

# 마이크로프로세서응용실험 LAB03 결과보고서

## Data Transfer

20181536 엄석훈

### 1. 목적

- 데이터 전달 명령어들의 종류와 동작에 대해 이해한다.
- Endian mode, addressing mode, bit-banding, pseudo-instruction에 대해 확인한다.
- 데이터의 유형 및 주소 해석에 대해 알아본다.
- 일부 directive의 역할을 확인한다.

### 2. 이론

#### 1) ARM assembly 주요 데이터 전달 명령어

**MOV Rd, Op2** : Op2의 데이터를 레지스터 Rd에 저장한다.

**ADR Rd, label** : PC에 상대적으로 적힌 label의 주소를 레지스터 Rd에 저장한다.

**LDR Rt, [Rn, #offset]** : word단위로 레지스터 Rn을 베이스 메모리 주소로 해서 offset만큼 떨어진 곳의 데이터를 레지스터 Rt에 저장한다.

**LDRB Rt, [Rn, #offset]** : byte단위로 레지스터 Rn을 베이스 메모리 주소로 해서 offset만큼 떨어진 곳의 데이터를 레지스터 Rt에 저장한다.

**LDRH Rt, [Rn, #offset]** : halfword단위로 레지스터 Rn을 베이스 메모리 주소로 해서 offset만큼 떨어진 곳의 데이터를 레지스터 Rt에 저장한다.

**STR Rt, [Rn, #offset]** : word단위로 레지스터 Rn을 베이스 메모리 주소로 해서 offset만큼 떨어진 곳에 레지스터 Rt의 데이터를 저장한다.

**STRB Rt, [Rn, #offset]** : byte단위로 레지스터 Rn을 베이스 메모리 주소로 해서 offset만큼 떨어진 곳에 레지스터 Rt의 데이터를 저장한다.

**STRH Rt, [Rn, #offset]** : halfword단위로 레지스터 Rn을 베이스 메모리 주소로 해서 offset만큼 떨어진 곳에 레지스터 Rt의 데이터를 저장한다.

## 2) Endian mode

Endian mode란 메모리에 데이터를 저장할 때의 순서를 의미한다. 데이터를 저장할 때 LSB를 낮은 주소에 저장하는 방식을 little-endian이라 하고, LSB를 높은 주소에 저장하는 방식을 big-endian이라 한다.

Address	Bits 31 – 24	Bits 23 – 16	Bits 15 – 8	Bits 7 – 0
0x1003 – 0x1000	Byte – 0x1003	Byte – 0x1002	Byte – 0x1001	Byte – 0x1000
0x1007 – 0x1004	Byte – 0x1007	Byte – 0x1006	Byte – 0x1005	Byte – 0x1004
...	Byte – 4xN+3	Byte – 4xN+2	Byte – 4xN+1	Byte – 4xN

그림 1 Little-endian 방식의 데이터 저장 예시

Address	Bits 31 – 24	Bits 23 – 16	Bits 15 – 8	Bits 7 – 0
0x1003 – 0x1000	Byte – 0x1000	Byte – 0x1001	Byte – 0x1002	Byte – 0x1003
0x1007 – 0x1004	Byte – 0x1004	Byte – 0x1005	Byte – 0x1006	Byte – 0x1007
...	Byte – 4xN	Byte – 4xN+1	Byte – 4xN+2	Byte – 4xN+3

그림 2 Big-endian 방식의 데이터 저장 예시

## 3) Addressing mode

Addressing mode는 프로그램에서 주소를 접근해야 할 때 원하는 주소에 접근하는 방법이다. 다양한 방법이 있지만 주로 offset addressing, pre-indexed addressing, post-indexed addressing의 3가지로 분류할 수 있다.

**Offset Addressing - [Rn, #offset]** : 레지스터 Rn의 값을 베이스 주소로 offset만큼 더해서 주소로 사용한다.

**Pre-indexed Addressing - [Rn, #offset]!** : 레지스터 Rn의 값을 베이스 주소로 offset만큼 더해서 주소로 사용한 뒤 Rn에 그 값을 저장한다.

**Post-indexed Addressing - [Rn], #offset** : 레지스터 Rn의 값을 주소로 사용하고 offset만큼 더해서 Rn에 새로운 주소를 저장한다..

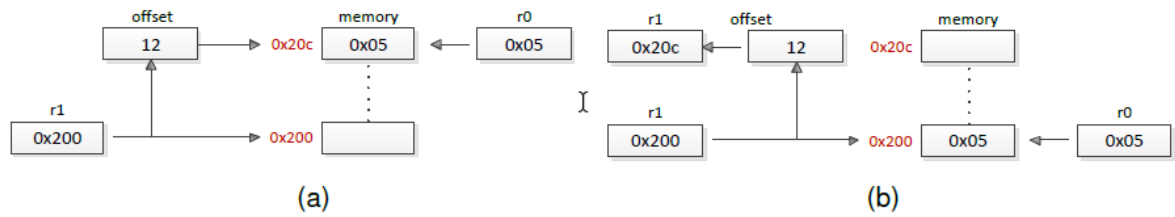


그림 3 (a) Pre-indexed Addressing, (b) Post-indexed Addressing

#### 4) Bit-banding

Bit-banding이란 메모리에서 특정 비트의 영역을 접근할 수 있도록 alias하는 것을 의미한다. 예를 들어 0x20000000번지의 1번째 bit를 0x22000000번지를 통해서도 동일하게 접근할 수 있는 것을 의미한다. 실험에서 사용하는 STM32F103RB보드에서는 메모리에서 SRAM과 Peripheral memory 두영역의 bit-banding을 제공해주고 있다. 이때 STM32F103RB 보드에서 bit-banding의 매핑 공식은 다음과 같다.

$$bit\_word\_addr = bit\_base\_band + (byte\_offset \times 32) + (bit\_number \times 4)$$

bit\_word\_addr는 타겟 비트의 alias된 주소이다.

bit\_base\_band는 alias region의 시작 주소이다.

byte\_offset은 타겟 비트가 포함된 byte가 bit\_base\_band에서 떨어진 byte수이다.

bit\_number는 타겟 비트가 해당 바이트에서 포지션이다. (0~7) 사이의 값이다.

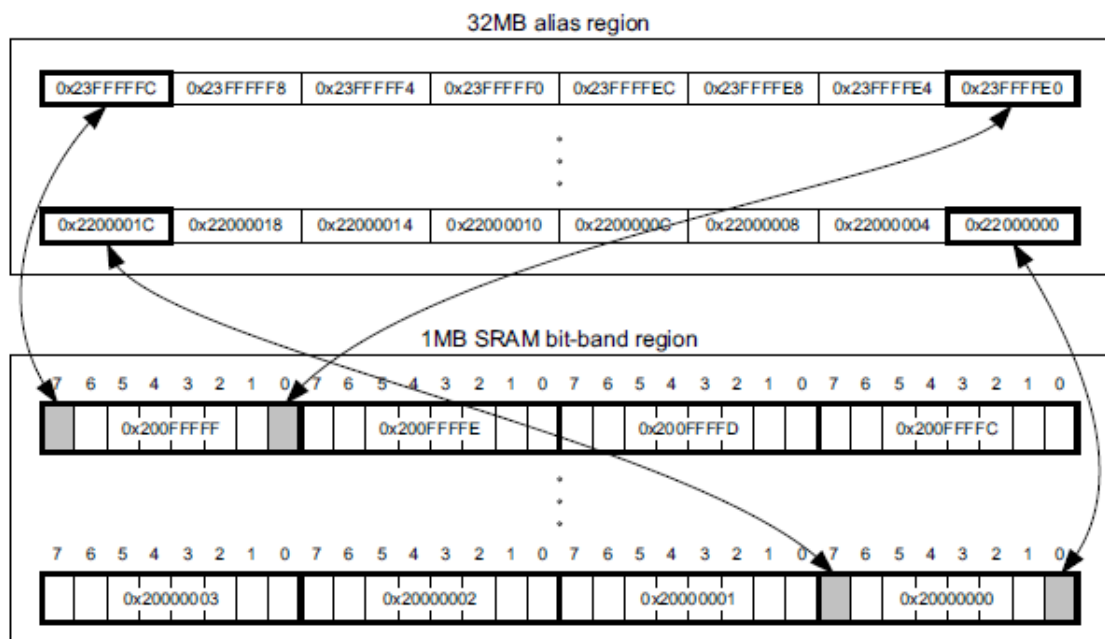


그림 4 bit-band mapping

Address range	Memory region	Instruction and data accesses
0x20000000-0x200FFFFFF	SRAM bit-band region	Direct accesses to this memory range behave as SRAM memory accesses, but this region is also bit addressable through bit-band alias.
0x22000000-0x23FFFFFFF	SRAM bit-band alias	Data accesses to this region are remapped to bit band region. A write operation is performed as read-modify-write. Instruction accesses are not remapped.

그림 5 SRAM memory bit-banding regions

Address range	Memory region	Instruction and data accesses
0x40000000-0x400FFFFFF	Peripheral bit-band region	Direct accesses to this memory range behave as peripheral memory accesses, but this region is also bit addressable through bit-band alias.
0x42000000-0x43FFFFFFF	Peripheral bit-band alias	Data accesses to this region are remapped to bit-band region. A write operation is performed as read-modify-write. Instruction accesses are not permitted.

그림 6 Peripheral memory bit-banding regions

## 5) Pseudo-instruction

Pseudo-instruction은 우리가 원하는 명령어를 직접 수행할 수 없어 어셈블러가 가능한 명령어의 조합으로 바꿔 번역하는 것을 말한다. 간단한 예를 들어 우리가 어셈블리어로 `ldr r0, 0x3456789a`라는 명령어를 작성하였을 때 원하는 명령과 데이터 `0x3456789a`를 32bit명령어에 전부 포함하는 것은 불가능하기 때문에 `0x3456789a`라는 데이터를 literal pool에 저장하고 `ldr r0, [pc, #20]`과 같이 다른 명령어로 바뀌서 번역하게 된다.

## 6) 데이터의 유형

우리가 사용하는 Cortex-M3기반 보드는 32-bit의 데이터 버스를 가지고 있다. 따라서 데이터는 8-bit인 바이트, 16-bit인 halfword, 32-bit는 word단위로 3가지의 데이터 유형만 지원한다. 다만 64-bit 데이터를 활용하는 명령어는 존재한다.

## 7) ARM assembly directive 일부

**AREA sectionname {attr}, {attr}...** : 코드나 데이터의 블록을 sectionname으로 지정해준다. attr에

는 코드 영역임을 지정하는 CODE, 데이터 영역임을 지정하는 DATA, 읽기전용인 READONLY, 읽기쓰기 가능한 READWRITE 등이 존재한다.

**name EQU expr{, type}** : 작성한 어셈블리에서 어셈블러가 name을 모두 expr로 치환해준다.

**ENTRY** : 프로그램의 시작 지점을 알려준다.

**{label} DCB expr{, expr}** : 1바이트 단위로 메모리를 할당해 데이터를 저장해준다.

**{label} DCW expr{, expr}** : 2바이트 단위로 메모리를 할당해 데이터를 저장해준다.

**{label} DCD expr{, expr}** : 4바이트 단위로 메모리를 할당해 데이터를 저장해준다.

**ALIGN {expr{, offset}}**: 이 위치 이후로 원하는 bit만큼 패딩을 넣어 align해준다.

**{label} SPACE expr** : 원하는 크기만큼 0으로 채워서 초기화한 뒤 공간을 할당해준다.

**LTORG** : 어셈블러가 literal pool을 이 directive 다음에 오도록 해준다.

**END** : 어셈블리 소스파일의 끝을 알려준다.

### 3. 실험 과정

#### 1) 실험 1

**STEP 1:** 1.5.4절과 마찬가지로 방법으로 실험을 수행하기 위한 작업 directory를 생성한다.

**STEP 2:** µVision을 통해 new project을 생성한다. 이 과정에서 23페이지의 STEP: 6에서와 마찬가지로 설정한다.

**STEP 3:** Program 3.3(lab3\_1.s)을 작성하고 앞 단계에서 생성한 프로젝트에 추가한 후 Project를 build한다.

```

1      area lab3_1,code
2      entry
3      __main proc
4      export __main [weak]
5  start ldr r0,=0x32
6        ldr r1,data1
7        adr r2,data1
8        ldrb r3,[r2,#0x04]
9        ldr r4,=data6
10       str r3,[r4]
11       ldr r5,data1
12       ldr r5,data2
13       ldr r5,data3
14       ldr r5,data4
15       ldr r5,data5
16       str r5,[r4,#0x4]
17       b .
18       endp
19       align
20  data1 dcd 0x12345678,0xbabeface
21  data2 dcb 0x12,0x34
22       align
23  data3 dcd 0x56,0x78
24  data4 dcw 0xab,0xcd
25  data5 dcb "Be Smart!"
26       align
27       area lab3data,data
28  data6 space 16
29       end

```

그림 7 직접 작성한 lab3\_1.s 코드

STEP 4: 실험의 편의를 위해 board의 연결 없이 MDK가 제공하는 simulator를 이용하여 프로그램의 debugging과 execution을 수행한다. 이를 위해서 Target 1에 포인터를 대고 마우스의 우측 버튼을 눌러 Options for Target 'Target 1'...을 선택한 후 그림 3.14의 좌측과 같이 Debug 탭의 좌측 상단의 Use simulator를 선택하고, 좌측 하단에 있는 Dialog DLL을 DARMSTM.DLL로 parameter를 -pSTM32F103RB로 각각 바꿔준다.

또한 assembler에 의해 생성되는 .lst 파일은 명령어들의 번역결과와 위치를 파악하는데 사용될 수 있는데, 이의 생성을 위해 그림 3.14의 우측과 같이 Asm 탭을 선택하고 Assmbler Option을 설정한다.

STEP 5: 작성한 프로그램의 첫 명령어인 line 5에 Breakpoint를 설정하고 이 명령어 전까지 프로그램을 수행한다. Debugging을 통해 한 명령어 씩 수행하면서 다음 단계에서 요구하는 내용을 확인한다. 이 때, Debug menu에서 Execution profiling – Show time을 선택하여 각 명령어가 수행되는데 소요되는 시간을 확인할 수 있도록 한다.

STEP 6: View menu를 통해 두 개의 Memory windows를 열고 Memory1은 0x08000140부터, 그리고 Memory2는 0x20000000부터 확인할 수 있도록 설정한다.

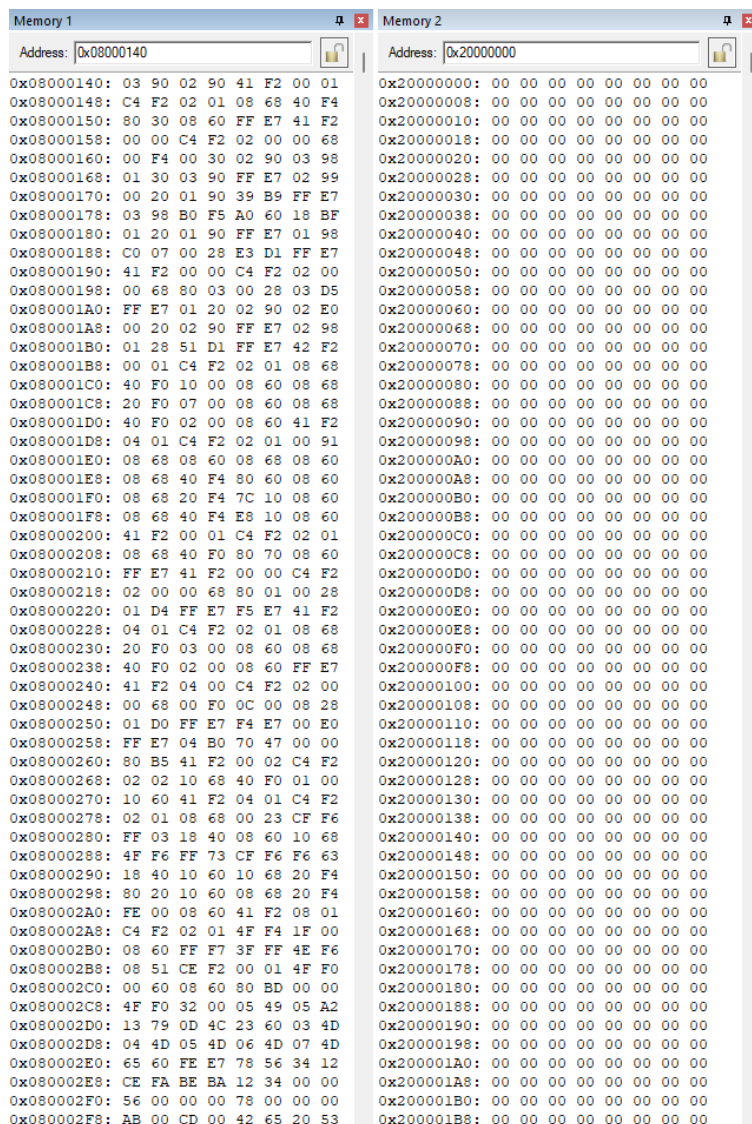


그림 8 Memory1, Memory2 window

STEP 7: Line 5 명령어를 assembler가 실제로는 MOV 명령어로 번역하였음을 확인할 수 있다. 번역된 명령어와 16진수로 표현된 machine code를 기록한다. 번역된 machine code 가운데 이 명령어의 operand에서 사용된 immediate data가 포함되어 있음을 볼 수 있는가?

```
5: start ldr r0,=0x32
0x080002C8 F04F0032 MOV r0,#0x32
```

그림 9 Line 5 disassembly

Line 5 명령어 ldr r0,=0x32는 어셈블러에 의해 MOV r0,#0x32로 번역되었다. 이의 16진수 machine

code는 F04F0032이다. 이때 immediate data 32는 MOV명령어에서 operand2로 명령어 코드에서 뒷부분에 위치한다. F04F0032에서 뒷부분 32에 operand2가 포함되어 있음을 확인할 수 있다. 이때 32앞부분 명령어의 뜻을 알아보기 위해서 destination register값을 r0에서 r8로 바뀌서 번역된 결과를 살펴보았다.

```
5: start ldr r8,=0x32
0x080002C8 F04F0832 MOV r8,#0x32
```

그림 10 ldr r8,=0x32로 바꾼 결과

그 결과 F04F0832로 명령어가 바뀐 것을 확인할 수 있다. 이를 통해 32앞의 08은 destination register값을 나타내고 있다고 유추할 수 있다.

**STEP 8: Disassembly window와 Memory1 window를 통해 data1에서 정의된 데이터들의 주소가 어떻게 배정되었는지 byte 단위로 확인한다.** 이 결과를 통해 현재 사용하고 있는 프로세서가 little endian mode에서 동작하고 있다고 볼 수 있는가? 또한, 다음 단계를 위해 data2에서 data5까지의 정의된 데이터들에 주소가 어떻게 배정되었는지 byte 단위로 확인하고 기록한다.

```
0x080002E4 5678    DCW    0x5678
0x080002E6 1234    DCW    0x1234
0x080002E8 FACE    DCW    0xFACE
0x080002EA BABE    DCW    0xBABE
0x080002EC 3412    DCW    0x3412
0x080002EE 0000    DCW    0x0000
0x080002F0 0056    DCW    0x0056
0x080002F2 0000    DCW    0x0000
0x080002F4 0078    DCW    0x0078
0x080002F6 0000    DCW    0x0000
0x080002F8 00AB    DCW    0x00AB
0x080002FA 00CD    DCW    0x00CD
0x080002FC 6542    DCW    0x6542
0x080002FE 5320    DCW    0x5320
0x08000300 616D    DCW    0x616D
0x08000302 7472    DCW    0x7472
0x08000304 0021    DCW    0x0021

0x080002E0: 65 60 FE E7 78 56 34 12
0x080002E8: CE FA BE BA 12 34 00 00
0x080002F0: 56 00 00 00 78 00 00 00
0x080002F8: AB 00 CD 00 42 65 20 53
0x08000300: 6D 61 72 74 21 00 00 00
```

그림 11 Disassembly window와 Memory1 window에서의 data 주소

주소	2E4	2E5	2E6	2E7	2E8	2E9	2EA	2EB
데이터	78	56	34	12	CE	FA	BE	BA
주소	2EC	2ED	2EE	2EF	2F0	2F1	2F2	2F3
데이터	12	34	00	00	56	00	00	00



주소	2F4	2F5	2F6	2F7	2F8	2F9	2FA	2FB
데이터	78	00	00	00	AB	00	CD	00
주소	2FC	2FD	2FE	2FF	300	301	302	303
데이터	42	65	20	53	6D	61	72	74
주소	304	305	306	307				
데이터	21	00	00	00				

표 1 data1 ~ data5의 저장된 주소

현재 프로세서가 어떤 endian 모드를 사용하고 있는 지 확인하기 위해서는 data1을 확인해보면 된다. 우리가 저장하고자 했던 데이터는 0x12345678이다. 근데 데이터에서 LSB가 포함된 78부분이 가장 낮은 주소인 0x080002E4에 저장된 것을 확인할 수 있다. 따라서 현재 little endian 모드를 사용하고 있다는 것을 확인할 수 있다. 또한 data1을 비롯해 data5까지 데이터가 저장된 주소는 위의 [그림 11]과 [표 1]에서 확인할 수 있다. data1은 0x080002E4 ~ 0x080002EB에 저장되어 있고 data2는 0x080002EC ~ 0x080002EF, data3는 0x080002F0 ~ 0x080002F7, data4는 0x080002F8 ~ 0x080002FB, data5는 0x080002FC ~ 0x08000307까지 저장되어 있다.

**STEP 9: Line 6 명령어와 line 7 명령어의 차이점을 알아본 후 수행을 통해 확인한다. r1과 r2에 저장되는 내용은 무엇을 의미하는가?**

```

6:      ldr r1,data1
0.019 us 0x080002CC 4905      LDR      r1,[pc,#20] ; @0x080002E4
7:      adr r2,data1
0.009 us 0x080002CE A205      ADR      r2,{pc}+0x18 ; @0x080002E4

```

그림 12 Line 6, 7 명령어

```

R0      0x00000032
R1      0x12345678
R2      0x080002E4

```

그림 13 Line 6, 7 실행 후 r1, r2의 값

Line 6 명령어는 ldr로 메모리에서 4바이트 데이터를 레지스터에 로드하는 명령어이고 line 7의 명령어 adr은 현재 pc-relative 주소를 계산해 레지스터에 로드하는 명령어이다. 이때 우리가 어셈블리를 작성할 때는 adr r2,data1으로 작성하였지만 어셈블러가 기계어로 번역하는 과정에서 data1을 {pc}+0x18처럼 pc-relative한 값으로 바꿔준 것을 확인할 수 있다. 따라서 line 6 명령어의 결과

data1의 값인 0x12345678이 r1에 저장되고 line 7 명령어의 결과 STEP 8에서 확인한 data1이 저장된 주소인 0x080002E4가 r2에 저장되었다.

여기서 추가적으로 두가지를 더 확인할 수 있다. 첫째, ldr은 메모리에 접근해서 값을 로드하는 명령어이고 adr은 주소 값을 계산해 로드하는 명령어이기 때문에 메모리에 접근하는 ldr 명령어는 2번의 사이클이 필요하고 adr은 1번의 사이클이면 충분하기 때문에 ldr 명령어의 수행 시간이 2배정도 긴 것을 확인할 수 있다.

두번째로 disassembly 코드에서 확인할 수 있는 [pc, #20]과 {pc}+0x18에서 각 pc값의 차이이다. 먼저 두 식이 계산된 pc-relative 주소는 모두 0x080002E4로 동일하다. 따라서 각 line에서 더해진 offset만큼을 빼면 line 6에서의 pc는 0x080002D0이고 line 7에서의 pc는 0x08002CC이라고 예상할 수 있다. 이에 대해 ARM의 3단계 파이프라인관점에서 살펴보면 line 6과 7의 명령어는 모두 2바이트로 같이 붙어서 fetch가 된다. 따라서 line 6 명령어에서의 [pc]는 다음 fetch 명령어 주소인 0x080002D0가 사용되고 line 7 명령어에서의 {pc}는 명령어가 fetch될 때의 주소인 0x080002CC가 사용되었다고 생각할 수 있다.

**STEP 10: Line 8 명령어의 수행 결과 r3에 저장되는 내용은 무엇인가? 이 명령어의 operand를 어떻게 해석할 수 있는가? 이 명령어를 수행하는데 소요된 시간을 기록한다.**

```

8:      ldrb r3, [r2, #0x04]
0.019 us 0x080002D0 7913      LDRB      r3, [r2, #0x04]
  
```

그림 14 Line 8 명령어

R0	0x00000032
R1	0x12345678
R2	0x080002E4
R3	0x000000CE

그림 15 Line 8 실행 후 r3 값

Line 8의 명령어를 해석해보면 operand인 [r2, #0x04]는 r2에 저장된 주소에서 0x04만큼 떨어진 곳이라는 뜻이고 ldrb r3는 1byte만큼을 로드해서 r3에 저장하는 명령어이다. STEP 9에서 확인하였듯이 data1의 주소가 저장되어 있는 r2값 0x080002E4에서 0x04만큼 더한 0x080002E8에서 1byte만큼 데이터를 로드한다. 그 결과 [표 1]에서 확인할 수 있듯이 0x080002E8에 저장되어 있는 CE라는 데이터를 r3에 로드한다. 이때 명령어를 수행하는데 소요된 시간은 0.019us이다.

STEP 11: Line 8 명령어를 ldrb에서 ldr로 변경하고 STEP 10 을 반복한다.

Line 8 명령어를 ldr로 두고 명령어의 operand에서 0x04를 0x01과 0x02로 변경하면서 STEP 10 각각 반복한다 r3에 저장되는 내용의 변화와 수행시간의 변경을 어떻게 설명할 수 있는가?

```
8:      ldr r3,[r2,#0x04]
0.019 us 0x080002D0 6853      LDR      r3,[r2,#0x04]
```

그림 16 ldr로 바꾼 line 8 명령어

R0	0x00000032
R1	0x12345678
R2	0x080002E4
R3	0xBABEFACE

그림 17 ldr로 바꾼 line 8 실행 후 r3 값

Line 8의 ldr 명령어의 ldrb와의 차이점은 ldr은 4바이트 데이터를 로드하는 명령어라는 점이다. Operand는 STEP 10과 동일하기 때문에 0x080002E8에서부터 0x080002EB까지의 4바이트의 데이터를 little endian 모드로 낮은 주소인 0x080002E8을 LSB쪽으로 해서 로드한다. 명령어를 수행하는데 소요된 시간은 0.019us이다.

```
8:      ldr r3,[r2,#0x01]
0.037 us 0x080002D0 F8D23001 LDR      r3,[r2,#0x01]
```

그림 18 0x01로 바꾼 line 8 명령어

R0	0x00000032
R1	0x12345678
R2	0x080002E8
R3	0xCE123456

그림 19 0x01로 바꾼 line 8 실행 후 r3 값

```
8:      ldr r3,[r2,#0x02]
0.028 us 0x080002D0 F8D23002 LDR      r3,[r2,#0x02]
```

그림 20 0x02로 바꾼 line 8 명령어

R0	0x00000032
R1	0x12345678
R2	0x080002E8
R3	0xFACE1234

그림 21 0x02로 바꾼 line 8 실행 후 r3 값

Line 8 명령어에서 operand에서 0x04를 0x01로 바꾼 결과 0x080002E5부터 0x080002E8까지의 4바이트 데이터를 로드하고 0x02로 바꾼 결과에서는 0x080002E6부터 0x080002E9까지의 데이터를 로드해서 r3에 저장한 것을 확인할 수 있다. 이때 0x01로 바꾼 명령어에서는 0.037us가 소요되고 0x02로 바꾼 명령어에서는 0.028us가 소요되었는데 0x04인 경우인 0.019us보다 각각 0.018us, 0.009us만큼 차이가 났다. 먼저 0x01과 0x02를 offset으로 사용하게 되면 주소가 4바이트 단위로

align되지 않았기 때문에 2번에 걸쳐 데이터를 로드해야 해서 한번의 사이클 시간인 0.009us의 추가 시간이 소요된 것이다. 거기다 더해 0x01의 경우는 2번에 걸쳐 로드해온 데이터를 1byte, 3byte로 unaligned된 상태에서 잘랐다 이어 붙이는 과정이 추가로 소요되기 때문에 한 사이클인 0.009us가 더 소요되었다.

따라서 정리하자면 ldr로 바꾸고 offset이 0x04인 경우는 ldrb와 동일한 시간이 소요되었고 offset이 0x02인 경우는 데이터를 2번에 걸쳐 로드하기 때문에 한번의 사이클이 더 소요되어 0.028us가 소요되고 offset이 0x01인 경우는 데이터가 unaligned되어있고 2번에 걸쳐 로드하기 때문에 2번의 사이클이 더 소요되어 0.037us라는 가장 긴 시간이 소요되었다.

**STEP 12: Line 9 명령어의 수행 결과 r4에 저장되는 내용은 무엇인가? 이 내용을 그림 3.3을 참조하여 해석해 보자. 이 때, line 1과 line 27에서 사용된 area이라는 directive의 역할과 여기서 사용된 attribute의 의미를 확인해 보자.**

```

9:      ldr r4,=data6
0.019 us 0x080002D2 4C0D      LDR      r4,[pc,#52] ; @0x08000308

```

그림 22 Line 9 명령어

R0	0x00000032
R1	0x12345678
R2	0x080002E4
R3	0x000000CE
R4	0x20000000

그림 23 Line 9 실행 후 r4 값

Line 9 명령어에서 사용된 data6는 space 16이다. 이때 space 16은 16byte에 해당하는 공간을 0으로 초기화한 뒤 할당해주라는 의미이다. 따라서 0x20000000영역에 16byte만큼 공간을 할당하고 r4에 할당된 메모리의 시작주소인 0x20000000을 로드했다. 이때 0x20000000이라는 메모리 주소의 의미를 해석하기 위해서 아래 [그림 24]를 살펴보면 0x20000000은 SRAM영역으로 읽기쓰기가 모두 가능한 데이터를 저장하는 영역이다.

그렇다면 왜 읽기쓰기가 가능한 SRAM영역에 공간을 할당해주었는지 알기 위해서는 line 1과 line 27에서 사용된 area라는 directive에 대해 알아야 한다. 우리가 작성한 코드에서 line 1에는 area lab3\_1,code라 작성되어 있다. 이는 이 directive 이후의 코드는 lab3\_1이라는 이름을 가진 코드 블록이고 code영역에 그 코드와 데이터를 저장한다는 의미이다. 그리고 line 27은 area

lab3data,data로 line 27부터 새로운 lab3data라는 이름의 코드 블록으로 이는 data영역에 저장한다는 의미이다. 따라서 code영역인 0x08000000부터 line 1부터 line 26까지의 코드가 저장되어 있고 data영역인 0x20000000부터 line 28의 데이터가 저장되어 있는 것이다.

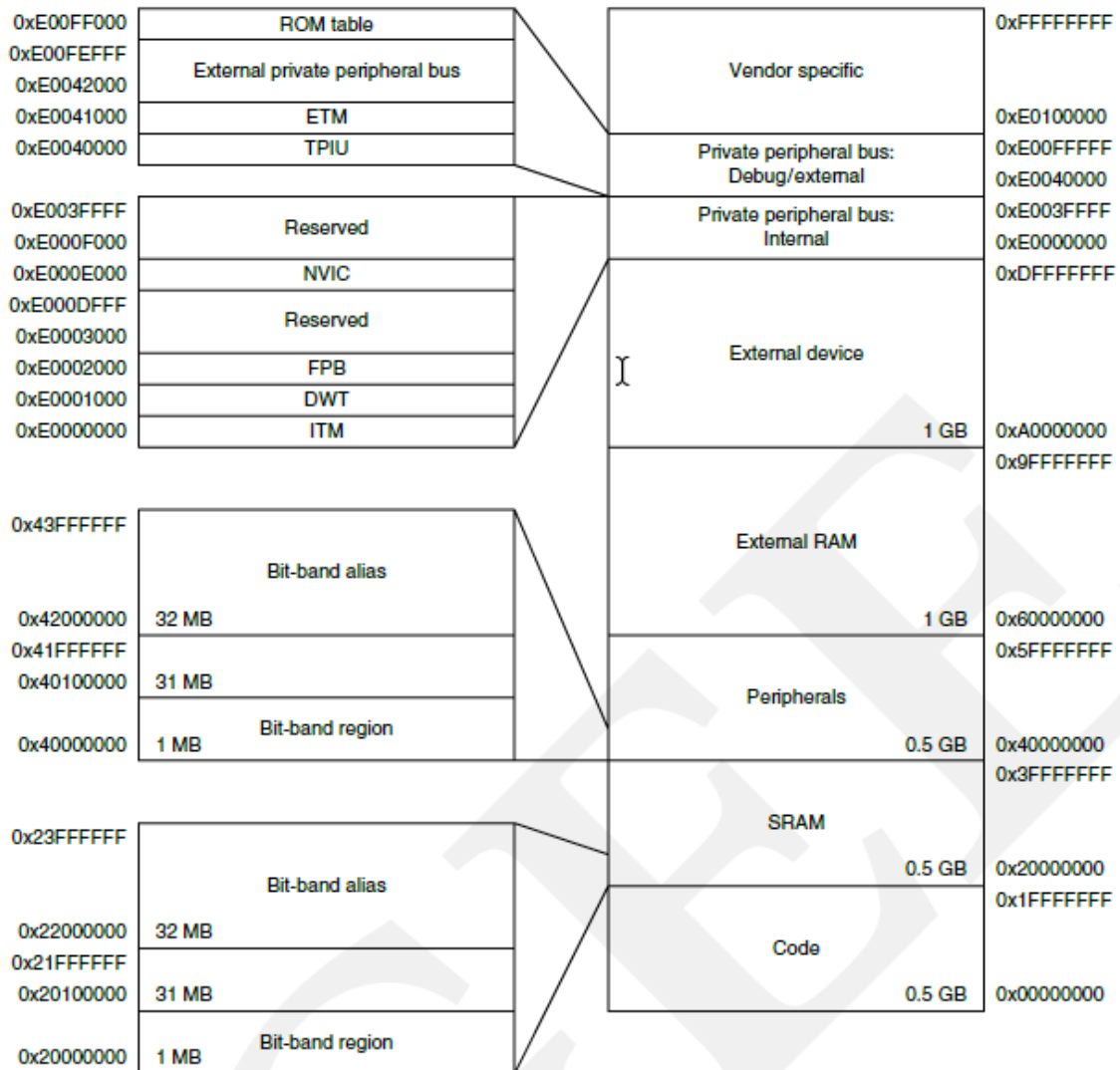


Figure 3.3: A Cortex-M3 predefined memory map(Yiu, 2010)

그림 24 STEP 12에서 참고 할 교재 그림 3.3

```

1      area lab3_1,code
2      entry
3      __main proc
4      export __main [weak]
5  start ldr r0,=0x32
6        ldr r1,data1
7        adr r2,data1
8        ldrb r3,[r2,#0x04]
9        ldr r4,=data6
10       ldr r5,=data7
11       str r0,[r5]
12       str r3,[r4]
13       ldr r5,data1
14       ldr r5,data2
15       ldr r5,data3
16       ldr r5,data4
17       ldr r5,data5
18       str r5,[r4,#0x4]
19       b .
20       endp
21       align
22  data1 dcd 0x12345678,0xbabeface
23  data2 dcb 0x12,0x34
24       align
25  data3 dcd 0x56,0x78
26  data4 dcw 0xab,0xcd
27  data5 dcb "Be Smart!"
28       align
29       area lab3data,data
30  data6 space 16
31  data7 space 16
32       end

```

그림 25 space 역할 확인을 위해 수정한 코드

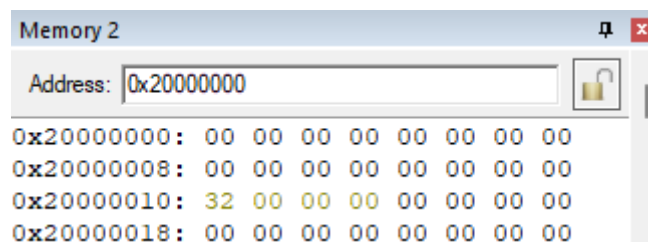


그림 26 space 역할 확인을 위한 Memory 2 window

추가적으로 space라는 directive의 역할을 좀 더 살펴보기 위해 약간의 코드를 추가해서 확인해보았다. [그림 25]는 line9, 10, 31이 새로 추가된 코드이다. 간단히 기능을 설명해보자면 data7을 통해서 새로운 16byte 공간을 할당 받고 그 주소를 r5에 저장해주었다. 그리고 r5가 가진 주소에 r0의 데이터 0x32를 저장해주었다. 그 결과 0x20000010에 32라는 데이터가 저장된 것을 확인할 수 있다. 이를 통해 0x20000000부터 0x2000000F까지 0으로 채워진 공간은 data6에 할당된 16byte

공간이라는 것을 간접적으로 확인할 수 있었다.

**STEP 13:** Line 10 명령어를 수행하기 전에 Memory2 window를 통해 보여지는 내용을 확인하고, 이 명령어를 수행한 후에는 어떻게 변하는지를 확인한다. Debugging 중인 프로그램을 처음부터 다시 수행하고 싶으면, 메뉴 중 RST라고 쓰여진 icon을 누르면 된다.

```

10:          str r3,[r4]
0.019 us 0x080002D4 6023      STR      r3,[r4,#0x00]

```

그림 27 Line 10 명령어

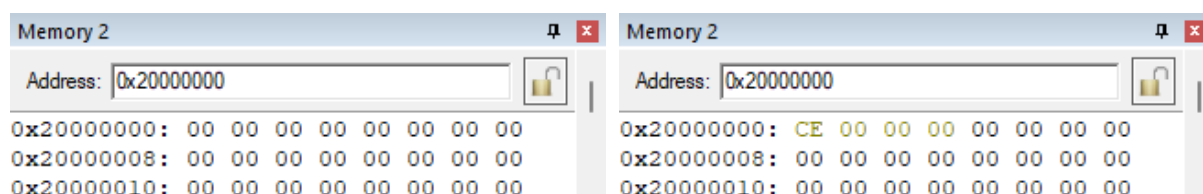


그림 28 Line 10 실행 전, 후 Memory 2 window

Line 10 명령어 실행 전에는 0x20000000부터 공간에는 0으로 초기화 되어 있다. Line 10 명령어를 해석해보면 r4에 들어있는 데이터를 주소로 해서 그 주소에 r3의 값을 저장하라는 의미이다. 따라서 line 10 명령어 수행 이후 기존 r3에 저장되어 CE라는 데이터가 0x20000000에 작성된 것을 [그림 28]을 통해서 확인할 수 있다.

**STEP 14:** Line 11부터 line 15까지 수행하면서 매번 r5의 내용을 기록한다. STEP 8에서 확인한 data1-data5의 byte 단위 데이터와 그 주소들을 참고하여 이 결과를 해석한다. Line 16을 수행한 후 Memory2 window를 통해 동작 내용을 확인한다.

```

11:          ldr r5,data1
0.019 us 0x080002D6 4D03      LDR      r5,[pc,#12] ; @0x080002E4
12:          ldr r5,data2
0.019 us 0x080002D8 4D04      LDR      r5,[pc,#16] ; @0x080002EC
13:          ldr r5,data3
0.019 us 0x080002DA 4D05      LDR      r5,[pc,#20] ; @0x080002F0
14:          ldr r5,data4
0.019 us 0x080002DC 4D06      LDR      r5,[pc,#24] ; @0x080002F8
15:          ldr r5,data5
0.019 us 0x080002DE 4D07      LDR      r5,[pc,#28] ; @0x080002FC
16:          str r5,[r4,#0x4]
0.019 us 0x080002E0 6065      STR      r5,[r4,#0x04]

```

그림 29 Line 11 ~ 16 까지의 명령어

R0	0x00000032	R0	0x00000032	R0	0x00000032
R1	0x12345678	R1	0x12345678	R1	0x12345678
R2	0x080002E4	R2	0x080002E4	R2	0x080002E4
R3	0x000000CE	R3	0x000000CE	R3	0x000000CE
R4	0x20000000	R4	0x20000000	R4	0x20000000
R5	0x12345678	R5	0x00003412	R5	0x00000056

그림 30 Line 11 실행 후 r5 값    그림 31 Line 12 실행 후 r5 값    그림 32 Line 13 실행 후 r5 값

R0	0x00000032	R0	0x00000032
R1	0x12345678	R1	0x12345678
R2	0x080002E4	R2	0x080002E4
R3	0x000000CE	R3	0x000000CE
R4	0x20000000	R4	0x20000000
R5	0x00CD00AB	R5	0x53206542

그림 33 Line 14 실행 후 r5 값    그림 34 Line 15 실행 후 r5 값

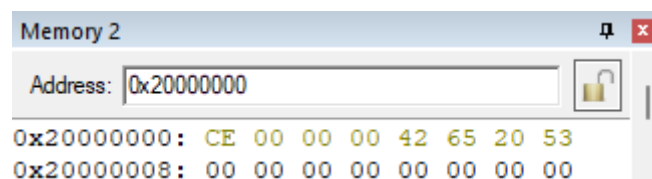


그림 35 Line 16 실행 후 memory 2 window

Line 11부터 line 15까지 모두 ldr명령어로 4바이트의 데이터를 로드하는 명령어이다. 이때 각 line 별로 실행 결과를 다음처럼 해석할 수 있다. 이 프로세스는 현재 little endian 모드를 사용하고 있기 때문에 4바이트의 데이터를 로드할 때 가장 낮은 주소의 데이터가 LSB쪽에 오도록 데이터를 로드한다. STEP 8에서 작성한 [표 1]을 참고하면 각 line을 실행하였을 때 r5에 어떤 데이터가 들어오는지 예상할 수 있다. Line 11의 경우는 data1의 시작주소인 0x080002E4 ~ 0x080002E8까지의 데이터를 로드한다. Line 12는 data2의 시작주소인 0x080002EC ~ 0x080002EF를 가져오는데 낮은 주소부터 12/34/00/00이 저장되어 있음으로 little endian으로 해석해서 00/00/34/12로 데이터를 가져온다. 비슷하게 Line 13도 data3의 시작주소인 0x080002F0 ~ 0x080002F3까지의 56/00/00/00을 뒤집어서 00/00/00/56으로 로드하고 line 14와 line 15도 각각 0x080002F8 ~ 0x080002FB, 0x080002FC ~ 0x080002FF에서 저장된 데이터를 각각 little endian 모드로 해석해 00/CD/00/AB와 53/20/65/42라는 데이터를 r5로 로드한다.

마지막으로 line 16은 r4값에 0x04를 더한 주소에 r5의 값을 저장하는 명령어이다. r4에는 0x20000000이 들어있음으로 0x04를 더한 0x20000004부터 4바이트 공간에 line 15 실행 후 r5에 저장되어 있는 값인 0x53206542의 값을 little endian 모드로 LSB를 포함하고 있는 42부분을 가장 낮은 주소인 0x20000004에 MSB를 포함하고 있는 53부분을 0x20000008에 저장한 것을 [그림 35]



를 통해서 확인할 수 있다.

STEP 15: Line 11 – line 16에서 ldr은 ldrb로, str은 strb로 변경한다. Project를 build한 후 STEP 14를 반복한다.

```

11:      ldrb r5,data1
0.019 us 0x080002D6 F89F5018 LDRB      r5,[pc,#24] ; @0x080002F2
12:      ldrb r5,data2
0.019 us 0x080002DA F89F501C LDRB      r5,[pc,#28] ; @0x080002FA
13:      ldrb r5,data3
0.019 us 0x080002DE F89F501C LDRB      r5,[pc,#28] ; @0x080002FE
14:      ldrb r5,data4
0.019 us 0x080002E2 F89F5020 LDRB      r5,[pc,#32] ; @0x08000306
15:      ldrb r5,data5
0.019 us 0x080002E6 F89F5020 LDRB      r5,[pc,#32] ; @0x0800030A
16:      strb r5,[r4,#0x4]
0.019 us 0x080002EA 7125      STRB      r5,[r4,#0x04]

```

그림 36 ldrb, strb 수정 후 line 11 ~ 16 까지의 명령어

R0	0x00000032
R1	0x12345678
R2	0x080002F0
R3	0x000000CE
R4	0x20000000
R5	0x00000078

그림 37 코드 수정 후 line 11 실행 후 r5 값

R0	0x00000032
R1	0x12345678
R2	0x080002F0
R3	0x000000CE
R4	0x20000000
R5	0x00000012

그림 38 코드 수정 후 line 12 실행 후 r5 값

R0	0x00000032
R1	0x12345678
R2	0x080002F0
R3	0x000000CE
R4	0x20000000
R5	0x00000056

그림 39 코드 수정 후 line 13 실행 후 r5 값

R0	0x00000032
R1	0x12345678
R2	0x080002F0
R3	0x000000CE
R4	0x20000000
R5	0x000000AB

그림 40 코드 수정 후 line 14 실행 후 r5 값

R0	0x00000032
R1	0x12345678
R2	0x080002F0
R3	0x000000CE
R4	0x20000000
R5	0x00000042

그림 41 코드 수정 후 line 15 실행 후 r5 값

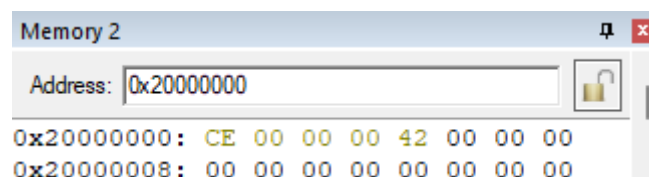


그림 42 코드 수정 후 line 16 실행 후 r5 값

코드를 ldr에서 ldrb로 str에서 strb로 수정하였을 때의 가장 큰 차이점은 데이터를 4바이트가 아

닌 1바이트 단위로 로드하고 저장한다는 것이다. 따라서 line 11 ~ line 16은 각각 data1 ~ data5가 저장되어 있는 시작주소의 첫번째 바이트만 가져와서 r5에 로드하게 된다. 다만 이번에는 ldrb의 명령어는 thumb2 명령어에서 32바이트를 차지하기 때문에 STEP 8의 [표 1]과는 다른 위치에 각 data1 ~ data5가 저장되어 있다. 따라서 아래의 [그림 43]을 참고하여 각 data의 시작주소에 접근하게 된다. 가장 먼저 line 11 실행 후에는 data1의 시작주소에 저장되어 있는 78을 r5에 로드한다. 비슷하게 line 12 실행 후에는 data2의 시작주소에 저장되어 있는 12를 r5에 로드하고, line 13 실행 후에는 data3의 시작주소에 저장되어 있는 56을 r5에 로드하고, line 14 실행 후에는 data4의 시작주소에 저장되어 있는 AB를 r5에 로드하고, 마지막으로 line 15 실행 후에는 data5의 시작주소에 저장되어 있는 42를 r5에 로드한다. 그리고 line 16 실행 후에는 STEP 14와 동일하게 0x20000004 메모리 주소에 r5가 가지고 있는 데이터에서 한 바이트에 해당하는 42를 저장해준다.

**STEP 16: Line 11 – line 16에서 ldr은 ldrh로, str은 strh로 변경한다. Project를 build한 후 STEP 14를 반복한다.**

```

11:      ldrh r5,data1
0.019 us 0x080002D6 F8BF5018 LDRH    r5,[pc,#24] ; @0x080002F2
12:      ldrh r5,data2
0.019 us 0x080002DA F8BF501C LDRH    r5,[pc,#28] ; @0x080002FA
13:      ldrh r5,data3
0.019 us 0x080002DE F8BF501C LDRH    r5,[pc,#28] ; @0x080002FE
14:      ldrh r5,data4
0.019 us 0x080002E2 F8BF5020 LDRH    r5,[pc,#32] ; @0x08000306
15:      ldrh r5,data5
0.019 us 0x080002E6 F8BF5020 LDRH    r5,[pc,#32] ; @0x0800030A
16:      strh r5,[r4,#0x4]
0.019 us 0x080002EA 80A5      STRH    r5,[r4,#0x04]

```

그림 43 ldrh, strh 수정 후 line 11 ~ 16 까지의 명령어

R0	0x00000032
R1	0x12345678
R2	0x080002F0
R3	0x000000CE
R4	0x20000000
R5	0x00005678

그림 44 코드 수정 후 line 11 실행 후 r5 값

R0	0x00000032
R1	0x12345678
R2	0x080002F0
R3	0x000000CE
R4	0x20000000
R5	0x00003412

그림 45 코드 수정 후 line 12 실행 후 r5 값

R0	0x00000032
R1	0x12345678
R2	0x080002F0
R3	0x000000CE
R4	0x20000000
R5	0x00000056

그림 46 코드 수정 후 line 13 실행 후 r5 값

R0	0x00000032
R1	0x12345678
R2	0x080002F0
R3	0x000000CE
R4	0x20000000
R5	0x000000AB

그림 47 코드 수정 후 line 14 실행 후 r5 값

R0	0x00000032
R1	0x12345678
R2	0x080002F0
R3	0x000000CE
R4	0x20000000
R5	0x00006542

그림 48 코드 수정 후 line 15 실행 후 r5 값

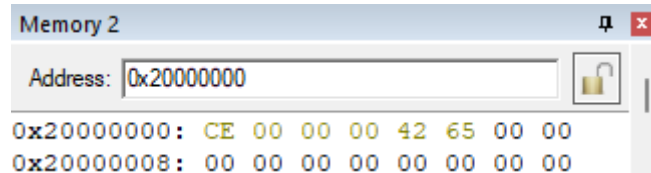


그림 49 코드 수정 후 line 16 실행 후 r5 값

STEP 16에서 사용한 `ldrh`와 `strh`는 `ldr`, `str`과 유사하지만 4바이트가 아닌 2바이트를 로드하고 저장하는 명령어이다. 따라서 line 11 명령어의 실행 결과 `data1`의 시작주소부터 78/56/34/12의 순서로 little endian 방식으로 저장되어 있으므로 2바이트만 읽어 0x5678이 r5에 로드된다. 마찬가지로 line 12 실행 후에는 `data2`의 시작주소부터 12/34 순서로 저장되어 있으므로 0x3412로 뒤집어서 r5에 로드하게 된다. 이와 같은 방식으로 line 13 실행 후에는 `data3`의 데이터 56이 r5에 로드되고, line 14 실행 후에는 `data4`의 데이터 AB가 R5에 로드되고, 마지막으로 line 15 실행 후에는 `data5`의 데이터 6542가 r5에 로드된다. 그리고 line 16 실행 후에는 STEP 14와 마찬가지로 0x20000004번지부터 2바이트의 공간에 r5의 데이터 0x6542를 little endian mode로 뒤집어서 LSB를 포함한 42가 0x20000004번지에 저장되고 65가 0x20000005번지에 저장된 것을 [그림 49]를 통해 확인할 수 있다.

**STEP 17: 위 세 단계의 반복을 통해 data에 부여되는 주소에 대한 해석, `ldr/ldrb/ldrh` 명령어의 차이, `word/halfword/byte` 단위 데이터, `dcd/dcw/dcb` directive들의 차이 등에 대해 확인한다.**

STEP 14부터 STEP 16까지 세 단계를 통해서 `ldr/ldrb/ldrh` 명령어의 차이를 알아보았다. 우선 기본적인 `ldr` 명령어는 32-bit cpu의 가장 기본적인 word인 4byte단위로 데이터를 로드해오고, `ldrb`는 byte단위로 `ldrh`는 2byte인 halfword단위로 데이터를 로드해오는 것을 알 수 있었다. 또한 `ldr`은 16-bit 명령어인데 비해서 `ldrb`와 `ldrh`는 32bit 명령어인 것도 확인할 수 있었다.

그리고 data를 저장할 때 사용하는 directive에는 `dcd/dcw/dcb`가 존재하는데 각 directive의 차이

는 어떤 데이터 단위로 잘라서 데이터를 저장할 것인지를 지시하는 directive이다. 가장 기본적인 dcd의 경우는 4byte인 word 단위로 데이터를 지정된 endian mode 형태로 저장하고 dbw는 2byte인 halfword 단위로 데이터를 저장하고, dcb는 1byte 단위로 데이터를 저장하는 것을 확인할 수 있었다.

**STEP 18:** Line 25에는 string data가 정의되었다. Memory1 window를 통해 이 string data가 어떻게 저장되었는지를 ASCII code를 참고하여 확인한다.

```
0x080002F8: AB 00 CD 00 42 65 20 53
0x08000300: 6D 61 72 74 21 00 00 00
```

그림 50 string data가 저장된 memory 1 window

주소	2FC	2FD	2FE	2FF	301	302	303	304	305
Hex	42	65	20	53	6D	61	72	74	21
ASCII	B	e	SP	S	m	a	r	t	!

표 2 string data의 hex, ascii 표현

```
0x080002F0: V...x.....Be Smart!...
0x08000308: ... .....
```

그림 51 string data가 저장된 memory1 window (ASCII)

우리가 코드에서 data5 dcb "Be Smart!"라고 작성하였기 때문에 이 문자열은 dcb의 directive에 따라서 1byte씩 잘려서 저장되게 된다. 따라서 우리가 읽기 편하도록 문자열이 뒤집히지 않고 나열되어 있다. 그리고 우리가 [그림 50]에서의 메모리 윈도우와 ASCII code를 참고하여 [표 2]처럼 저장된 데이터를 해석할 수도 있지만 [그림 51]처럼 Keil에서 지원하는 기능을 사용해서 메모리에 저장된 데이터를 ASCII 형식으로 바꿔서 편하게 볼 수도 있다.

**STEP 19:** 명령어를 한 번에 하나씩 수행하면서 Register window에서 R15(PC)의 내용이 어떻게 변경되는지 확인한다. 이 확인을 통해 PC는 다음에 수행될 명령어의 주소를 가르킨다고 일반화할 수 있는가? Line 17 명령어를 수행할 때 PC의 변화는 어떠한가? 이를 통해 line 17 명령어는 어떠한 동작을 하고 있다고 볼 수 있는가?

상태	PC
Line 5 실행 전	0x080002C8(line 5 명령어의 위치)
Line 5 실행 후 / Line 6 실행 전	0x080002CC(line 6 명령어의 위치)
Line 6 실행 후 / Line 7 실행 전	0x080002CE(line 7 명령어의 위치)
Line 7 실행 후 / Line 8 실행 전	0x080002D0(line 8 명령어의 위치)
Line 8 실행 후 / Line 9 실행 전	0x080002D2(line 9 명령어의 위치)
Line 9 실행 후 / Line 10 실행 전	0x080002D4(line 10 명령어의 위치)
Line 10 실행 후 / Line 11 실행 전	0x080002D6(line 11 명령어의 위치)
Line 11 실행 후 / Line 12 실행 전	0x080002D8(line 12 명령어의 위치)
Line 12 실행 후 / Line 13 실행 전	0x080002DA(line 13 명령어의 위치)
Line 13 실행 후 / Line 14 실행 전	0x080002DC(line 14 명령어의 위치)
Line 14 실행 후 / Line 15 실행 전	0x080002DE(line 15 명령어의 위치)
Line 15 실행 후 / Line 16 실행 전	0x080002E0(line 16 명령어의 위치)
Line 16 실행 후 / Line 17 실행 전	0x080002E2(line 17 명령어의 위치)
Line 17 실행 후	0x080002E2(line 17 명령어의 위치)

표 3 각 명령어 실행 전, 후 PC 상태

Line 5에 브레이크포인트를 설정한 상태로 디버깅 모드에서 run을 해서 line 5의 명령어가 실행되기 전 상태부터 프로그램에서 마지막 명령어인 Line 17 실행 이후까지의 모든 상태에서 PC의 값을 [표 3]에 기록해 두었다. 이를 통해서 각 상태에서의 PC가 다음에 수행될 명령어를 가르키고 있다고 일반화 할 수 있다. 다만 이번 프로그램에서 line 17의 명령어가 수행되고 나서도 PC는 변화가 없이 그대로 line 17 명령어의 위치를 가르키고 있다. 이처럼 branch 명령어가 있는 경우 등 특수한 상황에 따라서는 PC가 다음에 수행될 명령어를 가르키고 있지 않을 수 있다. 또한 line 17의 명령어는 현재의 위치로 branch 하라는 명령어로 이해할 수 있다. 이러한 명령어가 프로그램에 포함된 것은 이 프로그램에서 원하는 동작을 전부 수행한 뒤 계속해서 PC가 증가하다 보면 원하지 않는 영역의 코드도 실행할 수 있기 때문에 안전장치로 line 17과 같은 명령어를 포함시켰다.

**STEP 20:** Lines 19, 22, 26에 사용된 align이라는 directive의 역할은 무엇인가? 이 위치에 이 directive가 사용된 이유는 무엇인가? 이들 중 line 22를 comment처리하고 build한 후 수행하면 어떤 차이를 볼 수 있는가?

이번 코드에서 Line 19, 22, 26에서는 align이라는 directive가 사용되었다. 위의 이론파트에서도 설명하였지만 align이라는 directive는 원하는 bit혹은 기본적으로는 word단위로 메모리를 align 할 수 있도록 패딩을 주는 역할을 수행한다. Line 19, 22, 26에서는 혹시 모를 unaligned 데이터나 코드를 대비해서 추가해 주었다고 생각할 수 있다. 기본적으로는 어셈블러가 align을 해주기 때문에 align이라는 directive가 없더라도 동일하게 메모리에 올라가고 프로그램이 돌아간다. 이를 확인하기 위해서 line 22의 align을 주석처리를 하고 수행하면 다음 [그림 52]처럼 차이가 없는 것을 확인할 수 있다. 물론 작성한 프로그램에 따라서 align이라는 directive에 의해 차이가 생기는 경우도 있지만 이번 프로그램에서는 차이가 발생하지 않는다.

0x080002E0: 65 60 FE E7 78 56 34 12	0x080002E0: 65 60 FE E7 78 56 34 12
0x080002E8: CE FA BE BA 12 34 00 00	0x080002E8: CE FA BE BA 12 34 00 00
0x080002F0: 56 00 00 00 78 00 00 00	0x080002F0: 56 00 00 00 78 00 00 00
0x080002F8: AB 00 CD 00 42 65 20 53	0x080002F8: AB 00 CD 00 42 65 20 53
0x08000300: 6D 61 72 74 21 00 00 00	0x08000300: 6D 61 72 74 21 00 00 00
0x08000308: 00 00 00 20 00 00 00 00	0x08000308: 00 00 00 20 00 00 00 00

그림 52 (좌) 기존 memory window (우) 코멘트 후 memory window

## 2) 실험 2

**STEP 21:** Program 3.4(lab3\_2.s)를 작성한 후 project를 build한다.

```

1  sram_base equ 0x20000000
2  bb_alias equ 0x22000000
3      area lab3_2,code
4      entry
5  __main proc
6      export __main [weak]
7  start
8      ldr r0,=bb_alias
9      ldr r4,=sram_base
10     ldr r3,cmask
11     ldr r1,cdata
12     and r1,r3
13     str r1,[r4]
14 ;
15     ldr r1,cdata
16     str r1,[r4]
17     mov r5,#0
18     str r5,[r0,#0x2c]
19 ;
20     ldr r6,=0xbabeface
21     b .
22 ; ltorg
23 cdata dcd 0x34567890
24 cmask dcd 0xffffffff
25 endp
26 align
27 area lab3data,data
28 sdata space 0x100
29 end

```

그림 53 직접 작성한 lab3\_2.s 코드

STEP 22: Line 1과 line 2에서 사용된 equ라는 directive의 역할이 무엇인지 프로그램을 통해 이해해 본다.

```

8:      ldr r0,=bb_alias
0x080002C8 F04F5008 MOV     r0,#0x22000000
9:      ldr r4,=sram_base
0x080002CC F04F5400 MOV     r4,#0x20000000

```

그림 54 Line 8, 9의 disassembly

Line 1과 line 2에서 사용된 equ라는 directive의 역할을 알아보기 위해 프로젝트를 build해서 line 8과 line 9의 disassembly 결과를 보면 [그림 54]와 같이 나온다. 여기서 주목해야 할 점은 우리가 작성한 bb\_alias와 sram\_base라는 것이 우리가 위에서 equ와 같이 적어주었던 숫자로 치환되었다는 점이다. 이미 이론에서도 살펴보았지만 equ는 equ 앞에 이름을 뒤에 나오는 표현으로 치환해주는 역할을 하는 directive이다. 즉 우리가 잘 아는 C언어에서 #define과 같은 역할을 한다고 생각할 수 있다. 이를 통해서 equ는 예를 들면 0x20000000이라는 의미 없어 보이는 숫자에

sram\_base라는 이름을 붙여 코드의 가독성을 높여주는 역할을 할 수 있고 만약 다른 보드에서 다른 memory map을 가지고 있다면 sram\_base에 해당하는 숫자만 바꿔서 유지보수도 더 편하도록 만들어주는 유용한 기능을 하는 directive이다.

**STEP 23: Line 8 – line 11까지 명령어들을 수행하면서 operand에서 지정된 각 레지스터들에 저장되는 내용을 확인한다.**

```

8:      ldr r0,=bb_alias
0x080002C8 F04F5008 MOV      r0,#0x22000000
9:      ldr r4,=sram_base
0x080002CC F04F5400 MOV      r4,#0x20000000
10:     ldr r3,cmask
0x080002D0 4B06      LDR      r3,[pc,#24] ; @0x080002EC
11:     ldr r1,cdata
0x080002D2 4905      LDR      r1,[pc,#20] ; @0x080002E8

```

그림 55 Line 8 ~ line 11 명령어

R0	0x22000000
R1	0x34567890
R2	0x40021000
R3	0xFFFFF7FF
R4	0x20000000

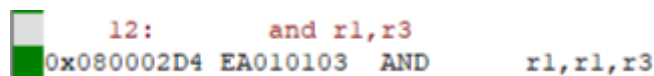
그림 56 Line 8 ~ line 11 수행 후 레지스터 값

STEP 22에서 살펴보았듯 line 8의 bb\_alias와 line 9의 sram\_base는 미리 지정해둔 숫자로 치환되어 명령어가 번역되었다. 그리고 ldr명령어 대신 mov로 바뀌어 immediate 숫자를 각각 r0에 0x22000000이 로드되고 r4에 0x20000000이 로드되었다. 이때 레지스터에 저장하려는 숫자가 4byte임에도 pseudo-instruction이 사용되지 않았는데 이 이유는 0또는 1일 많이 포함된 경우에는 32bit mov명령어에 포함되어 나타낼 수 있기 때문이다. 그리고 line 10과 line 11에서는 각각 cmask값인 0xfffff7ff는 r3에 로드되고, cdata값인 0x34567890는 r1에 로드되었다. 모든 결과는 [그림 56]에서 확인할 수 있으며 r2에는 우리가 어떠한 값도 로드하지 않았기 때문에 시스템이 시작되는 과정에서 들어간 값이 들어있다.

**STEP 24: Line 12의 동작은 두 operands를 논리적으로 AND 한 후 그 결과를 r1에 저장하는 명령어이다. 그 결과를 기록한다. cmask에서 정의된 데이터의 내용을 AND 명령어와 연결해서 이와 같은 동작이 갖는 의미를 생각해본다. (이와 같은 동작을 일반적으로 mask operation이라고 한다.)**



또한 Disassembly window를 통해서 이 명령어가 AND r1,r1,r3와 같이 번역된 것에 주목한다. 이러한 표기가 UAL(31페이지 참조)과 관련이 있다.



```

12:      and r1,r3
0x080002D4 EA010103 AND      r1,r1,r3
  
```

그림 57 Line 12 명령어

R0	0x22000000
R1	0x34567090
R2	0x40021000
R3	0xFFFFF7FF

그림 58 Line 12 실행 후 r1 값

먼저 line 12의 명령어는 r1의 값과 r3의 값을 논리적으로 and하고 그 결과를 r1에 저장하는 것이다. 그 결과 레지스터 r1에는 0x34567090이 저장되었다. 이러한 결과가 나온 것은 다음과 같다. 먼저 기존 r1에 저장되어 있던 cdata값 0x34567890을 2진법으로 나타내면 0011 0100 0101 0110 0111 1000 1001 0000이다. 그리고 r3에 저장되어 있던 cmask값 0xffff7ff를 2진법으로 나타내면 12번째 비트만 0이고 나머지는 1이다. 이를 바탕으로 두 숫자를 and하면 12번째 비트만 0이 되고 나머지는 기존 cdata의 비트가 유지된다. 그 결과를 다시 2진수로 적어보면 0011 0100 0101 0110 0111 0000 1001 0000이 되고 이를 16진법으로 나타내면 0x34567090으로 위에 저장되어 있는 r1값이 정확히 나오는 것을 알 수 있다. 이러한 계산 기법을 mask operation 또는 masking이라고 하는데 원하는 특정 비트만 0또는 1로 바꿔주고 나머지 bit는 바꾸지 않는 계산 기법을 의미한다. 임베디드 시스템에서는 특정 포트가 열려있나 닫혀있나 확인하는 등 다양한 장치들의 상태를 확인할 때 사용하거나 일반적인 소프트웨어에서 bitmap과 같은 자료구조를 사용할 때 이러한 masking 기법이 중요하게 사용된다.

추가적으로 어셈블리에서 and r1, r3라고 작성한 프로그램을 어셈블러가 번역한 내용을 disassembly window를 통해서 보면 and r1,r1,r3로 약간 바뀌어 번역된 것을 확인할 수 있다. 이는 thumb-2 명령어 체계가 만들어지면서 기존의 명령어 체계와의 호환성을 위해 새로 만들어진 UAL(unified assembler language)명령어의 일종이다. 따라서 우리가 thumb 명령어 같이 이전의 명령어를 사용하더라도 어셈블러가 자동으로 UAL 명령어 체계로 바뀌서 번역해준다. 보통은 자동으로 유리한 방향으로 번역을 해준다.

STEP 25: Line 13 명령어를 통해 메모리에 저장된 내용을 0x20000000번지부터 볼 수 있도록 설정된 memory window를 통해 확인한다.

```

13:      str r1,[r4]
14: ;
0x080002D8 6021      STR      r1,[r4,#0x00]

```

그림 59 Line 13 명령어

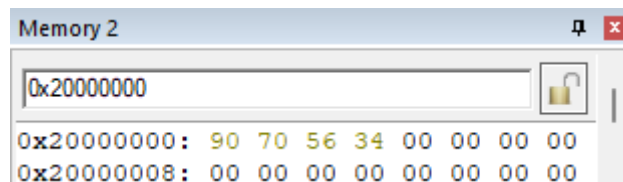


그림 60 Line 13 실행 후 0x20000000 memory 값

Line 13의 명령어를 해석해보면 r1에 있는 데이터 4바이트를 r4에 저장된 값을 주소로 하는 메모리에 저장하라는 명령어이다. STEP 24에서 보았듯 r1에는 마스킹한 데이터 0x34567090이 들어있고 r4에는 sram\_base인 0x20000000이 들어있다. 따라서 0x20000000에 0x34567090을 little endian mode로 저장을 해주면 거꾸로 90/70/56/34가 들어가게 되고 이를 [그림 60]에서 확인할 수 있다.

STEP 26: Line 15 – line 18은 위 동작을 bit-band operation을 통해 수행할 수 있음을 보여준다.

Line 15 – line 16의 동작을 통해 메모리에 0x20000000번지에 저장된 내용을 확인한다.

```

15:      ldr r1,cdata
0x080002DA 4903      LDR      r1,[pc,#12] ; @0x080002E8
16:      str r1,[r4]
0x080002DC 6021      STR      r1,[r4,#0x00]

```

그림 61 Line 15 ~ line 16 명령어

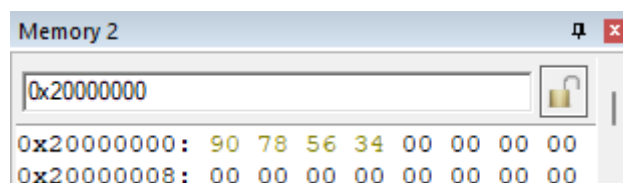


그림 62 Line 15 ~ line 16 실행 후 0x20000000 memory 값

먼저 line 15와 line 16 명령어는 기존의 cdata를 0x20000000 메모리에 다시 쓰는 과정이다. 이 결과는 [그림 62]에서 보이는 대로 다시 기존의 cdata가 저장되어 있는 것을 확인할 수 있다.

**STEP 27: Line 17 – line 18의 명령어는 데이터 0이 쓰여지는 주소는 몇 번지인가? 61페이지에서 설명된 내용을 통해 확인해 본다.**

```

17:      mov r5,#0
0x080002DE F04F0500 MOV      r5,#0x00
18:      str r5,[r0,#0x2c]
19: ;
0x080002E2 62C5      STR      r5,[r0,#0x2C]

```

그림 63 Line 17 ~ line 18 명령어

그리고 우선 line 17의 명령어는 간단하게 0을 r5에 저장하는 명령어이다. 이 프로그램 입장에서는 기존에도 r5에는 0이 저장되어 있었으므로 필요 없는 과정이라고 생각될 수 있지만 혹시 모를 상황을 대비하기 위해서 확실히 r5를 0이라는 값으로 채워서 초기화 해주어야 한다. 그리고 line 18명령어는 r5의 값 0을 r0에 들어있는 0x22000000에서 0x2c만큼 떨어진 곳에 저장하는 명령어이다. 즉 0x2200002C라는 주소에 0이라는 데이터를 저장해주는 것이다.

그렇다면 0x2200002C가 어떤 역할을 하는 주소인지 살펴보면 다음과 같다. 먼저 0x22000000부터는 위쪽의 [그림 24]에서 확인할 수 있듯이 bit banding을 위한 bit alias 영역이다. 0x2200002c가 어떤 bit를 alias하고 있는지 살펴보기 위해 이론 부분에서 언급했던 식을 이용해서 구해주면  $bit\_word\_addr = 0x2200002c$ ,  $bit\_base\_band = 0x22000000$ ,  $byte\_offset = 1$ ,  $bit\_number = 3$ 이 나오게 된다. 즉 0x20000000에서 1바이트 offset인 0x20000001의 4번째 비트를 0으로 바꿔준다는 뜻이다.

**STEP 28: Line 18 명령어를 수행 한 후 메모리의 0x20000000번지에 저장된 내용을 확인한다. 이 내용을 STEP: 26에서 기록한 내용과 비교한다. 또한 이 결과를 STEP: 25에서 기록한 내용과도 비교한다. 비교를 통해 mask operation에 비해 bit-band operation이 갖는 장점을 확인한다.**

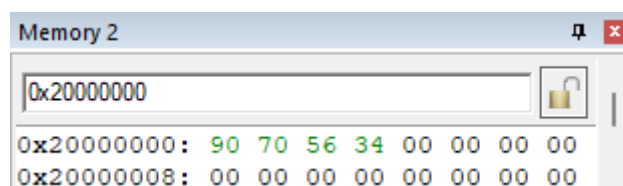


그림 64 Line 17 ~ line 18 실행 후 0x20000000 memory 값

Line 18을 수행한 뒤 0x20000000에 저장된 데이터를 memory window를 통해 살펴보면 0x34567090인 것을 확인할 수 있다. 이를 분석해보면 위의 STEP 24에서 보았듯 cdata를 2진법으로 나타내면 0011 0100 0101 0110 0111 1000 1001 0000이고 메모리는 little endian mode이기 때문에 LSB부터 출발해서 찾아보면 오른쪽에서 12번째 비트를 0으로 바꿔주는 역할을 하게 된다. 이를 0으로 바꾼 값은 0011 0100 0101 0110 0111 0000 1001 0000이고 16진법으로 쓰면 다시 0x34567090이 되게 된다.

즉 line 18 명령어를 수행하여 bit alias에 접근해서 원하는 bit를 바꿔주었더니 원하는 비트만 masking한 효과를 똑같이 낼 수 있었다. Line 18의 명령어를 통해서 STEP 26에서 cmask와 cdata를 masking해서 구한 결과와 동일한 결과를 낼 수 있었다. 이번에 수행한 bit-band operation은 STEP 26의 mask operation에 비해서 가장 큰 장점은 필요한 명령어의 수가 적다. STEP 26의 mask operation은 4개의 명령어가 필요한데 비해서 bit-band operation은 2개의 명령어만 필요했다. 따라서 소요시간도 적을뿐더러 복잡한 시스템이 되었을 때 인터럽트가 언제든지 발생할 수 있는데 mask operation의 경우는 bit를 바꾸는 과정에 여러 명령어가 소요되어 중간에 인터럽트가 걸려서 방해받거나 오류가 생길 가능성이 더 커지는데 반해 bit-band operation은 사실상 bit를 바꾸는 명령어는 1개뿐이기 때문에 이러한 걱정을 할 필요가 없다.

**STEP 29:** STEP: 28까지 이해한 내용과 61페이지부터 설명된 내용을 바탕으로 메모리의 0x20000005번지 byte data의 네 번째 bit (즉, LSB를 bit 0이라고 하기 때문에 bit 3에 해당)를 1로 설정하도록 프로그램을 변경해 본다.

```

19:          mov r6,#1
0x080002E4 F04F0601 MOV      r6,#0x01
20:          str r6,[r0,#0xac]
21: ;
0x080002E8 F8C060AC STR      r6,[r0,#0xAC]

```

그림 65 새로 작성한 코드

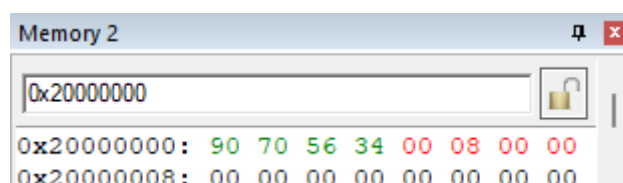
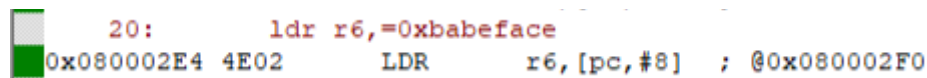


그림 66 새로 작성한 코드 memory 결과

위의 이론파트에서 설명한 공식에 대입을 하면 우리가 원하는 비트에 alias된 주소를 구할 수 있다. 주소 =  $0x22000000 + (5 \times 32) + (3 \times 4) = 0x22000000 + 0xAC$ 가 된다. 따라서 이를 바탕으로 해당 bit에 1을 작성하는 코드를 작성하기 위해서는 먼저 `mov r6,#1`로 비어 있는 r6 레지스터에 1을 넣어주고 `str r6,[r0,#0xac]`를 하면 된다. 그러면 r6에 들어있던 1이 r0인 0x22000000에 0xac를 더한 주소에 저장된다. [그림 65]를 통해 작성한 코드를 직접 확인할 수 있고 그 결과는 [그림 66]에서 확인할 수 있다. [그림 66]에서 왼쪽 상단 주소가 0x20000000임으로 0x20000005는 제일 위에 줄의 왼쪽에서 6번째 공간이다. 해당 주소의 4번째 비트를 바꿔주면 데이터는 00001000가 되고 16진법으로 나타내면 0x08이 되고 [그림 66]에서도 동일하게 제일 위에 줄 왼쪽에서 여섯 번째 숫자가 08로 적혀 있는 것을 확인할 수 있다.

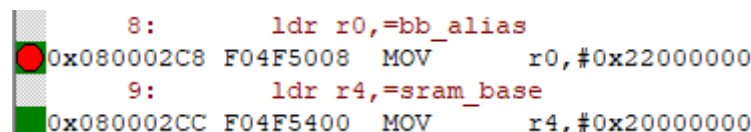
**STEP 30:** Line 20번 명령어는 pseudo-instruction(67페이지 참조)을 설명하는데 사용된다. 이 명령어가 line 8, line 9에서 사용된 명령어와 문법적으로 어떻게 다른가? Disassembly window를 통해 이 명령어의 assemble결과와 line 8, line 9 명령어의 assemble 결과와 비교해 본다. 또한 STEP: 7의 해석 결과와도 비교해 본다.



```

20:      ldr r6,=0xbabeface
0x080002E4 4E02      LDR      r6,[pc,#8] ; @0x080002F0
  
```

그림 67 Line 20 명령어



```

8:      ldr r0,=bb_alias
0x080002C8 F04F5008  MOV      r0,#0x22000000
9:      ldr r4,=sram_base
0x080002CC F04F5400  MOV      r4,#0x20000000
  
```

그림 68 Line 8 ~ line 9 명령어

Line 20과 line 8, line 9 모두 `ldr, rd,data`의 형식으로 작성을 하였으나 [그림 67]에서처럼 line 20은 pc-relative 주소를 기반으로 ldr명령어로 바뀌었고 [그림 68]에서 처럼 line 8, line 9의 명령어는 immediate 숫자를 바로 mov 명령어를 통해서 레지스터에 로드했다. 이러한 이유는 line 20에서 사용한 0xbabeface라는 32-bit 숫자를 다 포함해서 32-bit 명령어 내부에 저장할 수 없기 때문에 literal pool이라는 상수를 저장하는 공간을 만들어서 해당 주소에 0xbabeface라는 데이터를 넣어 두고 로드해오는 pseudo-instruction 방식을 취했다. Pseudo-instruction이란 우리가 작성한 명령어를 직접 수행할 수 없어 어셈블러가 가능하도록 새로 번역해서 만들어준 명령어를 의미한다. 반

면 STEP 7에서는 line 8, line 9와 유사하게 mov 명령어 내부에 데이터를 포함시킬 수 있어서 literal pool에 저장하고 참조하는 형식이 아닌 바로 명령어를 통해서 데이터를 레지스터에 로드했다.

**STEP 31: Line 20 명령어의 assemble 결과인 LDR r6, [pc, #8]을 해석해 본다. 이 때, Disassembly window를 통해 cmask로 정의 된 데이터 다음 위치에 source program에서 명시적으로 정의하지 않았던 데이터(0xBABEFACE)가 assembler에 의해 새롭게 추가된 사실에 주목한다.**

ARM/Thumb 명령어 체계 통해 표현할 수는 없으나 이와 같은 변경을 통해 수행할 수 있는 명령어를 pseudo-instruction이라고 한다.

0x080002E8	7890	DCW	0x7890
0x080002EA	3456	DCW	0x3456
0x080002EC	F7FF	DCW	0xF7FF
0x080002EE	FFFF	DCW	0xFFFF
0x080002F0	FACE	DCW	0xFACE
0x080002F2	BABE	DCW	0xBABE

그림 69 0x080002F0의 데이터

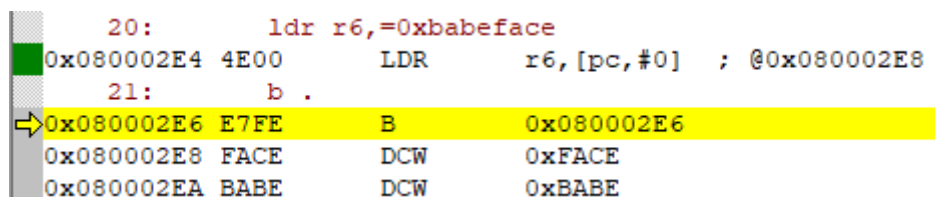
[그림 67]에서 볼 수 있듯이 line 20의 명령어는 ldr r6,[pc#8]으로 번역되었다. 먼저 [pc,#8]를 계산하면 0x080002F0가 된다. 즉 0x080002F0의 4바이트 데이터를 r6로 로드하는 명령어이다. 0x080002F0위치에 저장되어 있는 4바이트 데이터는 [그림 69]에서 볼 수 있듯이 0xbabeface이다. 이 위치를 살펴보면 우리가 저장한 데이터 cdata와 cmask 바로 다음에 우리가 정의하지 않았던 0xbabeface가 들어있는 것이다. 이는 STEP 30에서도 설명했지만 pseudo-instruction 사용을 위해 어셈블러가 만든 literal pool영역의 데이터이다.

**STEP 32: Line 20 명령어를 수행하는 과정에서 PC-relative addressing이 어떻게 이뤄지고 있는지 확인한다.**

Line 20 명령어에서 어셈블러는 ldr r6,[pc#8]로 pc-relative addressing을 사용하고 있다. Line 20 명령어의 주소는 [그림 67]에서 확인 가능하듯이 0x080002E4 이때의 pc는 0x080002E8이 된다. 그리고 여기서 8을 더해주면 0x080002F0가 되고 이 주소에 [그림 69]에서 보듯이 0xbabeface라는 데이터가 들어있다. 즉 pc-relative addressing을 통해서 명령어가 실행될 때 pc에서 8만큼 떨어진

거리의 데이터를 참조해서 사용한다. 이는 어셈블러가 어셈블리를 번역하는 2번에 걸쳐 번역하면서 명령어와 데이터의 위치를 파악하고 직접 계산해서 연결해주게 된다.

**STEP 33:** Line 22에서 comment처리를 위한 semicolon을 제거한 후 프로젝트를 build한다. Line 20번 명령어가 어떻게 assemble되었는지를 확인하고 위 과정과 비교한다. 이를 통해 **ltorg directive**의 역할을 이해한다.



```

20:      ldr r6,=0xbabeface
0x080002E4 4E00      LDR      r6,[pc,#0] ; @0x080002E8
21:      b .
⇒0x080002E6 E7FE      B        0x080002E6
0x080002E8 FACE      DCW      0xFACE
0x080002EA BABE      DCW      0xBABE
  
```

그림 70 Line 22 추가 후 disassembly 결과

Line 22에 주석을 없애고 ltorg directive를 포함해서 새로 빌드하고 결과를 살펴보면 [그림 70]과 같다. 기존 line 20의 어셈블러의 해석 결과는 ldr r6,[pc#8]였는데 ldr r6,[pc,#0]으로 바뀌었다. 즉 0xbabeface라는 데이터가 저장된 literal pool의 위치가 바뀐 것이다. 이는 ltorg directive때문인데 ltorg가 추가됨으로써 literal pool이 line 22에 붙어서 오도록 된 것이다. 즉 ltorg directive는 literal pool의 위치를 명령어가 접근가능한 영역이 되도록 가까이 가져오는 역할을 하는 directive이다. 이러한 directive가 필요한 것은 명령어와 literal pool사이의 거리가 너무 멀면 pc-relative addressing으로 접근할 수 없기 때문에 큰 프로그램의 경우 literal pool을 접근가능 하도록 가까이 위치시키는 것이 중요하다. 이러한 역할을 위해서 ltorg라는 directive가 사용된다.

## 4. Exercises

**1) 메모리 소자(SRAM)의 access cycle과 프로세서의 bus cycle에 대해 알아보자. 메모리와 프로세서를 연결할 때 timing관점에서 고려해야 할 사항은 무엇인가?**

SRAM은 읽기와 쓰기를 할 때 access cycle이 있다. 이는 간단히 요약하면 메모리에 접근하기 위해서는 다양한 신호가 들어와 메모리를 읽고 쓰는데 전기적인 신호이다 보니 각 신호가 들어오고 나서 안전하게 값을 보장하기 위해서 필요한 최소한의 시간이 있다. 이를 바탕으로 모든 신호가 안정되었을 때 데이터를 읽거나 쓸 수 있다.

다음으로 프로세서의 bus cycle은 프로세서가 다른 메모리나 장치들과 연결되었을 때 버스를 통해서 데이터나 주소, 신호 등을 이동한다. 아래 [그림 71]은 ARM7TDMI 기술 문서에서 가져온 것으로 MCLK라는 클럭에 따라서 주소와 데이터 중 어떤것을 버스에 통과시킬지 결정해서 이동시켜 주는 간단한 프로세서의 bus cycle 예시이다.

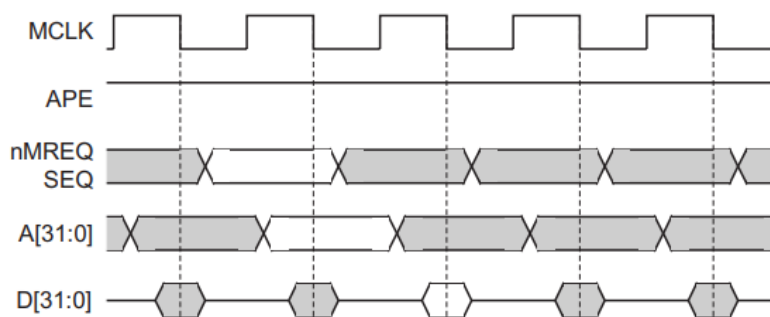


그림 71 간단한 bus cycle

그리고 이 프로세서와 메모리를 연결할 때 timing관점에서 고려해야 할 점으로는 프로세서와 메모리가 데이터의 전송에 필요한 최소한의 시간을 맞춰주어야 한다는 것이다. SRAM의 속도가 더 느리기 때문에 SRAM의 속도에 맞춰서 데이터를 이동시켜야 한다. 또한 당연한 이야기지만 두 장치의 클럭을 동기화 시켜주어야 하는 것도 매우 중요하다. 동기화 되어 있어야 같은 타이밍에 서로 데이터를 주고 받을 수 있기 때문이다.

## 2) 특정 마이크로컨트롤러의 memory map을 통해 assembly 프로그램 작성자 관점에서 확인해야 사항에는 어떤 것들이 있다고 생각하는가?

새로운 마이크로컨트롤러의 memory map을 보고 가장 먼저 확인해야 하는 것은 메모리의 각 영역이 어떤 주소영역을 얼마만큼 가지고 있는지 확인해야 한다. 예를 들어 ROM형태로 되어있는 주소는 어디인지, read/write 모두 가능한 RAM영역은 어디인지, 외부 장치를 연결했을 때 할당되는 주소를 확인하는 등 주소 영역별로 메모리의 역할을 확인해야 한다. 또한 memory map에 적혀 있는 인터럽트 벡터의 주소를 확인해서 인터럽트가 발생하였을 때 어떤 주소로 이동하는지 참고하여 어셈블리어를 작성해야 한다.



### 3) 프로세서의 endian mode가 주변장치/소자의 endian mode와 다를 경우에는 어떠한 문제가 발생할 수 있겠는가? 이에 대한 대책은 무엇이 있겠는가?

프로세서와 다른 endian mode를 사용하는 장치와 연결하게 되면 가장 큰 문제는 데이터를 이동시킬 때 잘못된 데이터를 이동시킬 수 있다는 점이다. 예를 들어 little endian mode로 0x12345678이라는 4바이트 데이터가 낮은 주소부터 78/56/34/12가 저장되어 있었다고 할 때 이를 big endian mode를 사용하는 장치가 4바이트를 읽으면 낮은 주소를 MSB방향으로 읽어 들이기 때문에 0x78563412로 데이터를 잘못 읽게 된다.

이에 대한 대책으로는 소프트웨어적으로 해결할 수도 있고 하드웨어적으로 해결할 수도 있다. 먼저 하드웨어적인 해결방법으로는 endian mode를 변환해주는 장치를 만들어서 붙여주면 필요할 때 마다 이 장치를 이용해 자동으로 endian mode를 알맞게 바꿔 데이터를 이동시킬 수 있다. 소프트웨어적인 해결방법으로는 1바이트씩 데이터를 읽어 들여 순서를 바꿔주어 직접 데이터를 자신의 endian 모드에 맞게 맞춰서 데이터를 사용하면 된다. 또는 데이터에 endian 모드와 관련한 데이터를 포함시켜 이동시킨 뒤 endian모드를 호환되도록 데이터를 처리한 뒤 사용하면 된다. 물론 가장 근본적인 해결책은 미리 프로세서와 장치의 endian을 확인하고 동일한 endian을 사용하는 장치를 사용하는 것이 가장 좋다.

### 4) 일정 크기를 갖는 데이터의 주소가 어떻게 배정되는가에 따라 align 또는 misalign되었다고 표현하는데, 데이터의 align이 이슈가 되는 이유는 무엇인가? 프로세서의 성능에 어떻게 영향을 미치는가?

메모리에서 데이터에 접근할 때 1byte씩 접근하는 것이 아니라 word단위로 데이터에 접근하게 된다. 이때 데이터가 align되어 있어야만 원하는 데이터를 한 번에 가져올 수 있다. 32-bit 프로세서를 예를 들자면 32-bit 데이터는 주소가 4의 배수에 위치해야 한 번에 데이터를 불러들일 수 있다. 또한 데이터가 16-bit인 경우에는 주소가 2의 배수에 위치해야 한 번에 데이터를 불러올 수 있다. 만약 데이터가 misalign되어있는 경우 원하는 데이터를 한 번에 가져오지 못하고 두 번에 나눠서 데이터를 가져와야 하고 그렇게 되면 프로세서가 수행해야 할 명령어가 늘어남으로 프로세서의 성능이 저하된다. 따라서 데이터가 align되어 있는 것은 매우 중요하다.

## 5) 실험과정 중 두 프로그램에서 사용된 데이터 이동 명령어들을 3.1절에 나열한 4가지 유형으로 분류해 보자.

이번 실험과정에서 사용한 데이터 이동 명령어의 리스트는 ADR, LDR, LDRB, LDRH, STR, STRB, STRH가 있다. 이 명령어들을 3.1절에 나열한 4가지 유형으로 분류하면 다음과 같다.

- 레지스터간 데이터 이동 : MOV
- 레지스터와 메모리간 데이터 이동 : LDR, LDRB, LDRH, STR, STRB, STRH
- 특수목적 레지스터와 일반 레지스터간 데이터 이동 : 없다.
- 레지스터에 직접적인 데이터 값의 저장 : ADR, MOV

## 6) 프로그램이 location independent하게 동작한다는 것을 PC-relative addressing과 연관지어 이해해보자.

프로그램이 location independent하게 동작한다는 것은 프로그램의 코드 또는 데이터가 메모리에 저장되어 있는 위치와 관련 없이 동작할 수 있다는 것을 의미한다. 물론 프로그램 수행 중 write 할 수 없는 ROM영역에 변수를 넣어두고 변수 값을 바꾸려는 물리적으로 불가능한 경우는 생각하지 않는다. ARM 어셈블리에서 pc-relative addressing이 가능하기 때문에 지금 실행하는 명령어가 어떤 위치에 있더라도 현재의 pc에서 떨어진 만큼 offset을 계산해서 데이터가 있는 곳을 접근할 수 있다. 따라서 코드와 데이터가 위치와 상관없이 떨어져 있어도 프로그램이 정상적으로 동작한다. 물론 예외적으로 offset이 너무 큰 경우에는 접근하려는 메모리 주소를 접근할 수 없기 때문에 이러한 경우는 주의를 해주어야 한다. 결론적으로 pc-relative addressing을 통해서 데이터의 위치와 관련없이 자유롭게 접근할 수 있기 때문에 프로그램이 location independent하게 동작한다고 이해할 수 있다.

## 5. 추가 실험

### 1) Unaligned data transfer 실험

실험에서 확인한 대로 일반적으로 코드를 작성하게 되면 align을 자동으로 맞춰준다. 하지만 간단

히 생각해보면 네트워크나 정말 작은 시스템 같은 환경에서는 데이터의 사이즈를 최대한 줄이는 것이 중요하다고 생각된다. 따라서 unaligned된 데이터를 이동시킬 때의 장점과 단점을 비교하기 위해서 추가 실험을 진행한다.

```
1      area lab3_3,code
2      entry
3      __main proc
4      export __main [weak]
5  start adr r0,data1
6      ldr r1,[r0,#0x00]
7      ldr r2,[r0,#0x01]
8      ldr r3,[r0,#0x02]
9      ldr r4,[r0,#0x03]
10     ldrh r5,[r0,#0x00]
11     ldrh r6,[r0,#0x01]
12     ldrh r7,[r0,#0x02]
13     ldrh r8,[r0,#0x03]
14     ;
15     ldr r0,=data2
16     str r1,[r0,#0x00]
17     str r2,[r0,#0x01]
18     str r3,[r0,#0x02]
19     str r4,[r0,#0x03]
20     strh r5,[r0,#0x00]
21     strh r6,[r0,#0x01]
22     strh r7,[r0,#0x02]
23     strh r8,[r0,#0x03]
24     b .
25     endp
26     align
27 data1 dcb 0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef
28     area lab3data,data
29 data2 space 16
30     end
```

그림 72 작성한 lab3\_3.s 코드

이를 확인하기 위해서 위 [그림 72]와 같이 lab3\_3.s 라는 코드를 작성하였다. 코드에 기능에 대해 간단히 설명하자면 먼저 실험에서 이동에 사용할 데이터를 data1에 바이트 단위로 저장한다. 그리고 data1의 주소를 r0로 가져온 뒤, ldr을 통해 4바이트 단위로 데이터를 읽어드리는데 offset을 0부터 3까지 바꿔가며 데이터를 로드한다. 그리고 ldrh로도 0부터 3까지 offset을 주며 2바이트씩 데이터를 로드한다. 그리고 space 16을 통해 메모리에 SRAM영역에 16바이트 할당 받고 그 주소를 r0에 다시 넘겨준다. 그리고 str 명령어를 통해 offset을 동일하게 0부터 3까지 바꿔가며 4바이트씩 메모리에 저장해주고 strh로도 동일하게 offset을 0부터 3까지해서 2바이트씩 메모리에 저장해주는 프로그램이다.

		5: start adr r0,data1		
0.009 us	0x080002C8 A00E	ADR	r0,{pc}+0x3C ; @0x08000304	
	6:	ldr r1,[r0,#0x00]		
0.019 us	0x080002CA 6801	LDR	r1,[r0,#0x00]	
	7:	ldr r2,[r0,#0x01]		
0.037 us	0x080002CC F8D02001	LDR	r2,[r0,#0x01]	
	8:	ldr r3,[r0,#0x02]		
0.028 us	0x080002D0 F8D03002	LDR	r3,[r0,#0x02]	
	9:	ldr r4,[r0,#0x03]		
0.037 us	0x080002D4 F8D04003	LDR	r4,[r0,#0x03]	
	10:	ldrh r5,[r0,#0x00]		
0.019 us	0x080002D8 8805	LDRH	r5,[r0,#0x00]	
	11:	ldrh r6,[r0,#0x01]		
0.028 us	0x080002DA F8B06001	LDRH	r6,[r0,#0x01]	
	12:	ldrh r7,[r0,#0x02]		
0.019 us	0x080002DE 8847	LDRH	r7,[r0,#0x02]	
	13:	ldrh r8,[r0,#0x03]		
	14:	;		
0.028 us	0x080002E0 F8B08003	LDRH	r8,[r0,#0x03]	
	15:	ldr r0,=data2		
0.019 us	0x080002E4 4809	LDR	r0,[pc,#36] ; @0x0800030C	
	16:	str r1,[r0,#0x00]		
0.019 us	0x080002E6 6001	STR	r1,[r0,#0x00]	
	17:	str r2,[r0,#0x01]		
0.037 us	0x080002E8 F8C02001	STR	r2,[r0,#0x01]	
	18:	str r3,[r0,#0x02]		
0.028 us	0x080002EC F8C03002	STR	r3,[r0,#0x02]	
	19:	str r4,[r0,#0x03]		
0.037 us	0x080002F0 F8C04003	STR	r4,[r0,#0x03]	
	20:	strh r5,[r0,#0x00]		
0.019 us	0x080002F4 8005	STRH	r5,[r0,#0x00]	
	21:	strh r6,[r0,#0x01]		
0.028 us	0x080002F6 F8A06001	STRH	r6,[r0,#0x01]	
	22:	strh r7,[r0,#0x02]		
0.019 us	0x080002FA 8047	STRH	r7,[r0,#0x02]	
	23:	strh r8,[r0,#0x03]		
0.028 us	0x080002FC F8A08003	STRH	r8,[r0,#0x03]	
	24:	b .		
0.028 us	0x08000300 E7FE	B	0x08000300	
	0x08000302 0000	DCW	0x0000	
	0x08000304 2301	DCW	0x2301	
	0x08000306 6745	DCW	0x6745	
	0x08000308 AB89	DCW	0xAB89	
	0x0800030A EFCD	DCW	0xEFCD	

그림 73 작성한 lab3\_3.s disassembly 코드와 수행시간

위에 [그림 73]을 통해서 각 명령어별로 수행되는데 걸린 시간을 확인할 수 있다. 가장 기본적인 ldr에서 offset이 없을 때는 2사이클인 0.019us로 가장 작은 시간이 소요되는 것을 알 수 있다. 그리고 offset이 1과 3인 경우에는 unaligned되어있기 때문에 데이터를 2번에 걸쳐 가져오고, 가져온 데이터를 정렬까지 해주어야 하기 때문에 총 4사이클이 소요되며 0.074us로 가장 긴 시간이 소요되었다. 다음으로 offset이 2인 경우도 unaligned되어 있는 경우이지만 데이터를 두 번에 걸

쳐 가져오기만 하면 되기 때문에 총 3사이클이 걸려 0.028us가 소요되었다. 비슷하게 ldrh의 경우도 offset이 0과 2인 경우는 align되어있기 때문에 0.019us로 2사이클이 소요되고 offset이 1과 3인 경우는 unaligned되어있기 때문에 2번에 걸쳐 데이터를 가져와 총 3사이클인 0.028us가 소요되었다. 이를 통해서 데이터를 가져올 때 aligned되어있지 않은 경우 align되어있는 경우보다 적게는 1사이클, 많게는 2사이클이나 더 소요되어 시간상으로 1.5~2배의 소요시간이 들었다.

마찬가지로 str과 strh 명령어를 사용하는 경우에도 ldr과 ldrh를 사용했을 때와 동일한 결과가 나왔다. str 명령어는 aligned되어있으면 2사이클인 0.019us, offset과 1과 3으로 unaligned되어 있으면 4사이클인 0.037us, offset이 2로 unaligned되어 있으면 3사이클인 0.028us가 소요되었다. 똑같이 strh명령어는 align되어있는 경우 2사이클 0.019us, unaligned된 경우는 3사이클인 0.028us가 소요되었다.

이번 추가실험을 통해서 align되어있는 경우와 unaligned된 경우의 데이터 이동에서의 시간 차이를 확인해보았다. 이를 통해 소요시간 측면에서는 무조건 align되어있는 것이 좋다는 것을 알 수 있었다. unaligned된 경우는 적게는 1.5배에서 2배까지 소요시간이 차이가 났다.

## 2) 추가적인 데이터 이동 명령어 실험

이번 추가 실험에서는 기존 실험에서 사용한 adr, mov, ldr, ldrb, ldrh, str, strb, strh를 제외한 다른 데이터 이동과 관련된 명령어들을 추가로 알아보고 동작을 알아보려고 한다. 그 중 여러 레지스터의 데이터를 가져오거나 저장할 수 있는 LDM, STM을 먼저 알아보고 다음으로는 exclusive하게 데이터를 로드하고 저장할 수 있는 LDREX와 STREX 명령어를 알아보려 한다.

명령어들을 확인하기 위한 코드는 아래 [그림 74]를 통해서 확인할 수 있다. 코드의 내용을 간략히 설명하자면 먼저 data1이 저장되었는 주소를 r0에 저장한다. 그리고 ldm 명령어를 통해 r0에 있는 4바이트 데이터를 각각 r1, r2에 저장한다. 그리고 데이터영역에 space 16을 통해 16바이트 공간을 할당 받고 그 주소를 r3에 저장한다. 그리고 stm 명령어를 통해서 r3위치에 r1, r2에 있는 값을 저장해주는 역할을 한다. 그리고 다음 부분에서는 ldrex를 통해 data1의 4바이트 데이터를 r4에 로드한다. 그리고 프로세서가 해당 메모리에 접근하지는 체크하도록 lock을 건다. 그리고 바로 다음 줄에서 strex를 통해 r4의 데이터를 r3의 주소에 넘기고 그 수행결과를 r5에 저장한다. 여기서 사용한 명령어는 아래에서 더 자세히 설명 할 예정이다.

```

1      area lab3_4,code,align=7
2      entry
3      main proc
4      export __main [weak]
5      adr r0,data1
6      ldm r0,{r1,r2}
7      ldr r3,=data2
8      stm r3,{r1,r2}
9      ;
10     ldrex r4,[r0]
11     strex r5,r4,[r3]
12     b .
13     endp
14     align
15     data1 dcd 0x01234567, 0x89abcdef
16     area lab3data,data
17     data2 space 16
18     end

```

그림 74 작성한 lab3\_4.s 코드

R0	0x08000318	<div>Memory 1</div> <div>Address: 0x20000000</div> <div>0x20000000: 67 45 23 01 EF CD AB 89</div> <div>0x20000008: 00 00 00 00 00 00 00 00</div>
R1	0x01234567	
R2	0x89ABCDEF	
R3	0x20000000	
R4	0x01234567	
R5	0x00000001	

그림 75 프로그램 수행 후 레지스터와 메모리 값

[그림 75]를 보면서 작성한 코드의 결과를 해석하며 각 명령어의 기능을 설명하고자 한다. 먼저 r0에는 예상대로 data1이 저장되어 있는 주소 0x08000318이 저장되었다. 그리고 ldm 명령어 하나로 r1과 r2에 0x01234567과 0x89abcdef가 모두 저장된 것을 확인할 수 있다. 그리고 r3에는 당연히 space 16으로 만들어준 공간의 주소 0x20000000이 저장되어 있고 해당 주소에 stm 하나의 명령어를 통해서 두 레지스터의 값을 모두 적어준 것을 확인할 수 있다.

이러한 ldm과 stm 명령어의 장점은 하나의 명령어로 여러 동작을 명령할 수 있어서 수행시간이 감소한다는 장점과 코드를 알아보기 쉽고 더 짧게 쓸 수 있다는 효율적인 장점이 있다. [그림 76]에서 볼 수 있듯이 ldm 명령어로 2개의 레지스터에 값을 로드할 때는 0.028us로 3사이클이 걸리는데 반해 원래대로 ldr 명령어를 2번 사용하면 0.019us + 0.019us로 총 0.038us로 4사이클이 걸리게 된다. 한 번에 지정하는 레지스터가 많을수록 더 많은 단축 효과를 가져온다. 마찬가지로 stm명령어도 수행시간이 줄어드는 것을 직접 확인할 수 있다.

		6:	ldm r0,{r1,r2}
0.028 us	0x08000302	E8900006	LDM r0,{r1-r2}
		7:	ldr r1,[r0]
0.019 us	0x08000306	6801	LDR r1,[r0,#0x00]
		8:	ldr r2,[r0]
0.019 us	0x08000308	6802	LDR r2,[r0,#0x00]
		9:	ldr r3,=data2
0.019 us	0x0800030A	4B07	LDR r3,[pc,#28] ; @0x08000328
		10:	stm r3,{r1,r2}
0.028 us	0x0800030C	E8830006	STM r3,{r1-r2}
		11:	str r1,[r3]
0.019 us	0x08000310	6019	STR r1,[r3,#0x00]
		12:	str r2,[r3]
0.019 us	0x08000312	601A	STR r2,[r3,#0x00]

그림 76 수행시간 비교를 위해 약간 수정한 코드

다음으로 [그림 74]의 line 10과 line 11 명령어는 ldrex 명령어와 strex명령어이다. 이 명령어는 주로 세트로 같이 사용되는 명령어로 본래의 기능은 ldr, str과 동일하다. 다만 데이터를 로드하고 작업을 한 뒤 저장하는 과정에서 프로세서가 해당 메모리에 접근하였는지 여부를 체크해 주는 기능을 포함하고 있는 명령어이다. 즉 로드와 스토어가 atomic 또는 mutual exclusive하게 동작할 수 있도록 해준다. 즉 해당 명령어 수행 도중 다른 명령어가 메모리에 접근하거나 인터럽트가 걸리는 등 방해가 받게 되면 그 사실을 프로그래머가 인지할 수 있도록 도와준다.

그 방법은 ldrex 명령어가 불리게 되면 해당 메모리에 락을 걸어 두고 프로세서가 해당 메모리에 접근하면 감지해서 알려준다. 위의 예시에서는 strex명령어 바로 다음에 오는 r5에 스토어의 성공 여부를 알려준다. 만약 실패하였다면 메모리에 프로세서가 접근했다는 뜻으로 r5에 1을 저장해주고 스토어는 하지 않는다. 만약 ldrex와 strlex사이에 메모리가 접근 받지 않았다면 r5에 0을 저장하고 str과 동일하게 스토어를 해준다.

하지만 이번 추가 실험에서는 디버깅모드를 사용하기 때문에 항상 atomic하게 동작하지 못했고 항상 r5에는 1이 저장되었다. 그 이유는 우리가 디버깅 모드를 사용할 때 명령어를 한줄한줄 실행시키기 위해서 명령어가 수행될 때 마다 프로세서가 인터럽트를 걸어서 멈춰주기 때문에 ldrex와 strex 사이에 어떠한 방해가 없더라도 디버깅 모드 자체에서 인터럽트를 걸기 때문에 항상 프로세서가 락을 건 메모리에 접근한다. 따라서 이번 추가 실험에서는 직접 확인해 볼 수는 없었지만 새로운 명령어를 배울 수 있었던 좋은 기회였다.

### 3) Pre-indexed address와 post-indexed 실험

3번째 추가 실험에서는 이론에서는 다뤘지만 실제 실험에서는 중요하게 다루지 않았던 pre-indexed address와 post-indexed address를 다루고자 한다. 이를 위해서 아래 [그림 77]처럼 코드를 작성하였다.

```
1      area lab3_5,code
2      entry
3      __main proc
4      export __main [weak]
5  start ldr r0,data1
6        ldr r1,=data3
7        str r0,[r1,#4]
8        str r0,[r1,#8]!
9        ldr r0,data2
10       str r0,[r1],#4
11       b .
12       endp
13       align
14  data1 dcd 0x12345678
15  data2 dcd 0xbabeface
16       area lab3data,data
17  data3 space 32
18       end
```

그림 77 직접 작성한 lab3\_5 코드

결과를 살펴보기 전에 코드를 간략하게 설명하면 space 32로 할당 받은 메모리 공간에 가장 기본적인 offset addressing으로 4바이트 떨어진 공간에 data1을 저장한다. 그리고 다음 줄에서는 pre-indexed addressing기법을 이용해 8바이트 떨어진 공간에 똑같이 data1을 저장하고 pre-indexed addressing임으로 r1에 들어있는 주소가 업데이트 된다. 그리고 post-indexed addressing기법을 이용해 data2를 r1이 가리키는 주소에 저장한 뒤 r1의 값을 업데이트 한다.

		0x20000000	
R0	0x12345678	0x20000000: 00 00 00 00 78 56 34 12	
R1	0x20000008	0x20000008: 78 56 34 12 00 00 00 00	

그림 78 Line 8까지 실행 후 레지스터와 메모리 값

[그림 78]에서 확인 가능한 것처럼 line 8까지 실행 후에는 r1이 0x20000008로 업데이트 되었고 정상적으로 0x20000000에서 각각 4바이트와 8바이트 떨어진 공간에 data1이 메모리에 저장된 것을 확인할 수 있다.



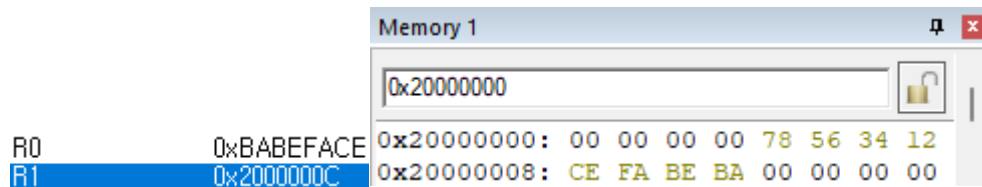


그림 79 Line 10까지 실행 후 레지스터와 메모리 값

그리고 [그림 79]에서 볼 수 있듯 post-indexed addressing으로 data2를 저장하였기 때문에 기존에 r0가 가리키던 0x20000008에 data2가 저장되고 그 후에 r1에 저장되어 있는 주소가 더해진 것을 확인할 수 있다.

이를 통해서 pre-indexed addressing 기법과 post-indexed addressing 기법의 기본적인 사용법과 동작을 알아보았다.

## 6. 결론

이번 3주차 실험을 통해서 우리가 사용하는 ARM 프로세서의 어셈블리 언어에서 데이터 이동과 관련한 여러가지 명령어를 이해하고 명령어를 사용하는 과정에서 중요한 개념들을 공부할 수 있었다. 예를 들어 2가지 endian모드의 차이를 이해하고 little endian 모드에서 데이터가 저장되고 로드되는 방식을 살펴보았다. 또한 pseudo-instruction이 사용되는 경우와 그 동작을 이해하고 그와 관련된 directive인 ldr의 동작을 이해했다. 마지막으로 중요하게 bit-alias 영역을 통해서 데이터 수정하는 것을 직접 해봄으로써 bit-banding의 방법과 장점을 익혔다. 그 외에도 몇 가지 directive와 어셈블리어를 해석하는 방법, µVision을 이용해서 작성한 프로그램을 디버깅 하는 방법이나 레지스터나 메모리 등 원하는 정보를 보는 방법을 익힐 수 있었다.

## 7. 참고문헌

서강대학교 전자공학과 (2023). 마이크로프로세서개론. p.27~78, Appendix A.

STMicroelectronics (2017). STM32F10xxx/20xxx/21xxx/L1xxx Cortex-M3 Programming Manual (Rev 6) p.22, p.27~29, p.60~66, p.79~80

STMicroelectronics (2021). STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced ARM-based 32-bit MCUs Reference Manual (Rev 21). p.53

ARM. (2004). ARM7TDMI Technical Reference Manual (Rev. r4p1)

ARM. (1998). ARM Software Development Toolkit Reference Guide (Version 2.50)

Yiu, J. (2010). The Definitive Guide To The ARM Cortex-M3 (2nd Edition)

히연. (2021). EMBEDDED RECIPES.