.3을 이용하여 프로젝트를 생성한다.

```
1       area lab4_3,code
2       entry
3   __main proc
4       export __main [weak]
5   strat
6       ldr r0,datal
7       bfc r0,#4,#7
8   ;
9       ldr r0,datal
10      ldr r1,data2
11      sbfx r0,r1,#8,#8
12  ;
13      ldr r0,data2
14      rev r1,r0
15  ;
16      ldr r0,data2
17      ldr r1,datal
18      sxtb r2,r0
19      sxth r2,r0
20      uxth r2,r0
21      sxtb r2,r1
22  ;
23      mov r1,#0x2
24      subs r1,r1,#0x1
25      subs r1,r1,#0x1
26      subs r1,r1,#0x1
27      subs r1,r1,#0x1
28      b .
29      endp
30      align
31  datal dcd 0xb1234567
32  data2 dcd 0xbabeface
33      end
```

그림 55 직접 작성한 lab4_3.s 코드

STEP 18: Lines 6-7의 수행을 통해 bfc 명령어가 어떻게 데이터를 bits 단위로 처리하는지 확인한다. 이 과정에서 A-7페이지를 참조한다.

```
    6:      ldr r0,datal
0x080002C8 480A      LDR      r0,[pc,#40]   ; @0x080002F4
    7:      bfc r0,#4,#7
    8: ;
0x080002CA F36F100A  BFC      r0,#4,#7
```

그림 56 Line 6 ~ line 7 명령어

R0	0xB1234007

그림 57 Line 7 수행 후 레지스터 값

Line 6을 통해서 data1인 0xB1234567을 r0에 로드하였다. 그리고 line 7에서 bfc r0,#4,#7 명령어을 수행하였는데 이는 bit field clear 명령어로 4번 bit부터 7개만큼의 bit를 0으로 reset한다. 따라서

원래 r0의 데이터를 2진수로 나타내면 1011 0001 0010 0011 0100 0101 0110 0111인데 여기서 4번 bit부터 7개의 bit를 reset하면 1011 0001 0010 0011 0100 0000 0000 0111, 즉 0xB1234007이 되는 것을 [그림 57]을 통해서 확인할 수 있다.

STEP 19: Lines 9-11의 수행을 통해 sbfx 명령어가 어떻게 데이터를 bits 단위로 처리하는지 확인한다. 이 과정에서 A-29페이지를 참조한다.

```
     9:        ldr r0,data1
0x080002CE 4809      LDR      r0,[pc,#36]  ; @0x080002F4
    10:        ldr r1,data2
0x080002D0 4909      LDR      r1,[pc,#36]  ; @0x080002F8
    11:        sbfx r0,r1,#8,#8
    12: ;
0x080002D2 F3412007  SBFX     r0,r1,#8,#8
```

그림 58 Line 9 ~ line 11 명령어

```
R0              0xFFFFFFFA
R1              0xBABEFACE
```

그림 59 Line 11 수행 후 레지스터 값

먼저 line 9를 통해 data1 0xB1234567을 r0에 저장하였고 line 10을 통해 data2인 0xBABEFACE를 r1에 저장하였다. 그리고 line 11에서 sbfx r0,r1,#8,#8 명령어를 수행하였는데 sbfx 명령어는 signed bit filed extract로 r1에서 8번 bit부터 8개 만큼을 복사해서 부호를 고려하여 r0에 저장하는 명령어이다. 따라서 r1의 8번 bit부터 8개만큼이면 2진수로 1111 1010임으로 이를 부호를 고려해서 r0에 저장하면 0xFFFFFFFA가 된다. 이 결과를 [그림 59]에서 확인할 수 있다.

STEP 20: Lines 13-14의 수행을 통해 little endian으로 표현된 데이터가 rev 명령어를 통해 big endian으로 전환됨을 확인한다.

```
    13:        ldr r0,data2
0x080002D6 4808      LDR      r0,[pc,#32]  ; @0x080002F8
    14:        rev r1,r0
    15: ;
0x080002D8 BA01      REV      r1,r0
```

그림 60 Line 13 ~ line 14 명령어

```
R0              0xBABEFACE
R1              0xCEFABEBA
```

그림 61 Line 14 수행 후 레지스터 값

먼저 line 13을 통해서 data2의 값 0xBABEFACE를 r0에 로드하였다. 그리고 line 14에서 rev r1,r0 명령어를 수행하였는데 rev는 word단위로 endian mode를 변환하는 명령어이다. 우리는 기본적으로 little endian을 사용하기 때문에 rev 명령어를 수행하면 big endian으로 바뀌게 된다. 0xBABEFACE를 big endian으로 바꾸면 바이트 단위로 잘려서 순서가 바뀌는데 따라서 0xCEFABEBA가 된다. 그리고 이를 [그림 61]을 통해서 확인할 수 있다.

STEP 21: Lines 16-21의 수행을 통해 sign extension이 어떻게 수행되는지 확인한다.

```
    16:         ldr r0,data2
0x080002DA 4807      LDR      r0,[pc,#28]   ; @0x080002F8
    17:         ldr r1,data1
0x080002DC 4905      LDR      r1,[pc,#20]   ; @0x080002F4
    18:         sxtb r2,r0
0x080002DE B242      SXTB     r2,r0
    19:         sxth r2,r0
0x080002E0 B202      SXTH     r2,r0
    20:         uxth r2,r0
0x080002E2 B282      UXTH     r2,r0
    21:         sxtb r2,r1
    22: ;
0x080002E4 B24A      SXTB     r2,r1
```

그림 62 Line 16 ~ line 21 명령어

R0	0xBABEFACE
R1	0xB1234567
R2	0xFFFFFFCE

그림 63 Line 18 수행 후 레지스터 값

먼저 line 16을 통해 data2인 0xBABEFACE를 r0로 로드하고 line 17을 통해서 data1인 0xB1234567을 r1에 로드하였다. 그리고 line 18에서 sxtb r2,r0를 수행하였는데 8-bit 데이터를 signed 32-bit로 extension하는 명령어로 r0에 저장된 8-bit 데이터는 0xCE로 이를 signed 32-bit로 늘리면 MSB가 1이었기 때문에 다 F로 채운 0xFFFFFFCE가 r2에 저장되며 이를 [그림 63]에서 확인 가능하다.

R0	0xBABEFACE
R1	0xB1234567
R2	0xFFFFFACE

그림 64 Line 19 수행 후 레지스터 값

다음으로 line 19에서 sxth r2,r0를 수행하였는데 16-bit 데이터를 signed 32-bit로 extension하는 명령어로 r0에 저장된 16-bit 데이터는 0xFACE로 MSB가 1이기 때문에 F로 다 채운 0xFFFFFACE가 r2에 저장되며 이를 [그림 64]에서 확인 가능하다.

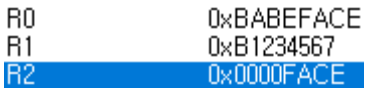

그림 65 Line 20 수행 후 레지스터 값

다음으로 line 20에서는 uxth r2,r0를 수행하였는데 16-bit 데이터를 unsigned 32-bit로 extension하는 명령어로 r0에 저장된 16-bit 데이터는 0xFACE임으로 여기에 0을 붙여 32-bit로 extension하면 0x0000FACE가 되고 이를 [그림 65]에서 확인 가능하다.

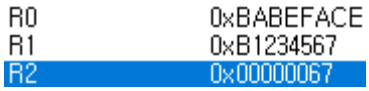

그림 66 Line 21 수행 후 레지스터 값

마지막으로 line 21에서는 line 18과 동일한 sxtb명령어로 이번에는 r1의 8-bit 데이터인 0x67을 signed 32-bit로 extension하는데 0x67은 MSB가 0이기 때문에 앞에 0을 붙여서 0x00000067이 되고 이를 r2에 저장하고 이를 [그림 66]에서 확인 가능하다.

STEP 22: Lines 23-27과 같이 연속적인 뺄셈을 수행하면서 그 결과와 함께 flag들의 변화를 확인한다.

```
    23:         mov r1,#0x2
0x080002E6 F04F0102  MOV      r1,#0x02
    24:         subs r1,r1,#0x1
0x080002EA 1E49      SUBS     r1,r1,#1
    25:         subs r1,r1,#0x1
0x080002EC 1E49      SUBS     r1,r1,#1
    26:         subs r1,r1,#0x1
0x080002EE 1E49      SUBS     r1,r1,#1
    27:         subs r1,r1,#0x1
0x080002F0 1E49      SUBS     r1,r1,#1
```

그림 67 Line 23 ~ line 27 명령어

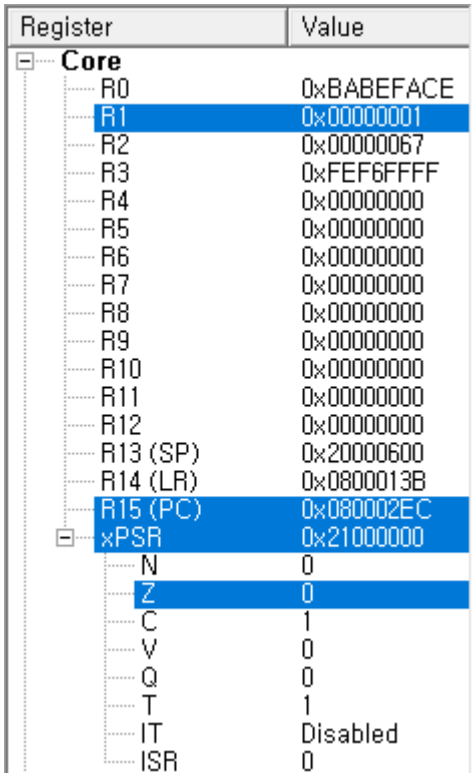

그림 68 Line 24 수행 후 레지스터 값

Line 23를 통해 r1에 2를 넣었다. 그리고 line 24에서 subs r1,r1,#1을 통해서 r1에서 1을 빼고 그 결과를 r1에 다시 넣고 flag를 업데이트 해주었다. 그 결과 당연히 r1에는 1이 저장되고 MSB가 0이기 때문에 N flag는 0, 결과가 0이 아니기 때문에 Z flag도 0, 뺄셈에서 borrow가 발생하지 않았기 때문에 C flag는 1이 되었다. 이 결과는 [그림 68]을 통해서 명확히 확인할 수 있다

다음으로 똑같이 한 번 더 r1에서 1을 빼서 r1에 저장하고 flag를 업데이트 해주었다. 그 결과 r1에는 0이 저장되었고 MSB가 0이기 때문에 N flag는 0, 연산 결과가 0이기 때문에 Z flag는 1, 이번에도 borrow는 발생하지 않았기 때문에 C flag는 1이 되었다. 이 결과는 아래 [그림 69]를 통해서 확인 가능하다.

.

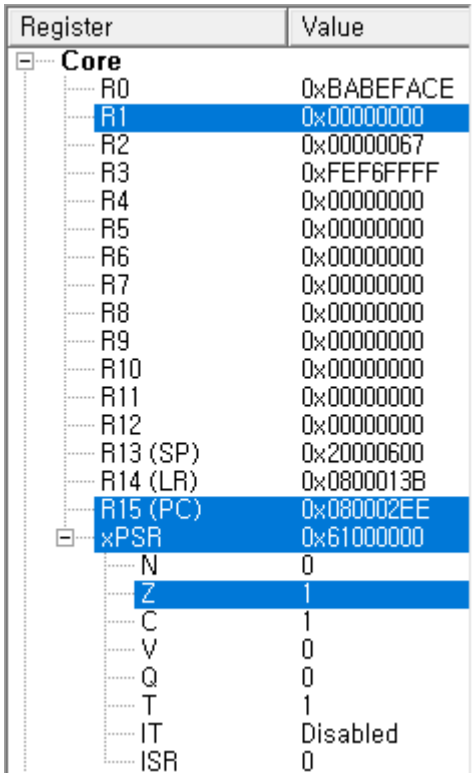

그림 69 Line 25 수행 후 레지스터 값

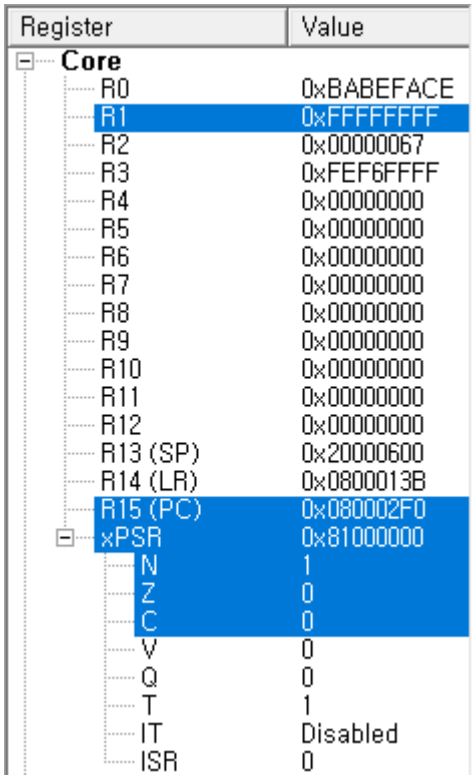

그림 70 Line 26 수행 후 레지스터 값

다음으로 똑같이 한 번 더 r1에서 1을 빼서 r1에 저장하고 flag를 업데이트 해주었다. 그 결과 r1에는 0에서 1을 빼주었기 때문에 -1을 의미하는 0xFFFFFFFF이 저장되었고 MSB가 1이기 때문에

N flag는 1, 연산 결과가 0이 아니기 때문에 Z flag는 0, 이번에는 borrow는 발생하였기 때문에 C flag는 0이 되었다. 이 결과는 위 [그림 70]를 통해서 확인 가능하다.

마지막으로 r1에서 1을 빼서 r1에 저장하고 flag를 업데이트 해주었다. 그 결과 r1에는 -1에서 1을 빼주었기 때문에 -2을 의미하는 0xFFFFFFFE이 저장되었고 MSB가 1이기 때문에 N flag는 1, 연산 결과가 0이 아니기 때문에 Z flag는 0, 이번에는 borrow는 발생하지 않았기 때문에 C flag는 1이 되었다. 이 결과는 아래 [그림 71]를 통해서 확인 가능하다.

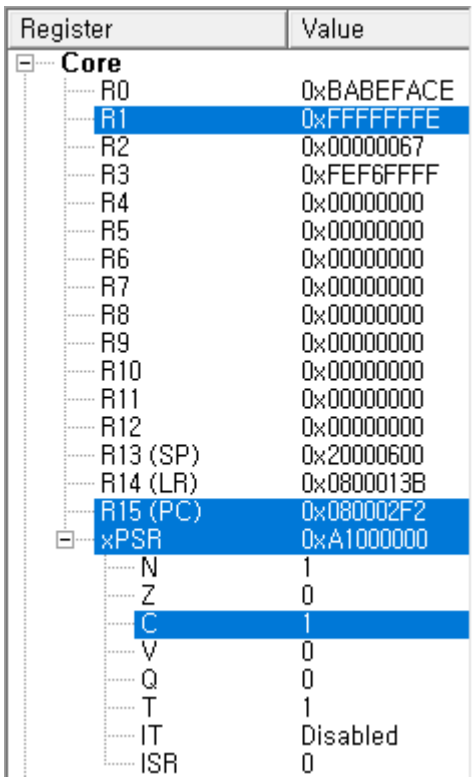

그림 71 Line 27 수행 후 레지스터 값

4. Exercises

1) 10진수에서 0-9 (또는 16진수에서 0x0-0xF)와 ASCII table에서의 '0'-'9' (또는 '0'-'9', 'A'-'F')는 어떻게 다른가? 이처럼 구별하여 사용하는 이유는 무엇인가?

10진수의 0-9또는 16진수의 0x0-0xF 말 그대로 숫자이다. 하지만 ASCII table에서의 '0'-'9', 'A'-'F'는 우리가 컴퓨터에서 문자를 나타내기 위해서 임의로 128개의 문자에 대해서 특정 8-bit로 약속을 한 것이다. 따라서 예를 들면 컴퓨터에서 8은 2진수로 1000으로 사용되지만 '8'은 00111000로 나타나진다. 이처럼 구분해서 사용 하는 이유는 우리가 컴퓨터를 사용해서 숫자만 다루는 것이 아니라 다양한 작업을 하는 과정에서 문자를 표시 할 필요도 있는데 이를 위해서 우리가 약속으로

ASCII 코드를 만든 것이기 때문에 반드시 구별하여 사용하여야 한다. 그렇지 않으면 헷갈릴 뿐만 아니라 잘못된 결과를 만들 수 있다.

2) 실험과정을 통해 확인한 내용을 중심으로 각 flag들의 역할을 기술해 보자.

실험과정에서 다룬 flag는 N, Z, C, Q flag이다. V flag는 다루지 않아서 추가 실험에서 다뤄 볼 예정이다. N flag는 용도는 연산된 데이터가 음수인지 양수인지 판단하는 flag인데 컴퓨터 입장에서는 사용자가 signed로 데이터를 사용하는지 unsigned로 데이터를 사용하는지 알 수 없기 때문에 MSB를 보고 1이라면 1로 set하고 0이라면 0으로 reset한다. 다음으로 Z flag는 가장 이해하기 쉬운 flag로 만약 명령어 수행 결과가 0인지 확인하는 flag로 연산 결과가 0이라면 Z flag를 1로 set하고 0이 아니라면 0으로 reset한다. 다음으로 C flag는 우리가 사용하는 cortex-m3에서는 다양한 용도로 사용한다. 먼저 덧셈과 같은 연산에서 carry bit가 발생하면 1로 set한다. 또는 뺄셈에서 borrow bit가 발생하지 않으면 1로 set한다. 그리고 shift와 관련한 명령어에서도 1bit짜리 여유공간으로 shift되었을 때 밀려난 bit를 저장하는 역할을 한다. 마지막으로 Q flag는 saturation의 발생 여부를 체크해주는 flag로 만약 명령어 수행 과정에서 saturation이 발생하면 Q flag를 1로 set하게 된다.

3) 연산 또는 bit 단위 데이터 처리 중 부호를 어떻게 해석하고 다룰 것인지를 결정하는 것이 중요하다. Shift 동작, 그리고 sign extension 명령어들을 통해서 부호들의 고려 여부에 따라 결과가 어떻게 다른지 확인해보자.

Shift 동작에서 부호가 중요한 것은 ASR명령어를 수행할 때 중요하다. 만약 ASR를 할 때 MSB가 1이라면 부호를 그대로 유지하기 위해서 shift를 하고 생긴 MSB를 계속해서 1로 채울 것이다. 하지만 만약 LSR로 부호를 고려하지 않는 명령어였다면 shift하고 생긴 MSB 공간을 0으로 채울 것이다.

다음으로 sign extension명령어에는 SXTB, SXTH, UXTB, UXTH로 총 4가지가 존재한다. 이 중에서 SXTB와 SXTH는 부호를 고려하는데 따라서 만약 extension하려는 데이터의 MSB가 1이라면 extension후 나머지 공간에 전부 1을 채우게 된다. 반면 UXTB와 UXTH는 그냥 남은 공간을 전부

0으로 채우게 된다.

이를 통해 shift나 sign extension 명령어들이 부호를 고려하는지 여부에 따라서 MSB에 1이 붙을 수도 있고 0이 붙을 수도 있음을 알 수 있다.

4) Sign extension을 C 언어의 type casting과 연관지어 해석해보자.

C언어에서 type casting은 특정 타입을 다른 타입으로 바꾸어 주는 것을 말한다. 컴파일러가 자동으로 수행할 수도 있고 사용자가 직접 지시할 수도 있다. 예를 들어 char a = 33; 이라는 코드를 통해 a에 33을 저장하고 (int)a 와 같이 a를 사용하면 char 타입이던 a가 int 타입으로 취급받게 된다. 이때 당연히 (int)a도 3의 값을 가지게 된다. 이때 내부적으로 사용되는 것이 sign extension이다. 이 경우는 c언어에서 int가 부호를 고려하는 타입이기 때문에 sign을 고려해서 extension을 하게 된다. 하지만 만약 unsigned char를 unsigned int로 type casting하는 경우는 부호를 고려하지 않는 sign extension을 하게 된다. 즉 c언어와 같은 high level language에서 데이터 타입에 따라서 어떤 sign extension을 사용할지 정해지게 된다.

5) MOV r4, r6, LSL #4와 같이 명령어의 operand에도 shift 명령어가 포함된다. Barrel shift를 통한 이러한 구현이 ARM의 강점으로 소개되는데 그 이유를 문헌을 통해 확인해 보자.

STM32F10xxx/20xxx/21xxx/L1xxx Cortex-M3 Programming Manual의 52쪽을 살펴보면 flexible second operand에서 레지스터에 shift연산을 추가적으로 수행할 수 있다는 설명이 나온다. 우리가 위에서 살펴본 ASR, LSR, LSL, ROR, RRX 모든 shift 명령어를 레지스터의 데이터에 수행할 수 있으며 이에 따라 carry flag가 업데이트 된다면 carry flag도 업데이트 된다. 단, 레지스터의 값이 업데이트 되지는 않는다.

아래 [그림 72]는 ARM7TDMI Technical Reference Manual에서 가져온 그림으로 이 그림을 통해서 하드웨어적으로 명령어에서 shift가 어떻게 가능한지 확인할 수 있다. 그림을 보면 레지스터의 데이터가 버스를 통해 barrel shifter를 통과해 32-bit ALU로 들어가는 것을 확인할 수 있다. 따라서 레지스터에서 ALU로 데이터가 이동하는 과정에서는 반드시 barrel shifter를 지나는 것을 확인할

수 있다. 따라서 명령어를 수행할 때 shift 여부와는 상관없이 모두 동일한 시간이 소요되고 따라서 필요하다면 레지스터의 데이터에서 shift 명령까지 수행한 데이터를 MOV와 같은 명령어의 operand로 사용할 수 있다.

따라서 원래 barrel shifter가 없었다면 2개의 명령어를 통해서 수행해야 하는 동작을 하나의 동작으로 가능하게 함으로써 ARM 프로세서를 사용할 때 좀 더 flexible하고 efficient한 프로그램을 만들 수 있다.

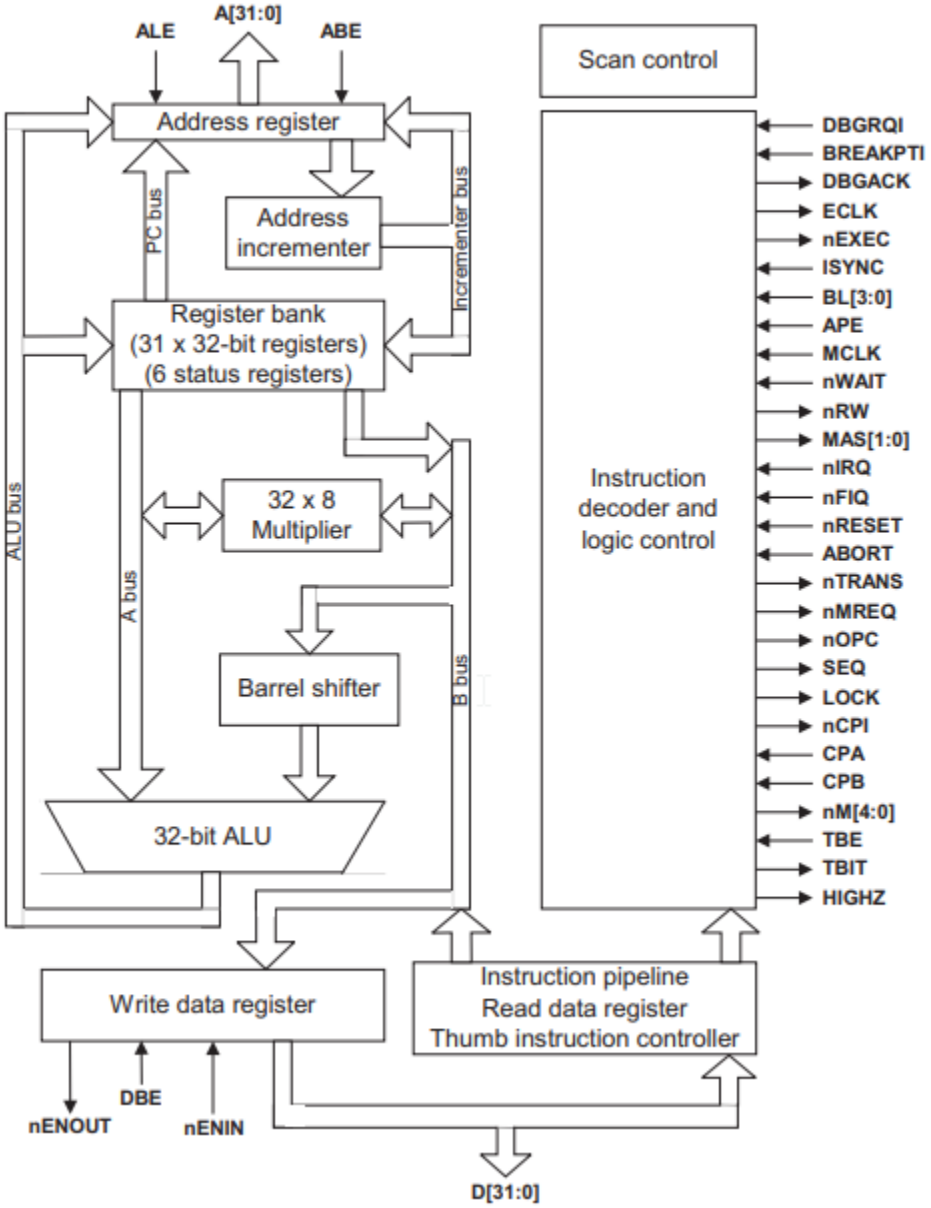

그림 72 ARM7TDMI main processor architecture

6) CMP 명령어와 SUBS 명령어의 공통점과 차이점에 대해서 알아보자. 마찬가지로 TST 명령어와 ANDS 명령어에 대해서도 알아보자.

먼저 SUBS와 CMP모두 명령어를 수행하기 위해서 Rn과 operand2가 필요하다. 두 명령어의 공통점은 Rn에서 operand2를 빼고 그 결과를 이용해 flag를 업데이트 한다는 점이 동일하다. 다만 차이점으로는 SUBS는 연산 결과를 Rn또는 다른 Rd가 있다면 Rd에 저장하는데 비해서 CMP명령어는 연산 결과는 버린다는 차이점이 있다.

마찬가지로 TST와 ANDS 명령어를 수행하기 위해서 Rn과 operand2가 필요하다. 두 명령어의 공통점은 Rn과 operand2를 AND연산을 수행한 뒤 그 결과를 이용해 flag를 업데이트 한다는 점이 동일하다. 다만 차이점으로는 ANDS는 연산 결과를 Rn또는 다른 Rd가 있다면 Rd에 저장하는데 비해서 TST 명령어는 연산 결과를 버린다는 차이점이 있다.

따라서 이러한 명령어들의 차이점을 잘 알고 상황에 알맞게 잘 사용해주어야 한다.

5. 추가 실험

1) Fixed-point arithmetic

우리가 컴퓨터로 데이터를 다룰 때 정수만 다루는 것이 아니라 실수도 다룰 필요가 있다. 따라서 실수를 나타내기 위한 방법 중 하나로 fixed-point라는 방법이 있다. 먼저 32-bit fixed-point는 MSB는 부호를 위한 sign bit, 그 다음 15bit는 정수부를 나타내고, 그 다음 16bit는 소수부를 나타낸다. 이때 정수부와 소수부의 bit크기는 약속만 된다면 임의로 수정하여도 괜찮다.

숫자로 예를 들어보면 10진수 7.625는 2진수로 111.101로 나타낼 수 있고 10진수 0.25는 2진수로 0.01로 나타낼 수 있다. 이를 이용해서 아래 [그림 73]의 프로그램에서 계산을 해보려 한다.

```
1        area lab4_4,code
2        entry
3   __main proc
4        export __main [weak]
5   strat ldr r0,data1
6        ldr r1,data2
7        add r2,r0,r1
8        sub r3,r0,r1
9        smull r4,r5,r0,r1
10       lsl r5,#16
11       lsr r4,#16
12       orr r6,r4,r5
13       b .
14       endp
15       align
16  data1 dcd 0x0007A000 ;7.625
17  data2 dcd 0x00004000 ;0.25
18       end
```

그림 73 직접 작성한 lab4_4.s 코드

R0	0x0007A000
R1	0x00004000
R2	0x0007E000
R3	0x00076000
R4	0x0000E800
R5	0x00010000
R6	0x0001E800

그림 74 코드 실행 후 레지스터 값

먼저 data1에는 7.625를 fixed point방법으로 저장해주고 data2에는 0.25를 fixed point 방법으로 저장해주었다. 그리고 line 5, line 6을 통해서 r0, r1에 각각 데이터를 로드했다. 그리고 line 7을 통해 간단히 덧셈을 한 결과를 r2에 저장하였다. r2에 저장된 값을 보면 0x0007E000으로 fixed point로 해석하면 7.875를 나타낸다. 명령어 하나로 간단하게 덧셈이 가능한 것을 확인할 수 있다. 뺄셈도 마찬가지로 line 8을 통해 수행하면 r3에 0x00076000이 저장되고 이를 해석하면 7.375로 정확한 결과가 나온 것을 확인할 수 있다. 하지만 곱셈의 경우는 약간 복잡하다. 먼저 32-bit 데이터 2개를 곱하기 때문에 64-bit결과가 나올 수 있음으로 smull 명령어를 사용해준다. 그리고 그 결과를 소수부 부분만큼 shift해주면 결과가 나온다. 하지만 32-bit 레지스터 2개를 이어서 shift해줘야 해서 각각 레지스터를 shift하고 or하는 방법으로 계산해주었다. 그 과정이 line 9부터 line 12이다. 즉 곱셈 하나를 위해서 4개의 명령어를 사용했다. 곱셈을 수행한 최종 결과는 r6에 0x0001E800이 저장되었고 이를 해석하면 1.90625가 나오고 제대로 계산된 것을 확인할 수 있다. 이처럼 fixed point연산은 덧셈과 뺄셈에는 매우 유용하지만 곱셈의 경우는 많은 명령어가 필요한 것을 확인할 수 있었다.

2) Floating-point arithmetic

실수를 표현하는 다른 방법으로 floating point라는 방법이 사용되는데 가장 표준적으로 많이 사용하는 방법은 IEEE 754라는 방법이다. 32-bit에서 MSB는 부호를 위한 sign bit로 사용하고 다음 8-bit는 지수부, 다음 23-bit는 가수부이다. 먼저 지수부는 8-bit 공간을 이용해서 -127부터 128까지 나타내기 위해서 8-bit저장된 숫자에서 127을 빼서 생각한다. 그리고 가수부 부분은 원하는 값을 fixed point처럼 정규화해 1.xx*2^n꼴로 나타내고 맨 앞의 1은 제외하고 xx부분을 가수부로 저장한다.

예를 들어 보자면 위의 7.625라는 값을 floating point방식으로 나타내면 다음과 같아. 7.625를 정규화 하면 1.11101*2^2임으로 부호는 +, 지수부는 127+2, 가수부는 11101이 된다. 즉 0100 0000 1111 0100 0000 0000 0000 0000가 된다.

이 floating point로 가능한 연산 중 간단한 덧셈을 구현한 프로그램은 아래와 같다. 먼저 지수부를 맞춰주고 가수부를 더한 뒤 정규화를 하는 과정으로 진행된다.

```
1        area lab4_5,code
2        entry
3   __main proc
4        export __main [weak]
5   strat ldr r0,datal
6        mov r1,r0
7        bfc r1,#23,#9
8        orr r1,#0x00800000
9        add r2,r1,r1
10       lsr r2,#1
11       bfc r2,#23,#9
12       mov r3,r0
13       bfc r3,#0,#23
14       add r3,#0x00800000
15       orr r4,r3,r2
16       b .
17       endp
18       align
19  datal dcd 0x40F40000 ;7.625
20       end
```

그림 75 직접 작성한 lab4_5.s 코드

R0	0x40F40000
R1	0x00F40000
R2	0x00740000
R3	0x41000000
R4	0x41740000

그림 76 코드 실행 후 레지스터 값

프로그램은 간단하게 계산하기 위해서 지수부가 동일하게 7.625+7.625를 계산하였다. 먼저 계산 결과는 10진수로 15.25, 이를 floating point로 표현하면 0100 0001 0111 0100 0000 0000 0000 0000, 즉 0x41740000이 나오게 된다. 그 과정을 살펴보면 line 7을 통해서 지수부를 삭제하고 가수부만 남긴 뒤 line 8를 통해 가수부 앞에 1.을 붙여서 line 9를 통해 가수부끼리 더해주었다. 그리고 그 결과를 정규화 하기 위해 line 10에서 shift해주고 1칸 shift 했음으로 지수부에 1을 더해주기 위해 line 14를 수행하였다. 그리고 지수부와 가수부를 orr로 합쳐주면 계산 결과가 나오게 된다. [그림 76]을 통해서 r4에 우리가 원했던 0x41740000이 잘 저장되어 있는 것을 확인할 수 있다. 이처럼 floating point를 사용하면 지수부와 가수부를 나눠 맞추고 계산하고 다시 맞추고 하는 복잡한 과정을 거쳐야 하지만 더 큰 범위의 실수를 더 정확하게 표현할 수 있다는 장점이 있다.

3) V flag

실험과정에서 다양한 flag를 살펴보았는데 V flag에 대해서는 실험해보지 않아 추가 실험을 진행해보았다.

```
1        area lab4_6,code
2        entry
3   __main proc
4        export __main [weak]
5   strat ldr r0,data1
6        ldr r1,data2
7        ldr r3,data3
8        ldr r4,data4
9        adds r2,r0,r1
10       adds r5,r3,r4
11       b .
12       endp
13       align
14  data1 dcd 0x67000000
15  data2 dcd 0x41000000
16  data3 dcd 0xB1234567
17  data4 dcd 0xA0000000
18       end
```

그림 77 직접 작성한 lab4_6.s 코드

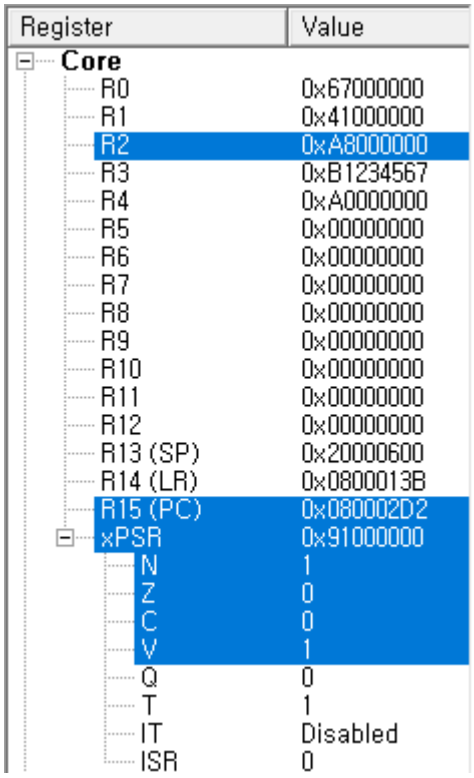

그림 78 Line 9 실행 후 레지스터 값

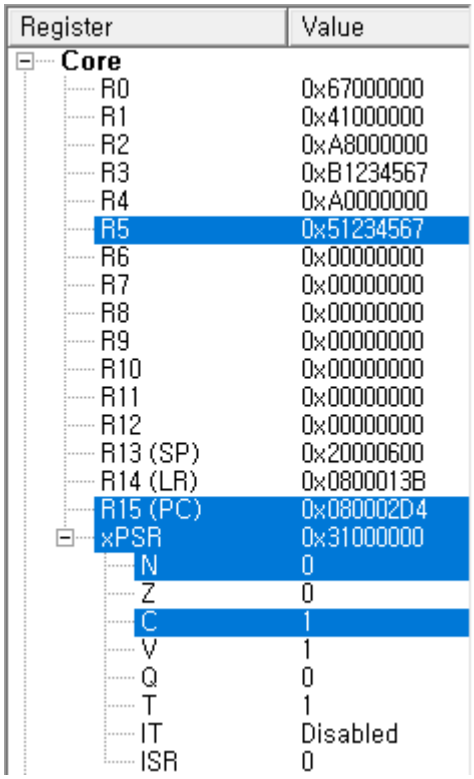

그림 79 Line 10 실행 후 레지스터 값

V flag는 singed 숫자의 계산을 할 때 주로 사용하는 flag이다. 따라서 31번째 bit에서 overflow가 발생해서 32번째 bit에 영향을 미쳤을 때 V flag가 1로 set된다. 이에 대해 확인하기 위해 두가지

예를 들어서 [그림 77]처럼 프로그램을 작성해보았다.

먼저 line 5 ~ line 8까지 r0, r1, r3, r4에 연산에 사용할 데이터를 로드했다. 그리고 line 9에서 adds를 통해 r0과 r1의 값을 더했다. r0에는 2진수로 0110 0111 0000 0000 0000 0000 0000 0000가 들어있고 r1에는 0100 0001 0000 0000 0000 0000 0000 0000가 들어있다. 따라서 singed 숫자로 보면 두 숫자는 모두 양수이다. 하지만 r0와 r1을 더하면 1010 1000 0000 0000 0000 0000 0000 0000가 되고 0xA8000000이 되고 MSB가 1이되어 양수 더하기 양수의 결과가 음수가 된다. 따라서 V flag가 1로 set 되었고 이를 [그림 78]에서 확인할 수 있다. 다음 예시로 r3에는 2진수로 1011 0001 0010 0011 0100 0101 0110 0111가 들어있고 r4에는 1010 0000 0000 0000 0000 0000 0000 0000가 들어있어 음수 + 음수를 계산하였다. 그 결과는 0001 0101 0001 0010 0011 0100 0101 0110 0111가 되고 32-bit로 자르게 되면 0101 0001 0010 0011 0100 0101 0110 0111만 남아 음수 더하기 음수가 양수가 된다. 즉 이 경우에도 overflow가 발생하였기 때문에 V flag가 1로 set된 것을 [그림 79]에서 확인할 수 있다.

이를 통해서 V flag는 singed 숫자끼리 연산을 진행할 때 overflow가 발생해 부호가 바뀌게 되면 set 되는 것을 확인할 수 있다.

4) MLA, UMLAL 명령어

이번 추가실험에서는 실험과정에서 다루지 않았던 명령어 중 MLA와 UMLAL 명령어의 동작에 대해 실험해보았다. 비슷한 명령어로 MLS나 SMLAL 명령어도 있지만 MLA, UMLAL 명령어와 동작이 비슷하기 때문에 위 2개의 명령어의 동작만 확인해보았다.

```
1        area lab4_7,code
2        entry
3   __main proc
4        export __main [weak]
5   strat ldr r0,data1
6        ldr r1,data2
7        ldr r2,data3
8        mla r3,r0,r1,r2
9        ldr r4,data4
10       ldr r5,data5
11       umlal r4,r5,r1,r2
12       b .
13       endp
14       align
15  data1 dcd 0x00004000
16  data2 dcd 0x00000003
17  data3 dcd 0x00006000
18  data4 dcd 0x00001234
19  data5 dcd 0x00005678
20       end
```

그림 80 직접 작성한 lab4_7.s 코드

R0	0x00004000
R1	0x00000003
R2	0x00006000
R3	0x00012000
R4	0x00013234
R5	0x00005678

그림 81 프로그램 수행 후 레지스터 값

코드를 간단히 설명하면 먼저 r0, r1, r2에 각각 연산에 사용할 data1, data2, data3를 로드했다. 그리고 line 8에서 mla r3,r0,r1,r2 명령어를 통해서 r0과 r1값의 곱에 r2를 더해서 r3에 저장하는 명령어를 수행했다. 그 결과 r0 곱하기 r1은 0xC000이고 여기에 r2의 값 0x6000을 더해 0x12000이 되고 이는 r3에 저장된 것을 [그림 81]에서 확인 가능하다. 다음으로 r4, r5에 각각 data4와 data5를 불러온 뒤 line 11에서 umlal r4,r5,r1,r2 명령어를 수행하였다. 그 결과는 r1 곱하기 r2의 결과를 r4를 낮은 비트를, r5를 높은 비트를 저장하는 레지스터로 해서 곱하기 결과인 64-bit데이터를 기존 r4와 r5에 저장되어 있던 데이터와 더하는 명령어이다. 그 결과 r1, r2의 곱 0x12000에 기본 r4에 저장되어 있던 0x1234가 더해져 0x13234가 r4에 저장되고 r5에는 0에 기존 데이터 0x5678이 더해져 0x5678이 저장되어 있는 것을 [그림 81]을 통해서 확인 가능하다. 이번 추가 실험을 통해서 mla명령어와 umlal명령어의 동작과 사용법을 알아보았다.

5) 64-bit multiplication

실험 과정에서 64-bit의 덧셈은 구해보았는데 곱셈도 충분히 있을 수 있다. 따라서 64-bit 데이터 끼리의 곱셈을 통해 128-bit의 결과가 나오는 경우를 한번 살펴보았다.

```
     5: strat ldr r0,data1
0x080002C8 4806      LDR      r0,[pc,#24]  ; @0x080002E4
     6:           ldr r1,data2
0x080002CA 4907      LDR      r1,[pc,#28]  ; @0x080002E8
     7:           ldr r2,data3
0x080002CC 4A07      LDR      r2,[pc,#28]  ; @0x080002EC
     8:           ldr r3,data4
0x080002CE 4B08      LDR      r3,[pc,#32]  ; @0x080002F0
     9:           umull r4,r5,r1,r3
0x080002D0 FBA14503  UMULL    r4,r5,r1,r3
    10:           umlal r5,r6,r0,r3
0x080002D4 FBE05603  UMLAL    r5,r6,r0,r3
    11:           umlal r5,r6,r1,r2
0x080002D8 FBE15602  UMLAL    r5,r6,r1,r2
    12:           umlal r6,r7,r0,r2
0x080002DC FBE06702  UMLAL    r6,r7,r0,r2
```

그림 82 직접 작성한 lab4_8.s 코드

R0	0x01234567
R1	0x89ABCDEF
R2	0x19981024
R3	0xBABEFACE
R4	0x0CBD1C52
R5	0xCEF7DC81
R6	0xB00C3A00
R7	0x001D1ECA

그림 83 프로그램 수행 후 레지스터 값

64-bit 데이터의 곱셈에서의 방법과 위에서 배운 UMLAL명령어를 활용해서 64-bit 데이터와 64-bit 데이터의 곱셈을 구현해보았다. 프로그램 수행 전 결과를 예상해보면 먼저 첫번째 64-bit 데이터는 0x0123456789abcdef로 2진법으로는 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111, 10진법으로는 81,985,529,216,486,895이다. 두번째 64-bit 데이터는 0x19981024babeface로 2진법으로는 0001 1001 1001 1000 0001 0000 0010 0100 1011 1010 1011 1110 1111 1010 1100 1110, 10진법으로는 1,844,241,797,346,163,406이다. 이 두 수를 곱하면 결과는 10진법으로는 151,201,139,758,590,183,298,041,929,927,564,370이고 16진법으로는 0x1d1ecab00c3a00cef7dc810cbd1c52이고 2진법으로 나타내면 0001 1101 0001 1110 1100 1010 1011 0000 0000 1100 0011 1010 0000 0000 1100 1110 1111 0111 1101 1100 1000 0001 0000 1100 1011 1101 0001 1100 0101 0010으로 128-bit 데이터인 것을 확인할 수 있다.

이 계산을 수행하기 위해 두 숫자를 32-bit 데이터 2개로 나눠 저장한 뒤 각 숫자를 r0, r1, r2, r3

에 로드했다. 그리고 UMUL명령어를 통해서 낮은 비트끼리 숫자를 진행해서 r4, r5에 저장해주었다. 그리고 UMLAL을 이용해서 이미 있는 r5값에 포함해서 r0와 r3의 곱을 진행한 뒤 r5, r6에 저장해주고 마찬가지로 r1, r2를 곱해서 r5, r6에 기존값에 더해서 저장하였다. 그리고 마지막으로 r6, r7에 r0와 r2를 곱해서 기존의 값에 더해 저장하였다. 그 결과를 살펴보면 r7, r6, r5, r4의 순서로 큰 비트로 128-bit 데이터가 저장되었다. 그리고 그 결과를 살펴보면 0x 001D1ECA B00C3A00 CEF7DC81 0CBD1C52로 우리가 예상한 결과 값과 동일하게 나온 것을 확인할 수 있다. 이 과정을 통해서 64-bit 데이터끼리 곱을 통해 128-bit 결과가 나오는 연산을 수행해 보았다.

6. 결론

이번 4주차 실험을 통해서 우리가 사용하는 ARM 프로세서의 어셈블리 언어에서 데이터 연산과 관련한 여러가지 명령어를 익히고 명령어를 직접 사용해보면서 이해할 수 있었다. 가장 기본적인 논리 연산자들을 비롯해 이러한 논리 연산자를 활용할 수 있는 다양한 방법과 산술 연산자, shift 연산자도 살펴보았고 마지막에는 bit field와 sign extension 관련한 명령어도 배웠다. 또한 이 과정에서 앞으로 중요하게 사용될 flag 값이 어떻게 변화하는지, 언제 변화하는지 등도 익힐 수 있었다. 마지막 추가 실험을 정수가 아니라 실수로 연산을 하는 방법 등 다양한 데이터 연산과 관련한 내용을 추가적으로 공부하였다. 이를 통해 앞으로 ARM 어셈블리 언어를 사용해 데이터를 다루거나 편하게 코드를 작성하고 코드를 읽고 동작을 이해하는 폭을 넓힐 수 있었다.

7. 참고문헌

서강대학교 전자공학과 (2023). 마이크로프로세서개론. p.79 ~ 95, Appendix A

STMicroelectronics (2017). STM32F10xxx/20xxx/21xxx/L1xxx Cortex-M3 Programming Manual (Rev 6) p.72 ~ 91

STMicroelectronics (2021). STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced ARM–based 32-bit MUCs Reference Manual (Rev 21).

ARM. (2001). ARM7TDMI Technical Reference Manual (Rev 3).

히언. (2021). EMBEDDED RECIPES.