

마이크로프로세서응용실험 LAB06 결과보고서

Subroutines

20181536 엄석훈

1. 목적

- Subroutine의 호출 및 복귀와 관련된 명령어들과 그들의 동작에 대해 이해한다.
- Stack의 활용방법, 연관 명령어, subroutine과의 연관성 등을 이해한다.
- Subroutine으로의 변수(parameter) 전달 방식에 대해 이해한다.
- 개별적인 파일에 작성된 subroutine의 assemble/compile 및 link를 위한 directives의 역할을 이해한다.

2. 이론

1) Subroutines

크기가 큰 프로그램을 작성할 때 기능에 따라서 나누어 코드를 작성하는 것이 가독성과 재사용성, 유지보수의 측면에서 모두 유리하다. 이렇게 큰 프로그램을 작성할 때 subroutine을 사용하면 프로그램을 작은 단위로 쪼개어 작성할 수 있다. 또한 특히 어셈블리로 subroutine을 작성할 때는 언제든지 다른 subroutine으로 호출되거나 인터럽트가 걸릴 수 있으므로 항상 이전상태를 기억하고 다시 복구해주는 부분까지 잘 고려해야 한다.

일반적으로 BL명령어를 이용해서 이전의 PC를 LR에 저장하고 subroutine을 호출하는데 다시 원래 흐름으로 돌아가기 위해서는 LR의 값을 PC로 잘 복원해야 한다. 또한 subroutine안에서 다른 subroutine을 호출하는 경우도 빈번한데 이러한 경우 LR의 값이 갱신되어 이전의 PC를 잃어버릴 수 있다. 이러한 상황을 대비하여 subroutine이 호출될 때 stack 영역에 이전의 상태를 모두 보관하고 다시 복원하는 방식으로 코드를 작성하면 안전하게 subroutine을 사용할 수 있다.

2) Stacks

Stack은 컴퓨터공학에서 매우 자주 사용되는 자료구조 중 하나로 LIFO(last-in first-out)의 방식으로 마지막에 삽입(push)된 데이터가 가장 먼저 나오는(pop) 자료구조이다. 따라서 stack에서는 데이터를 넣거나 뺄 위치인 top을 유지하는 것이 중요한데 주로 R13(SP) 레지스터에서 이를 관리한다. 그리고 stack에는 메모리의 특정 부분을 end of stack과 bottom of stack으로 지정해서 stack 영역을 벗어나거나 침범되지 않도록 관리한다.

보통 stack을 처음 초기화 하면 sp를 bottom of stack을 가리키도록 초기화한 뒤 데이터를 추가하면 sp를 감소시키고 데이터를 top위치에 저장한다. 그리고 데이터를 뺄 때는 top에서 데이터를 가져온 뒤 sp를 증가시켜 top의 위치를 변경한다. 이때 sp는 항상 word단위로 관리한다. 물론 ARM 프로세서에서 선호에 따라 sp를 증가시키는 방향으로 하거나 sp의 이동과 데이터의 push, pop의 순서를 변경할 수 있다. 하지만 일반적으로는 위에 설명한 대로 sp는 감소하는 방향으로 진행하고 push를 할 때는 sp를 먼저 이동하고 pop을 할 때는 sp에서 데이터를 빼고 sp를 증가시킨다.

마지막으로 subroutine에서 stack을 사용하는 경우는 다음과 같다. 새로운 subroutine에서 레지스터를 사용해야 하는 경우 이전에 레지스터에 저장되어 있던 값을 유지하기 위해 사용할 레지스터의 내용을 stack에 push하고 subroutine의 동작을 수행한 뒤 다시 해당 내용들을 pop해서 이전의 상태를 복구한 뒤 이전의 흐름으로 돌아간다. 이때 lr의 내용을 push해서 stack에 저장하고 pop은 pc에 해서 branch하지 않고 바로 PC의 값을 이전으로 돌리기도 한다. 이처럼 stack을 활용하면 subroutine을 안전하게 사용할 수 있다.

3) Parameter passing

Subroutine을 사용할 때 원하는 값을 가지고 원하는 결과를 만들기 위해서는 parameter를 전달해 주어야 하는 경우가 많다. 간단히 예를 들어 두 숫자에 대해 복잡한 연산을 해주는 subroutine이 있다고 할 때 parameter에 해당하는 두 숫자를 subroutine에게 알려주어야 한다. 이러한 parameter를 전달하는 방식을 몇 가지 살펴보면 register를 이용한 방식, stack을 이용한 방식, reference를 전달하는 방식 3가지가 있다.

먼저 register를 이용하는 방식은 전달해야 할 parameter가 얼마 없을 때 주로 사용하는 방식으로

subroutine을 호출하기 전에 특정 레지스터에 parameter로 전달하고자 하는 값을 저장한 뒤 subroutine을 호출하여 해당 subroutine에서 전달받은 레지스터 값을 그대로 사용할 수 있게 하는 방식이다. 이 방식을 사용할 때는 subroutine이 호출되고 이전의 레지스터의 상태를 stack에 저장할 때 parameter를 저장하고 있는 레지스터는 제외하고 stack에 저장해주어야 하며, 속도가 가장 빠르고 간단하다는 장점이 있다.

다음으로 stack을 이용한 parameter전달 방식이 있다. 일반적으로 고급 언어에서 많은 parameter를 전달할 때 일부 parameter는 레지스터를 통해 전달하고 나머지 parameter는 stack을 통해서 전달하는 방식을 취하고 있다. 먼저 subroutine을 호출하기 전에 전달하려는 parameter를 stack에 저장하고 subroutine을 호출한다. 그리고 일반적으로 subroutine에 들어가면 처음에 이전의 레지스터의 상태를 stack에 저장해준다. 그리고 subroutine에서 stack에 push한 만큼 계산해서 offset으로 사용해 sp와 offset을 이용해서 subroutine호출 전에 stack에 저장하였던 내용을 접근한다. Stack 자료구조가 LIFO구조를 가지고 있지만 반드시 top에서 가리키고 있는 데이터를 접근하지 않고 offset을 이용하여 중간의 데이터를 접근할 수도 있기 때문에 사용 가능한 방식이다.

마지막으로 reference를 이용해서 parameter를 전달하는 방식이 있는데 다르게 말하면 reference하려는 주소를 parameter로 생각해서 전달하는 방식이다. 즉 subroutine은 전달받은 주소를 이용해서 데이터가 저장되어 있는 장소로 접근하여 데이터를 참조하는 방식이다. C언어에서 포인터의 개념과 유사한 방식이라고 생각할 수 있다. 따라서 subroutine에서 reference할 주소를 위의 register를 이용한 방식이나 stack을 이용한 방식으로 subroutine에게 전달하여 해당 subroutine에서 주소를 이용하여 데이터를 접근하는 방식인 것이다.

4) Modular programming directives

큰 프로그램을 작성할 때 subroutine으로 나누어 프로그래밍을 하는데 이 때 여러 사람이 함께 작업을 하기 위해서 또는 subroutine도 기준을 가지고 분류하기 위해서 여러 파일에 나누어 작성하는 경우가 많다. 이때 각 파일이 정상적으로 compile되어 linking되기 위해서는 다른 파일에 해당 subroutine또는 데이터가 있다고 알려주어야 한다. 이를 위해서 사용될 프로그램에서는 GLOBAL/EXPORT directive를 사용하고 사용할 프로그램에서는 IMPORT/EXTERN directive를 이용해서 assembler에게 다른 파일을 참조하라고 알려준다.

3. 실험 과정

1) 실험 1

STEP 1: Program 6.1을 이용해 프로젝트를 생성한다.

```
1  STACK_BASE EQU 0x20000100
2      area lab6_1,code
3      entry
4  __main proc
5      export __main [weak]
6  start ldr sp,=STACK_BASE
7      ldr r1,=0x1234
8      ldr r2,=0x4321
9      bl swap
10 ; doing something here using the result
11     ldr r1,=-0x4321
12     ldr r2,=-0x1234
13     bl swap
14 ; doing something here using the result
15 stop b stop
16     endp
17 swap proc
18     push {r3,lr}
19     cmp r1,r2
20     bgt r2m
21     mov r3,r2
22     mov r2,r1
23     mov r1,r3
24 r2m pop {r3,pc}
25     endp
26     end
```

그림 1 직접 작성한 lab6_1.s 코드

STEP 2: Line 6 명령어까지 수행한 후 sp의 내용을 확인한다.

Line 9 명령어까지 수행한 후 pc, lr의 내용을 확인한다.

Register	Value
Core	
R0	0x080002C9
R1	0xE000ED08
R2	0x40021000
R3	0xFE6FFFFF
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000100
R14 (LR)	0x0800013B
R15 (PC)	0x080002CC

그림 2 Line 6 수행 후 레지스터 값

Line 6 명령어를 수행하면 단순히 sp(r13)에 line 1에서 선언한 STACK_BASE값인 0x20000100이 저장된 것을 위 [그림 2]처럼 확인할 수 있다.

```

9:      bl swap
10: ; doing something here using the result
0x080002D4 F000F805 BL.W    0x080002E2 swap
11:      ldr r1,=-0x4321
0x080002D8 4906      LDR     r1,[pc,#24] ; @0x080002F4
12:      ldr r2,=-0x1234
0x080002DA 4A07      LDR     r2,[pc,#28] ; @0x080002F8
13:      bl swap
14: ; doing something here using the result
0x080002DC F000F801 BL.W    0x080002E2 swap
15: stop  b stop
16:      endp
17: swap proc
0x080002E0 E7FE      B       0x080002E0
18:      push {r3,lr}
→0x080002E2 B508      PUSH    {r3,lr}

```

그림 3 명령어의 주소값

Register	Value
Core	
R0	0x080002C9
R1	0x00001234
R2	0x00004321
R3	0xFE6FFFFF
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000100
R14 (LR)	0x080002D9
R15 (PC)	0x080002E2

그림 4 Line 9 수행 후 레지스터 값

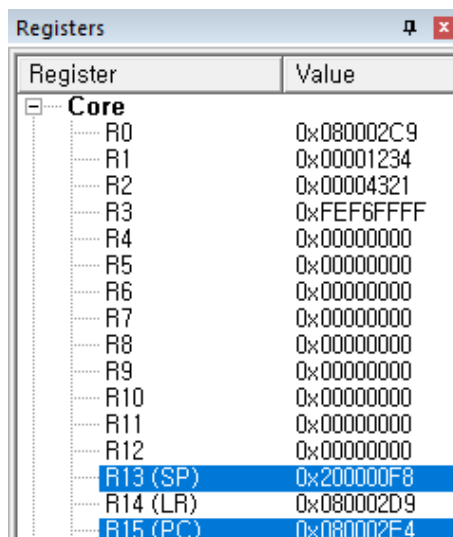
그리고 Line 9 bl swap 명령어 수행 후 [그림 3]에서 확인 가능한 line 17 swap의 주소인 0x080002E2로 PC값이 바뀌고 LR에는 swap에서 돌아오고 나서 수행해야 할 PC값인 line 11에 LSB가 1인 상태인 0x080002D9가 저장된 것을 위 [그림 4]에서 확인할 수 있다.

STEP 3: Lines 17-24에 소개된 subroutine swap에서 수행하는 일을 파악한다.

Line 18 명령어를 수행한 후 sp, 그리고 sp의 내용을 주소로 하는 메모리영역(stack)의 내용을 확인한다. 이 명령어의 대상으로 r3과 lr이 선택된 이유는 무엇인가?

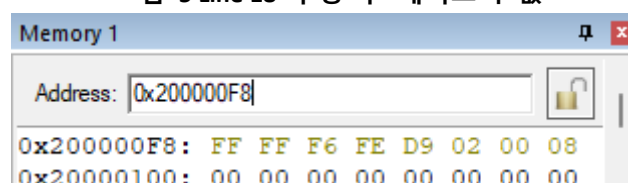
Line 24 명령어를 수행하기 전후의 r3, pc, sp의 내용을 파악한다. 이 명령어 수행에 따른 pc 내용의 변경은 어떤 의미로 해석할 수 있는가?

[그림 1]의 lines 17-24의 내용을 분석해보면 먼저 r3과 lr에 저장되어 있던 값을 stack에 저장한다. 다음으로 r1과 r2의 값을 비교해서 r1이 더 크다면 r2m으로 branch해서 스택에 저장되어 있던 r3와 lr의 값을 r3와 pc로 pop해서 r3의 값은 복구하고 기존 lr의 위치로 PC를 옮겨서 원래의 흐름으로 돌아간다. 반면 r1이 r2보다 작거나 같다면 r3를 임시저장소로 사용해서 r1과 r2의 값을 바꾼 뒤 pop을 통해 원래의 흐름으로 돌아간다. 즉 이 subroutine은 r1과 r2의 값을 비교해 더 크거나 같은 값을 r1에 저장하고 다른 하나는 r2에 저장하는 동작을 수행한다.



Register	Value
Core	
R0	0x080002C9
R1	0x00001234
R2	0x00004321
R3	0xFE6FFFFF
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x200000F8
R14 (LR)	0x080002D9
R15 (PC)	0x080002E4

그림 5 Line 18 수행 후 레지스터 값

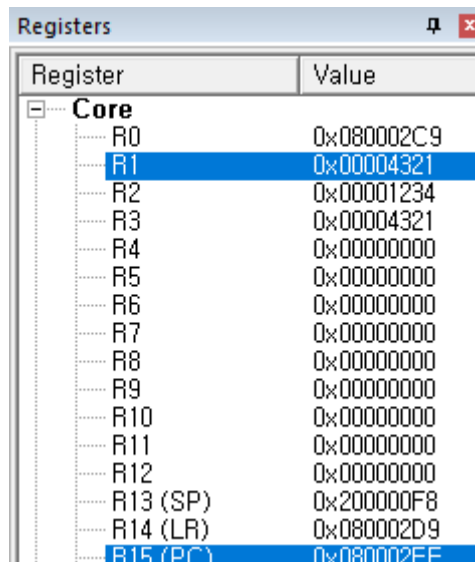


Memory 1	
Address	Value
0x200000F8	FF FF F6 FE D9 02 00 08
0x20000100	00 00 00 00 00 00 00 00

그림 6 Line 18 수행 후 메모리 값

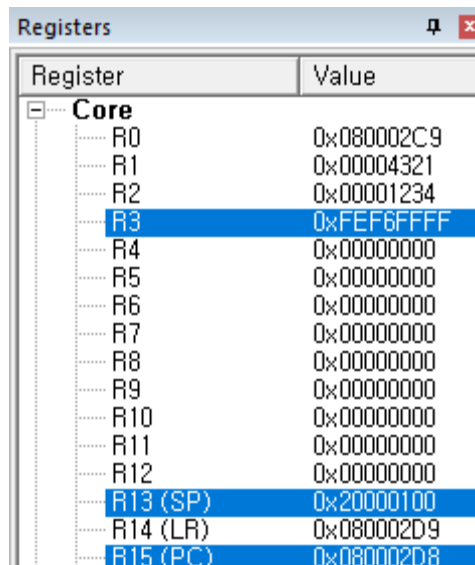
Line 18 명령어 수행 결과 sp의 값은 [그림 5]에서와 같이 0x200000F8이 저장되었는데 이는 stack에 8byte만큼 저장하였기 때문에 0x20000100에서 8을 뺀 값이 저장된 것이다. 그리고 [그림 6]을 보면 sp가 가리키는 곳에 r3와 lr의 내용이 저장된 것을 확인할 수 있다. Stack은 주소가 감소하는 방향으로 데이터를 저장하기 때문에 주소가 큰 쪽이 먼저 저장된 r3가 저장되었고 LSB가 주소가 낮은 쪽으로 저장되는 little endian방식으로 저장된 것 또한 알 수 있다.

Stack에 r3와 lr의 값을 저장한 이유는 r3는 해당 subroutine에서 사용할 예정이기 때문에 이전의 값을 잃어버리지 않기 위해 stack에 저장한 것이고 lr은 다른 subroutine으로 들어가 lr값을 잃어버리는 경우를 예방하고 bl명령어를 사용할 때 받은 돌아갈 주소를 stack에 저장하였다가 pop을 할 때 바로 pc에 저장하여 원래의 흐름으로 돌아가기 위해서 저장하였다.



Register	Value
Core	
R0	0x080002C9
R1	0x00004321
R2	0x00001234
R3	0x00004321
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x200000F8
R14 (LR)	0x080002D9
R15 (PC)	0x080002EE

그림 7 Line 24 수행 전 레지스터 값

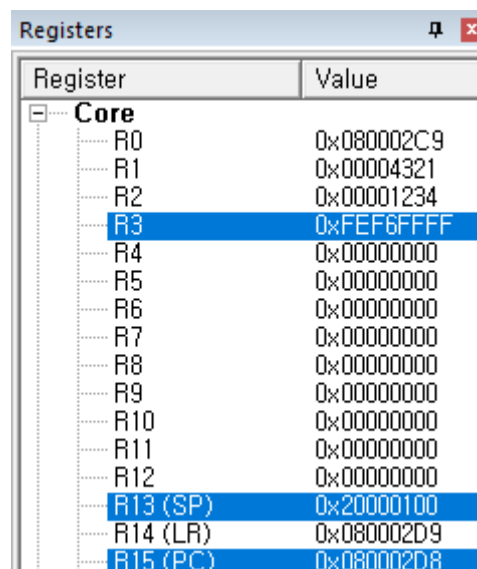


Register	Value
Core	
R0	0x080002C9
R1	0x00004321
R2	0x00001234
R3	0xFEFFFFFF
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000100
R14 (LR)	0x080002D9
R15 (PC)	0x080002D8

그림 8 Line 24 수행 후 레지스터 값

마지막으로 Line 24 명령어 수행 전 레지스터의 값은 [그림 7]을 통해 확인할 수 있다. r3에는 0x00004321로 subroutine에서 동작을 수행할 때 사용하였던 값이 들어있고 PC는 subroutine의 흐름대로 진행하였을 경우의 다음 명령어의 위치인 0x080002EE를 저장하고 있고 sp는 stack의 top인 0x200000F8을 그대로 잘 저장하고 있다. 그리고 line 24 pop {r3, pc}를 수행한 후 레지스터 값은 [그림 8]을 통해 확인할 수 있다. R3는 stack에 저장하였던 이전의 r3값인 0xFE6FFFF가 다시 복구되었고 sp는 pop을 하였음으로 다시 8이 증가해서 0x20000100을 저장하고 있고 PC는 기존 stack에 저장하였던 lr값이 복구되어 0x080002D8이 된 것을 확인할 수 있다. 이 명령어의 수행에 따라 bl 명령어를 수행할 때 lr에 저장하였던 원래 프로그램의 흐름으로 돌아갈 pc값을 stack에 저장하였다가 pc값에 pop해줌으로써 branch명령어 없이 다시 원래 프로그램의 흐름으로 돌아갈 수 있도록 하였다고 해석할 수 있다.

STEP 4: STEP 2 – STEP 3 과정을 lines 11 – 13의 명령어들에 대해서도 반복한다.



Register	Value
Core	
R0	0x080002C9
R1	0x00004321
R2	0x00001234
R3	0xFE6FFFF
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000100
R14 (LR)	0x080002D9
R15 (PC)	0x080002D8

그림 9 Line 11 수행 전 레지스터 값

Register	Value
Core	
R0	0x080002C9
R1	0xFFFFBCDF
R2	0xFFFFEDCC
R3	0xFE6FFFFF
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000100
R14 (LR)	0x080002E1
R15 (PC)	0x080002E2

그림 10 Line 13 수행 후 레지스터 값

[그림 9]를 통해 sp에는 원래 stack의 bottom인 0x20000100을 가리키고 있는 것을 확인할 수 있다. [그림 10]을 통해서도 bl swap이후 pc는 swap명령어의 주소인 0x080002E2가 되고 다시 돌아올 주소인 bl swap 명령어 다음 주소인 0x080002E0에 LSB가 1인 0x080002E1이 lr에 저장된 것을 확인할 수 있다.

Lines 17-24의 subroutine이 수행하는 동작은 위에서도 설명하였듯 r1과 r2를 비교해 크거나 같은 수를 r1에 다른 하나를 r2에 저장하는 동작을 수행한다.

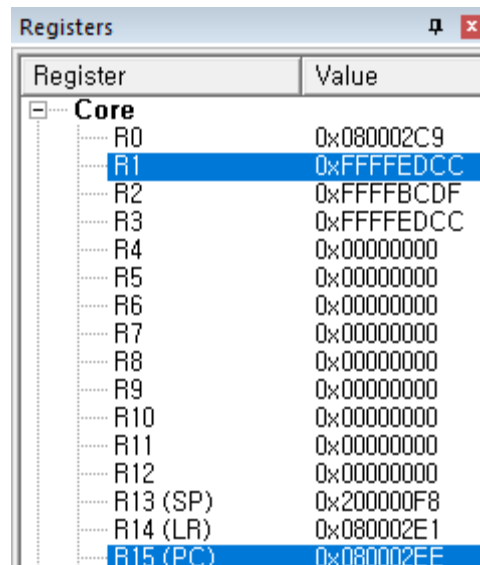
Register	Value
Core	
R0	0x080002C9
R1	0xFFFFBCDF
R2	0xFFFFEDCC
R3	0xFE6FFFFF
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x200000F8
R14 (LR)	0x080002E1
R15 (PC)	0x080002E4

그림 11 Line 18 수행 후 레지스터 값

Address	Value
0x200000F8	FF FF F6 FE E1 02 00 08
0x20000100	00 00 00 00 00 00 00 00

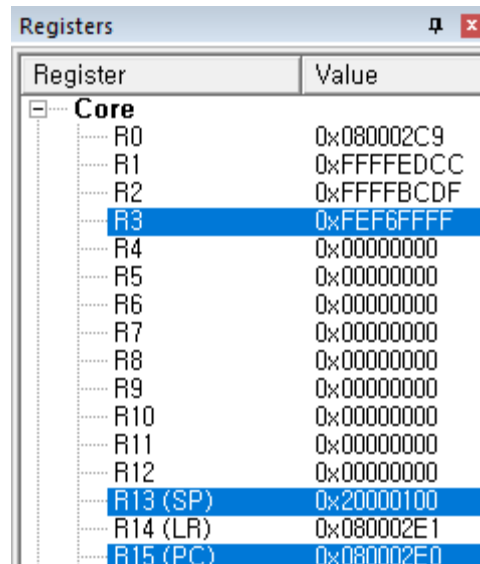
그림 12 Line 18 수행 후 메모리 값

Line 18의 push 명령어 수행 이후 [그림 11]에서 확인 가능하듯 sp는 0x20000100에서 8을 뺀 0x200000F8이 저장되었고 메모리의 stack 영역에는 기존 r3와 lr에 저장되어 있던 값이 저장된 것을 [그림 12]에서 확인 가능하다. 위에서도 설명하였듯 r3는 이 subroutine에서 사용하는 레지스터임으로 기존 r3의 값을 나중에 복구하기 위해 stack에 저장하였고 lr은 나중에 pc로 pop을 해서 원래의 흐름으로 돌아가기 위해 stack에 저장하였다.



Register	Value
Core	
R0	0x080002C9
R1	0xFFFFEDCC
R2	0xFFFFBCDF
R3	0xFFFFEDCC
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x200000F8
R14 (LR)	0x080002E1
R15 (PC)	0x080002EE

그림 13 Line 24 수행 전 레지스터 값



Register	Value
Core	
R0	0x080002C9
R1	0xFFFFEDCC
R2	0xFFFFBCDF
R3	0xFEFFFFFF
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000100
R14 (LR)	0x080002E1
R15 (PC)	0x080002E0

그림 14 Line 24 수행 후 레지스터 값

그리고 line 24 수행 전 [그림 13]에서 볼 수 있듯 r3는 subroutine에서 사용하던 값, sp는 stack의 top인 0x200000F8을 PC는 원래 흐름대로의 다음 명령어 주소인 0x080002EE를 저장하고 있다. 그러나 line 24 수행 이후 [그림 14]에서 볼 수 있듯 r3는 기존의 값으로 복구되고 PC는 bl swap 명령어 다음 명령어를 가리키고 sp는 전부다 pop 하였으므로 stack의 bottom인 0x20000100을

가리키는 것을 확인할 수 있다. Pop 명령어를 통해 기존 bl하면서 lr에 저장하였던 값을 pc에 그대로 넣어주어 branch없이 프로그램이 원래의 흐름으로 돌아갈 수 있었다.

STEP 5: 어떤 레지스터(들)을 통해 subroutine에 변수들이 어떻게 전달되고 있다고 파악했는가?

이러한 변수전달 방식이 갖는 제약사항은 어떤 것들이 있겠는가?

이 프로그램에서 subroutine swap에는 레지스터 r1과 r2에 parameter를 저장하여 전달하고 있다. Subroutine호출 전 전달할 parameter를 r1과 r2에 저장하고 subroutine을 호출하여 r1과 r2의 값을 사용할 수 있도록 해주었다. 이러한 변수 전달 방식은 빠르고 간단하다는 장점이 있지만 만약 전달해야 할 parameter가 많게 되면 레지스터에는 한계가 있기 때문에 전부 다 전달할 수가 없다는 제약사항이 있다.

2) 실험 2

STEP 6: Program 6.2를 이용해 프로젝트를 생성한다.

```
1  STACK_BASE EQU 0x20000100
2      area lab6_2,code
3      entry
4  __main proc
5      export __main [weak]
6  start ldr sp,=STACK_BASE
7      ldr r1,=0x1234
8      ldr r2,=0x4321
9      stmfd sp!,{r1,r2}
10     bl swap
11     ldmfd sp!,{r1,r2}
12     ; doing something here using the result
13     ldr r1,=-0x1234
14     ldr r2,=-0x4321
15     stmfd sp!,{r1,r2}
16     bl swap
17     ldmfd sp!,{r1,r2}
18     ; doing something here using the result
19 stop  b stop
20     endp
21 swap proc
22     stmfd sp!,{r4-r6,lr}
23     ldr r4,[sp,#0x10]
24     ldr r5,[sp,#0x14]
25     cmp r4,r5
26     bgt r2m
27     mov r6,r5
28     mov r5,r4
29     mov r4,r6
30     str r4,[sp,#0x10]
31     str r5,[sp,#0x14]
32 r2m  ldmfd sp!,{r4-r6,pc}
33     endp
34     end
```

그림 15 직접 작성한 lab6_2.s 코드

STEP 7: Line 9 명령어의 수행 전후에 sp, 그리고 stack 영역에 저장되는 내용의 변화를 확인한다.

Register	Value
Core	
R0	0x080002C9
R1	0x00001234
R2	0x00004321
R3	0xFEFFFFFF
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000100
R14 (LR)	0x0800013B
R15 (PC)	0x080002D4

그림 16 Line 9 수행 전 레지스터 값

Register	Value
Core	
R0	0x080002C9
R1	0x00001234
R2	0x00004321
R3	0xFEFFFFFF
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x200000F8
R14 (LR)	0x0800013B
R15 (PC)	0x080002D6

그림 17 Line 9 수행 후 레지스터 값

Line 9 수행 전에는 [그림 16]에서 보이듯 sp에 stack의 bottom 초기값인 0x20000000이 들어있었다. 그리고 Line 9로 r1, r2의 값을 stack에 저장한 뒤 8만큼 감소해서 sp에는 0x200000F8이 저장된 것을 [그림 17]에서 확인할 수 있다.

Memory 1	
Address:	0x200000F8
0x200000F8:	34 12 00 00 21 43 00 00
0x20000100:	00 00 00 00 00 00 00 00

그림 18 Line 9 수행 후 메모리 값

또한 [그림 18]에서 확인할 수 있듯이 r1, r2에 들어있던 값이 word 단위로 전체가 메모리의 stack 영역에 잘 저장된 것을 확인할 수 있다.

STEP 8: Line 22 명령어의 수행 전후에 sp, 그리고 stack 영역에 저장되는 내용의 변화를 확인한다. 이 명령어를 통해 stack에 r4-r6, lr을 저장하는 이유는 무엇인가? 또한, 이 명령어를 push 명령어로 문법을 고려하여 바꾸어 보자.

Register	Value
Core	
R0	0x080002C9
R1	0x00001234
R2	0x00004321
R3	0xFEFFFFFF
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x200000F8
R14 (LR)	0x080002DB
R15 (PC)	0x080002F0

그림 19 Line 22 수행 전 레지스터 값

Register	Value
Core	
R0	0x080002C9
R1	0x00001234
R2	0x00004321
R3	0xFEFFFFFF
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x200000E8
R14 (LR)	0x080002DB
R15 (PC)	0x080002F2

그림 20 Line 22 수행 후 레지스터 값

이전과 비슷하게 [그림 19]에서 line 22를 수행하기 전의 sp는 현재 stack의 top인 0x200000F8을 가리키고 있는 것을 확인할 수 있다. 그리고 line 22 명령어를 통해 stack에 4개의 값을 저장하였으므로 0x200000F8에서 0x10만큼 감소한 0x200000E8이 된 것을 [그림 20]을 통해서 확인할 수 있다.

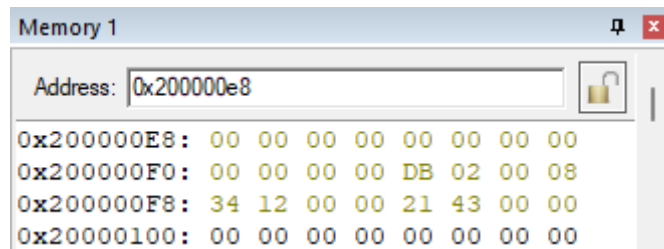


그림 21 Line 22 수행 후 메모리 값

또한 [그림 21]에서 볼 수 있듯이 기존 [그림 19]에서 r4, r5, r6, lr에 해당하는 값이 차례대로 stack에 저장된 것을 확인할 수 있다.

이때 line 22 명령어를 통해서 해당 레지스터의 값을 stack에 저장한 것은 r4, r5, r6 레지스터를 이번 subroutine에서 사용하는데 해당 값을 임시로 stack에 저장하였다가 subroutine이 끝나고 복원하기 위해서 stack에 저장하였다. 또한 lr값은 다른 subroutine으로 들어가 lr값을 잃어버리지 않도록 예방하고 나중에 subroutine이 끝나고 pc의 값으로 넣어주어 별 다른 branch명령어 없이 프로그램의 흐름을 원래의 흐름으로 돌아가기 위해서 stack에 저장하였다.

그리고 stmfd대신 동일한 동작을 push 명령어를 사용해서 아래 [그림 22]와 같이 작성할 수 있다. 이렇게 push 명령어로 바꾸면 가독성도 더 좋아지고 코드 작성 시 실수할 가능성이 줄어들 수 있다.

```

21 swap proc
22     push {r4-r6,lr}
23     ldr r4,[sp,#0x10]
24     ldr r5,[sp,#0x14]
25     cmp r4,r5
26     bgt r2m
27     mov r6,r5
28     mov r5,r4
29     mov r4,r6
30     str r4,[sp,#0x10]
31     str r5,[sp,#0x14]
32 r2m ldmfd sp!,{r4-r6,pc}
33     endp

```

그림 22 push 명령어로 수정한 line 22 코드

STEP 9: Line 23-24 명령어들을 사용하는 목적은 무엇인가? 이 명령어들을 통해 변수들이 subroutine으로 전달되고 있다고 볼 수 있는가?

Line 23-24의 명령어는 sp와 offset을 이용해서 subroutine 호출 전에 stack에 저장하였던 parameter를 r4와 r5로 로드하는 역할을 수행한다. 이 명령어들을 통해서 r4로 0x1234, r5로

0x4321을 정상적으로 전달한 것을 아래 [그림 23]을 통해서 확인할 수 있다. 그리고 당연하게도 sp의 값에는 전혀 변동이 없이 안전하게 원하는 값만 가져온 것 또한 확인할 수 있다.

Register	Value
Core	
R0	0x080002C9
R1	0x00001234
R2	0x00004321
R3	0xFEFFFFFF
R4	0x00001234
R5	0x00004321
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x200000E8
R14 (LR)	0x080002DB
R15 (PC)	0x080002F0

그림 23 Line 24 수행 후 레지스터 값

STEP 10: Line 32 명령어의 수행 전후에 sp, 그리고 r4-r6, lr들의 내용의 변화를 확인한다. 또한, 이 명령어를 문법을 고려하여 pop 명령어로 바꾸어 보자.

Register	Value
Core	
R0	0x080002C9
R1	0x00001234
R2	0x00004321
R3	0xFEFFFFFF
R4	0x00004321
R5	0x00001234
R6	0x00004321
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x200000E8
R14 (LR)	0x080002DB
R15 (PC)	0x080002FE

그림 24 Line 32 수행 전 레지스터 값

Register	Value
Core	
R0	0x080002C9
R1	0x00001234
R2	0x00004321
R3	0xFE6FFFFF
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x200000F8
R14 (LR)	0x080002DB
R15 (PC)	0x080002DA

그림 25 Line 25 수행 후 레지스터 값

Line 32 수행 전의 [그림 24]에서 보이듯 sp는 0x200000E8이다. 그리고 다른 r4, r5, r6은 모두 subroutine에서 사용된 값으로 저장되어 있고 lr은 0x0800002DB로 subroutine에서 바뀐 적 없이 그대로 들어있다. 그리고 line 32 수행 후 [그림 25]에서 확인 가능하듯 먼저 sp는 총 4개의 데이터를 pop했기 때문에 0x10만큼 증가한 0x200000F8이 저장되어 있고 r4, r5, r6는 모두 subroutine이 호출되고 subroutine이 끝나고 복원하려 했던 값이 그대로 다시 저장된 것을 확인할 수 있다. 또한 lr의 값은 line 32 명령어로 로드하지 않았으므로 그대로이고 대신 stack에 저장되어 있던 기존 lr값을 pc로 로드하였기 때문에 pc값에 0x0800002DB에서 LSB를 무시한 0x0800002DA 가 로드되어 원래의 프로그램 흐름으로 돌아간 것을 확인할 수 있다.

또한 line 32의 원래 ldmfd 명령어 대신 더 편리하고 가독성 좋은 pop 명령어로 동일한 동작을 수행하도록 변경한 명령어는 아래 [그림 26]을 통해서 확인 가능하다.

```

21 swap proc
22     push {r4-r6,lr}
23     ldr r4,[sp,#0x10]
24     ldr r5,[sp,#0x14]
25     cmp r4,r5
26     bgt r2m
27     mov r6,r5
28     mov r5,r4
29     mov r4,r6
30     str r4,[sp,#0x10]
31     str r5,[sp,#0x14]
32 r2m pop{r4-r6,pc}
33     endp

```

그림 26 pop 명령어로 수정한 line 32 명령어

STEP 11: Line 11 명령어를 수행한 후 sp, 그리고 r1, r2의 내용을 확인한다.

Register	Value
Core	
R0	0x080002C9
R1	0x00004321
R2	0x00001234
R3	0xFEFFFFFF
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000100
R14 (LR)	0x080002DB
R15 (PC)	0x080002DC

그림 27 Line 11 수행 후 레지스터 값

Line 11 명령어를 통해서 stack에 저장되어 있던 데이터를 r1과 r2로 로드하였다. 그 결과 기존 sp 인 0x200000F8에서 8이 증가된 0x20000100가 sp에 저장되었다. 그리고 r1과 r2에는 subroutine에 의해 순서가 변경된 값이 로드되었다. 원래는 r1에 0x1234, r2에 0x4321이었는데 swap subroutine에 의해 stack에 저장되어 있던 순서가 바뀌었고 그대로 stack에서 로드하였으므로 r1에는 0x4321, r2에는 0x1234가 저장된 것을 [그림 27]을 통해서 확인할 수 있다.

3) 실험 3

STEP 12: Program 6.3를 이용해 프로젝트를 생성한다.

1	STACK_BASE EQU 0x20000100	28	fill_array proc
2	ARRAY_BASE EQU 0x20000100	29	push {r0-r3,lr}
3	ARRAY_SIZE EQU 0x8	30	ldr r0,[sp,#0x14] ;ARRAY_BASE
4	area lab6_3, code	31	ldr r1,[sp,#0x18] ;ARRAY_SIZE
5	entry	32	ldr r2,[sp,#0x1c] ;FILL_PATTERN
6	__main proc	33	mov r3,#0
7	export __main [weak]	34	lp1 str r2,[r0,r3, LSL #2]
8	start ldr sp,=STACK_BASE	35	add r3,#1
9	ldr r2,=ARRAY_BASE	36	cmp r3,r1
10	ldr r3,=ARRAY_SIZE	37	blt lp1
11	ldr r4,=0x5555aaaa	38	pop {r0-r3,pc}
12	push {r2-r4}	39	endp
13	bl fill_array	40	
14	add sp,#12	41	; fill_pointer(int *array, int size, int pattern)
15		42	{ int *p;
16	ldr r4,=0x77777733	43	for (p = &array[0]; p < &array[size]; p = p + 1)
17	push {r2-r4}	44	*p = pattern;
18	bl fill_pointer	45	}
19	add sp,#12	46	fill_pointer proc
20	stop b stop	47	push {r0-r3,lr}
21	endp	48	ldr r0,[sp,#0x14] ;ARRAY_BASE
22		49	ldr r1,[sp,#0x18] ;ARRAY_SIZE
23	; fill_array(int array[], int size, int pattern)	50	ldr r2,[sp,#0x1c] ;FILL_PATTERN
24	{ int i;	51	add r3,r0,r1, LSL #2
25	for (i = 0; i < size; i += 1)	52	lp2 str r2,[r0],#4
26	array[i] = pattern;	53	cmp r0,r3
27	}	54	blt lp2
		55	pop {r0-r3,pc}
		56	endp
		57	end

그림 28 직접 작성한 lab6_3.s 코드

STEP 13: A-27페이지를 통해 이 프로그램에서 사용되는 push 명령어와 pop명령어의 동작을 파악하고 이 명령어의 수행 결과를 그림 6.7과 비교해본다.

먼저 push 명령어는 push{cond} register의 꼴로 사용되며 register는 괄호 안에 레지스터 여러 개 여 포함시킬 수 있으며 push를 수행하면 register의 데이터를 sp가 가리키고 있는 stack의 top에서부터 주소를 감소시키는 방향으로 레지스터의 데이터를 word 단위로 저장하고 그만큼 sp를 빼서 업데이트 한다. 비슷하게 pop 명령어는 pop{cond} register의 꼴로 사용된다. Pop 명령어를 수행하면 sp가 가리키고 있는 stack의 top에서부터 주소를 증가시키는 방향으로 데이터를 word 단위로 읽어 들여 register에 저장한다. 그리고 데이터를 읽어 들인만큼 sp를 증가시켜서 업데이트 한다. 이때 만약 pc로 push를 하면 pop 명령어를 수행하면 새로운 pc로 자동으로 branch하게 된다. 교재의 그림 6.7과 비교해보면 맨 처음 sp는 0x20000100을 가리키고 있다가 push {r2-r4}를 통해서 총 12byte의 데이터를 stack에 저장하고 sp는 12감소한 0x200000F4가 되었다가 다시 한번 push {r0-r3,lr}을 통해서 총 20byte의 데이터를 stack에 저장하여 sp는 다시 20감소한 0x200000E0가 된 것을 확인할 수 있다. 그리고 만약 이 상태에서 pop을 하게 되면 sp인 0x200000E0의 값부터 pop한 데이터의 양만큼 sp가 증가할 것이다.

아래 [그림 29]를 통해서 push 명령어 2번을 사용하여 교재 그림 6.7과 동일한 상태를 확인할 수 있다. 또한 [그림 30]을 통해서 subroutine 마지막에 pop을 하고 난 뒤의 레지스터 값을 확인할 수 있다. 원래 subroutine 수행 전의 상태로 돌아온 것과 sp가 0x200000F4로 증가한 것 또한 확인할 수 있다.

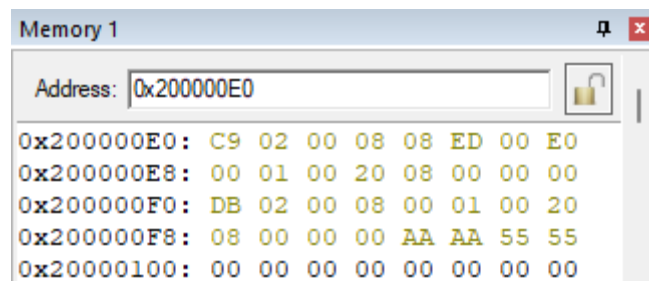


그림 29 push 명령어 수행 후 그림 6.7과 동일한 상태

Register	Value
Core	
R0	0x080002C9
R1	0xE000ED08
R2	0x20000100
R3	0x00000008
R4	0x5555AAAA
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x200000F4
R14 (LR)	0x080002DB
R15 (PC)	0x080002DA

그림 30 pop 명령어 수행 후 레지스터 값

STEP 14: Line 29 명령어의 대상 register들은 어떻게 선정되는지 생각해보자.

Line 29 명령어는 fill_array라는 subroutine안의 명령어로 push {r0-r3,lr}을 수행하고 있다. 레지스터 r0, r1, r2, r3은 모두 이 subroutine내부에서 사용해야 하는데 나중에 subroutine이 끝나고 복원해 주기 위해서 stack에 저장할 하는 것이다. 그리고 lr은 혹시 모를 상황으로 다른 함수로 branch를 할 상황을 대비해서 stack에 저장하는 것 이기도 하고 subroutine이 끝나고 lr의 값을 pc로 push 해서 원래의 프로그램 흐름으로 바로 branch하려는 목적이기도 하다.

STEP 15: Lines 30-32의 명령어들에서와 같이 메모리의 주소가 지정되는 이유를 그림 6.7 또는 Program 6.3을 근거로 확인해보자.

먼저 subroutine에 들어가기 전에 __main함수에서 line 9-12를 통해서 ARRAY_BASE, ARRAY_SIZE, 0x5555aaaa라는 데이터를 stack에 저장해주었다. 이때의 sp는 0x200000F4이다. 이 상태에서 subroutine에 들어와서 바로 r0-r3,lr 총 20byte의 데이터를 stack에 추가로 저장해주어 sp는 0x200000E0가 되었다. 이 상태에서 sp는 __main함수에서 stack에 저장하였을 때 보다 sp는 0x14만큼 더 감소하여 있다. 따라서 __main함수에서 저장해 두었던 ARRAY_BASE, ARRAY_SIZE, 0x5555aaaa라는 데이터에 접근하기 위해서는 현재 sp에서 0x14, 0x18, 0x1C만큼 offset을 더해줘야 접근할 수 있다. 따라서 이러한 근거를 바탕으로 lines 30-32와 같이 메모리의 주소를 지정한 것이다.

STEP 16: Line 38 명령어는 의미적으로 어떤 동작을 수행하는지 설명해보자.

Lines 38 명령어는 fill_array라는 subroutine이 모두 끝나고 마지막 명령어로 pop {r0-r3,pc}라는 명령어이다. 이는 subroutine이 처음 수행될 때 stack에 r0-r3,lr를 push하였으므로 subroutine의 마지막에 해당 데이터를 다시 pop해서 원래의 값으로 복원해주는 동작을 수행한다. 그리고 lr의 데이터를 stack에 push하였다가 pc에 pop함으로써 추가적인 branch 명령어 없이 프로그램의 pc를 이동시켜 원래의 프로그램 흐름으로 돌아가는 동작을 한다. 즉 line 38 명령어는 subroutine을 수행한 뒤 원래의 상태로 다시 돌아가는 동작을 수행한다.

STEP 17: Line 14(19) 를 통해 sp의 내용을 보정해 주지 않으면 어떤 문제가 발생할 수 있는지 설명해보자.

이번 lab6_3.s 코드에서 line 14와 line 19의 add sp,#12 명령어를 삭제하더라도 프로그램이 수행되는 데는 아무 문제가 발생하지 않고 동일한 동작을 수행한다. 다만 line 14와 line 19 명령어 없이 프로그램을 수행한 경우는 아래 [그림 31]처럼 sp값이 0x200000E8이 되어있다. 즉 stack에 의미있게 저장되어 있는 값은 전혀 없는데 sp가 초기값과 다르게 감소하였다. 즉 만약 이런 문제가 큰 프로그램에서 발생하거나 여러 번 발생하게 되면 stack의 공간을 점점 낭비하게 되어 stack공간이 부족해지고 나중에는 stack을 사용할 때 스택의 범위를 초과하는 stack overflow가 발생할 수 있다. 이렇게 되면 우리가 프로그램이 어떻게 동작할지 예상할 수 없다. 따라서 우리는 line 14와 line 19에 sp의 내용을 보정해주는 add sp,#12와 같은 명령어를 추가하여 stack을 사용한 뒤 sp를 원래의 위치로 보정해야 한다.

Register	Value
Core	
R0	0x080002C9
R1	0xE00ED08
R2	0x20000100
R3	0x00000008
R4	0x77337733
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x200000E8
R14 (LR)	0x080002E3
R15 (PC)	0x080002E2

그림 31 Line 14, 19 삭제 후 프로그램 수행 후 레지스터 값

STEP 18: fill_array와 fill_pointer subroutine의 수행을 통해 array를 index를 이용하여 access하는 방식과 pointer를 이용하여 access하는 방식이 assembly language를 통해 어떻게 구현되는지 확인한다. 또한 어떤 방식이 어떤 측면에서 더 유리하다고 할 수 있는지 설명해보자.

```

29:      push {r0-r3,lr}
0.056 us 0x080002E8 B50F      PUSH      {r0-r3,lr}
30:      ldr r0,[sp,#0x14] ;ARRAY_BASE
0.019 us 0x080002EA 9805      LDR       r0,[sp,#0x14]
31:      ldr r1,[sp,#0x18] ;ARRAY_SIZE
0.019 us 0x080002EC 9906      LDR       r1,[sp,#0x18]
32:      ldr r2,[sp,#0x1c] ;FILL_PATTERN
0.019 us 0x080002EE 9A07      LDR       r2,[sp,#0x1c]
33:      mov r3,#0
0.009 us 0x080002F0 F04F0300 MOV       r3,#0x00
34:      lpl str r2,[r0,r3, LSL #2]
0.148 us 0x080002F4 F8402023 STR       r2,[r0,r3,LSL #2]
35:      add r3,#1
0.074 us 0x080002F8 F1030301 ADD       r3,r3,#0x01
36:      cmp r3,r1
0.074 us 0x080002FC 428B      CMP       r3,r1
37:      blt lpl
0.204 us 0x080002FE DBF9      BLT       0x080002F4
38:      pop {r0-r3,pc}
39:      endp
40: ;
41: ;      fill_pointer(int *array, int size,
42: ;      {      int *p;
43: ;          for (p = &array[0]; p < &arra;
44: ;              *p = pattern;
45: ;      }
46: fill_pointer proc
0.074 us 0x08000300 BD0F      POP       {r0-r3,pc}

```

그림 32 fill_array disassembly 코드

```

47:      push {r0-r3,lr}
0.056 us 0x08000302 B50F      PUSH      {r0-r3,lr}
48:      ldr r0,[sp,#0x14] ;ARRAY_BASE
0.019 us 0x08000304 9805      LDR       r0,[sp,#0x14]
49:      ldr r1,[sp,#0x18] ;ARRAY_SIZE
0.019 us 0x08000306 9906      LDR       r1,[sp,#0x18]
50:      ldr r2,[sp,#0x1c] ;FILL_PATTERN
0.019 us 0x08000308 9A07      LDR       r2,[sp,#0x1c]
51:      add r3,r0,r1, LSL #2
0.009 us 0x0800030A EB000381 ADD       r3,r0,r1,LSL #2
52:      lp2 str r2,[r0],#4
0.148 us 0x0800030E F8402B04 STR       r2,[r0],#0x04
53:      cmp r0,r3
0.074 us 0x08000312 4298      CMP       r0,r3
54:      blt lp2
0.204 us 0x08000314 DBFB      BLT       0x0800030E
55:      pop {r0-r3,pc}
0.074 us 0x08000316 BD0F      POP       {r0-r3,pc}

```

그림 33 fill_pointer disassembly 코드

[그림 32]와 [그림 33]에서 각각 fill_array 함수와 fill_pointer 함수의 disassembly 코드를 확인할 수 있다. 각 코드의 앞부분과 뒷부분은 push 명령어와 pop 명령어 그리고 동작을 위한 기본적인 변수들을 레지스터로 로드하는 과정임으로 동일한 동작을 수행한다. 그리고 각각 r3에 c언어로 보았을 때 fill_array 함수에서는 반복문을 위한 변수 i, fill_pointer 함수에서는 배열의 끝 주소를 저장하였다가 반복문을 탈출할지 말지를 결정하는 부분으로 값을 주었다.

먼저 fill_array 함수의 loop를 살펴보면 r3가 증가하면서 배열의 크기보다 작다면 계속해서 r2에 저장되어 있는 pattern을 r3의 오프셋을 이용해서 접근하여 데이터를 저장한다. 그리고 매번 r3와 r1의 배열의 사이즈를 비교해서 반복문을 계속할지 종료할지 결정한다. 반면 fill_pointer 함수의 loop를 살펴보면 r0에는 배열의 시작주소를 r3에는 배열의 끝주소를 각각 저장하여 r0의 위치에 pattern을 저장하면서 동시에 r0까지 업데이트를 하고 r0와 r3를 비교하여 반복문을 계속할지 종료할지 결정하게 된다.

즉 fill_array 함수는 반복문의 조건을 위한 변수를 하나 더 사용하고 있는 것이다. 따라서 loop문을 구성하고 있는 코드를 보면 fill_array 함수는 4줄, fill_pointer 함수는 3줄로 코드의 사이즈 측면에서 fill_pointer 함수가 더 유리하다. 또한 위의 [그림 32]과 [그림 33]을 비교하여 보면 fill_array 함수의 loop문 수행시간은 0.5us이고 fill_pointer 함수의 loop문 수행시간은 0.426으로 훨씬 짧은 수행시간을 보여주고 있음으로 수행시간 측면에서도 fill_pointer 함수가 더 유리하다.

4) 실험 4

STEP 19: Program 6.3의 lines 22-56을 별도의 파일로 구성한 후 Program 6.3에서는 삭제한다.
이 두 파일을 포함하는 프로젝트를 생성한다.

```

1  STACK_BASE EQU 0x20000100
2  ARRAY_BASE EQU 0x20000100
3  ARRAY_SIZE EQU 0x8
4      area lab6_4_1, code
5      entry
6  __main proc
7      export __main [weak]
8      extern fill_array
9      extern fill_pointer
10 start ldr sp,=STACK_BASE
11      ldr r2,=ARRAY_BASE
12      ldr r3,=ARRAY_SIZE
13      ldr r4,=0x5555aaaa
14      push {r2-r4}
15      bl fill_array
16      add sp,#12
17      ;
18      ldr r4,=0x77337733
19      push {r2-r4}
20      bl fill_pointer
21      add sp,#12
22 stop  b stop
23      endp
24      end

```

그림 34 직접 작성한 lab6_4_1.s 코드

```

1      area lab6_4_2, code
2      export fill_array
3      export fill_pointer
4      ;
5      ;   fill_array(int array[], int size, int pattern)
6      ;   {
7      ;       for (i = 0; i < size; i += 1)
8      ;           array[i] = pattern;
9      ;   }
10 fill_array proc
11     push {r0-r3,lr}
12     ldr r0,[sp,#0x14] ;ARRAY_BASE
13     ldr r1,[sp,#0x18] ;ARRAY_SIZE
14     ldr r2,[sp,#0x1c] ;FILL_PATTERN
15     mov r3,#0
16 lp1  str r2,[r0,r3, LSL #2]
17     add r3,#1
18     cmp r3,r1
19     blt lp1
20     pop {r0-r3,pc}
21     endp
22     ;
23     ;   fill_pointer(int *array, int size, int pattern)
24     ;   {
25     ;       for (p = &array[0]; p < &array[size]; p = p + 1)
26     ;           *p = pattern;
27     ;   }
28 fill_pointer proc
29     push {r0-r3,lr}
30     ldr r0,[sp,#0x14] ;ARRAY_BASE
31     ldr r1,[sp,#0x18] ;ARRAY_SIZE
32     ldr r2,[sp,#0x1c] ;FILL_PATTERN
33     add r3,r0,r1, LSL #2
34 lp2  str r2,[r0],#4
35     cmp r0,r3
36     blt lp2
37     pop {r0-r3,pc}
38     endp

```

그림 35 직접 작성한 lab_6_4_2.s 코드

STEP 20: 그림 6.8을 통해 이해한 바에 따라 두 프로그램 파일에 적당한 directive를 추가한후 assemble/link한다. 이 과정에서 오류 메시지를 발견하면 그 의미를 파악하고 프로그램을 수정한다. 프로그램이 원래 Program 6.3과 동일하게 동작하는지 확인한다.

[그림 34]의 코드는 `_main` 함수의 코드이다. 이 코드에는 `fill_array` 함수와 `fill_pointer` 함수가 없지만 다른 파일에 들어있다는 것을 어셈블러에게 알려주기 위해서 `extern fill_array`와 `extern fill_pointer` 라는 directive를 추가하였다. 그리고 [그림 35]의 코드는 `fill_array` 함수와 `fill_pointer` 함수의 코드가 적혀 있는데 다른 함수에서 이 함수들을 사용할 수 있도록 `export fill_array`와 `export fill_pointer`라는 directive를 추가하여 주었다.

그리고 이 두 파일을 모두 하나의 프로젝트에 넣고 build 하여 주었다. 그 과정에서 오류 메시지는 발생하지 않았다. 그리고 아래 [그림 36]처럼 프로그램이 모두 수행된 뒤 메모리에 정상적으로 `fill_pointer` 함수에 의해서 `pattern`이 저장된 것을 확인할 수 있다. 즉 하나의 파일에 저장하였던 program 6.3과 동일하게 동작하는 것을 알 수 있다.

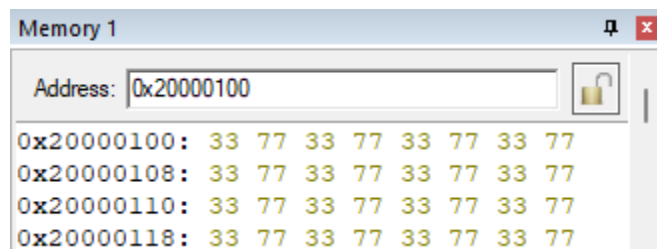


그림 36 lab6_4 수행 후 메모리 값


```
lab6_4_1.lst X
C: > Users > seokhun > Desktop > Coding > EEE4183_Microprocessor-Experiment > Week 6 > Listings > lab6_4_1.lst
10      2 00000000 20000100
11      |
12      |         ARRAY_BASE
13      |         EQU           0x20000100
14      3 00000000 00000008
15      |
16      |         ARRAY_SIZE
17      |         EQU           0x8
18      4 00000000          area    lab6_4_1, code
19      5 00000000          entry
20      6 00000000          __main  proc
21      7 00000000          export  __main [weak]
22      8 00000000          extern  fill_array
23      9 00000000          extern  fill_pointer
24     10 00000000 F8DF D01C
25      |
26      |         start  ldr      sp,=STACK_BASE
27     11 00000004 4A06    ldr      r2,=ARRAY_BASE
28     12 00000006 F04F 0308 ldr      r3,=ARRAY_SIZE
29     13 0000000A 4C06    ldr      r4,=0x5555aaaa
30     14 0000000C B41C    push     {r2-r4}
31     15 0000000E F7FF FFFE bl      fill_array
32     16 00000012 B003    add      sp,#12
33     17 00000014      ;
34     18 00000014 4C04    ldr      r4,=0x77337733
35     19 00000016 B41C    push     {r2-r4}
36     20 00000018 F7FF FFFE bl      fill_pointer
37     21 0000001C B003    add      sp,#12
38     22 0000001E E7FE    stop     b      stop
39     23 00000020      endp
40     24 00000020      end
```

그림 37 lab6_4_1.lst 파일

그리고 추가적으로 branch의 동작을 확인하기 위해서 lst파일을 확인하여 해당 bl fill_array와 bl fill_pointer가 어셈블리어로 어떻게 번역되었는지 확인해보았다. 그 결과는 [그림 37]과 같은데 두 명령어 모두 F7FF FFFE으로 기계어가 번역된 것을 알 수 있다. 아래 ARM Architecture Reference Manual에서 가져온 [그림 38]을 보면 BL 명령어의 format을 확인할 수 있는데 F7FF FFFE를 이 format에 대입해보면 imm값이 0x1FFFFE로 signed 숫자로 보면 -2로 LAB05에서 공부하였듯 (현재 명령어의 PC + 4 - (-2) * 2)임으로 다시 현재의 PC로 이동하는 명령어라고 해석할 수 있다. 이는 어셈블러가 lab6_4_1.s 코드를 컴파일 하는 과정에서 어셈블러가 fill_array와 fill_pointer라는 subroutine이 다른 파일에 들어있다는 것은 알지만 어떤 메모리 주소를 가지게 될지는 현재 시점

에서는 알 수 없기 때문에 다른 PC로 branch한다고 해석하는 대신 F7FF FFFE로 해석했다고 이해할 수 있다. 그리고 lab6_4_1.o 파일과 lab6_4_2.o 파일을 링킹하여 메모리에 실제 보드에 로드할 때 실제 물리적인 주소가 정해지게 되고 이때 링커가 offset을 계산하여 다시 b 명령어를 만들어 준다.

Encoding T1

ARMv4T, ARMv5T*, ARMv6*, ARMv7 if J1 == J2 == 1
ARMv6T2, ARMv7 otherwise

BL<c> <label>

Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	imm10										1	1	J1	1	J2	imm11										

```
I1 = NOT(J1 EOR S); I2 = NOT(J2 EOR S); imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

4. Exercises

1) 본 장을 통해 이해한 내용을 바탕으로 이제까지 프로그램에 포함되었던 `export __main [weak]` (예: Program 6.3의 line 5)의 의미를 어떻게 해석할 수 있겠는가? 이 단서를 프로젝트를 생성할 때 사용해온 startup codes에서 찾을 수 있겠는가?

실험 4에서 확인한 대로 `export` 명령어는 다른 파일에서 `__main` 함수를 호출할 수 있도록 만들어 주는 directive이다. 즉 보드에 `__main` 함수가 로드되어도 바로 수행되는 것이 아니라 다른 함수에 의해 호출되어 수행된다는 것을 예상할 수 있다. 또한 `[weak]` 속성의 뜻은 만약 링킹하는 과정에서 같은 symbol 이름을 가진 다른 `__main`이 존재한다면 다른 symbol이 더 높은 우선순위를 뜻한다는 의미이다. 즉 만약 `[weak]` 속성을 가진 `__main`과 그냥 `__main` 함수가 2개 있다면 링커는 `[weak]` 속성이 없는 `__main` 함수를 우선으로 사용한다는 뜻이다. 아래 [그림 38]은 `__main` 함수를 2개 만들어서 하나는 `[weak]` 속성을 주지 않았더니 처음 보드가 수행되고 `__main` 함수가 호출될 때 `[weak]` 속성이 없는 `__main` 함수로 branch한 것을 확인할 수 있었다.

```

10
11  __main proc
12      export __main
13  stop2 b stop2
14      endp
```

그림 38 다른 `__main` 함수로 branch

그리고 위에서 약간 말한대로 __main함수 또한 다른 함수에 의해 호출된다고 이해할 수 있는데 이는 보드가 처음 수행되고 나서 자동으로 수행되는 startup code가 __main함수를 호출한다.

```

135:                LDR    R0, =SystemInit
0x080000EC 4809    LDR    r0,[pc,#36] ; @0x08000114
136:                BLX    R0
0x080000EE 4780    BLX    r0
137:                LDR    R0, =__main
0x080000F0 4809    LDR    r0,[pc,#36] ; @0x08000118
138:                BX      R0
139:                ENDP

```

그림 39 startup code

위의 [그림 39]는 보드가 수행되고 나서 바로 수행되는 첫 명령어가 포함된 startup code이다. 간단히 해석해보면 맨 처음에는 SystemInit 함수를 호출하여 보드의 초기 설정을 해주고 다시 돌아와 __main함수를 bx 명령어로 호출하여 __main함수를 호출하여 우리가 작성한 프로그램을 수행하기 시작하는 것을 알 수 있다. 따라서 __main함수도 startup code가 적혀 있는 파일에서 접근할 수 있도록 export directive를 사용해주어야 한다. 그리고 한가지 더 생각해보면 startup code를 내가 수정하면 처음 호출되는 함수를 __main 함수가 아니라 다른 함수를 호출할 수도 있을 것이라 생각할 수 있는데 이를 직접 해보면 아래 [그림 40], [그림 41]과 같이 꼭 __main이라는 이름이 아니더라도 내가 startup code에서 다른 함수를 호출하도록 하면 다른 이름으로도 호출할 수 있었다.

```

131 Reset_Handler  PROC
132                EXPORT Reset_Handler            [WEAK]
133                IMPORT mymain
134                IMPORT SystemInit
135                LDR    R0, =SystemInit
136                BLX    R0
137                LDR    R0, =mymain
138                BX      R0
139                ENDP

```

그림 40 직접 수정한 startup code

```

4                area lab6_4_1, code
5                entry
6 mymain proc
7                export mymain [weak]
8                extern fill_array
9                extern fill_pointer
10 start ldr sp,=STACK BASE

```

그림 41 startup code에서 mymain을 호출

2) 본 장에서 소개된 Modular programming directives을 C 프로그래밍에서의

prototyping과 연관이어 생각해보자.

C언어에서 prototyping이란 코드에서 사용하고자 하는 함수의 형태를 미리 선언하여 해당 소스 코드에서 함수를 사용하고자 할 때 해당 함수의 형태를 기반으로 컴파일을 하기 위해 작성하는 것을 말한다. C언어에서 컴파일러는 코드를 위에서 아래로 순차적으로 컴파일 하는데 함수를 작성하는 순서에 따라서 해당 함수가 어떤 parameter와 return의 형식을 가지는지 알 수 없기 때문에 미리 함수의 형태를 prototyping하여 코드가 정상적으로 컴파일 될 수 있도록 해준다. 이에 대한 근거로 C reference 사이트에서 말한대로 prototype은 미래의 함수 호출에 대비해서 미리 함수의 형태를 선언하는 것을 의미한다.

New-style (C89) function definition. This definition both introduces the function itself and serves as a function prototype for any future function call expressions, forcing conversions from argument expressions to the declared parameter types.

- https://en.cppreference.com/w/c/language/function_definition

마찬가지로 이번 장에서 사용한 modular programming directives는 어떤 subroutine이 이 소스파일에서는 선언되어 있지 않지만 어셈블리에게 해당 symbol의 subroutine이 다른 파일에 선언되어 있다고 알려주어 정상적으로 컴파일 될 수 있도록 해주는 역할을 한다. 즉 C의 prototyping과 어셈블리의 export나 extern directive모두 컴파일러나 어셈블리에게 해당 함수가 다른 곳에 선언되어 있으니 걱정 말고 컴파일하라고 알려주는 역할을 한다.

3) 실험을 통해 확인한 내용을 바탕으로 stack overflow / underflow가 어떤 때 발생할 수 있는지 구체적으로 생각해보자. 이러한 상황의 발생이 왜 문제가 되는가?

Stack overflow는 스택을 push하다가 스택의 top이 end of stack을 지나서 주어진 스택 영역을 초과해서 데이터를 저장하였을 때의 상황을 의미한다. 그리고 stack underflow는 스택을 pop하다가 top이 bottom of stack을 지나서 주어진 스택의 영역을 초과해서 데이터를 로드하였을 때의 상황을 의미한다.

예를 들어 subroutine에서 stack을 사용할 때 데이터를 4개의 데이터를 push하고 3개의 데이터만 pop을 하는 과정을 계속하게 되면 해당 subroutine이 호출될 때 마다 sp가 4byte씩 감소하고 스택의 공간이 점점 낭비되게 된다. 그러다 보면 주어진 stack의 공간을 초과해서 데이터를 저장하게 되고 stack overflow가 발생하게 된다. 또는 stack에 push한 데이터 보다 pop을 더 많이 하는

subroutine이 있을 때 해당 subroutine이 호출되다 보면 stack pointer의 값이 계속해서 증가하고 stack의 시작주소를 초과된 영역을 침범하여 underflow가 발생한다.

이와 같은 overflow나 underflow가 발생하는 경우 원치 않는 장소에 데이터가 저장되거나 로드되는 경우가 발생할 수 있고 또한 다른 코드나 데이터를 삭제하는 등 프로그램이 예상할 수 없는 동작을 수행하게 된다.

4) Subroutine에 변수(parameter)를 전달하는 세가지 방식을 비교하여보자.

Subroutine에 변수를 전달하는 세가지 방식에는 레지스터를 이용한 방식, stack을 이용한 call by value parameter passing, 그리고 stack을 이용한 call by reference parameter passing 방법이 있다. 먼저 레지스터를 이용한 parameter passing은 함수를 호출하기 전 미리 원하는 레지스터에 parameter를 저장하고 subroutine을 호출하여 그대로 레지스터에 저장된 데이터를 사용하는 방식이다. 아래 [그림 43]의 ARM Procedure Call Standard(APCS)에 따르면 일반적으로 고급언어에서 parameter를 passing할 때 어셈블리 단계에서 보면 r0, r1, r2, r3를 통해서 argument 즉 parameter를 passing한다. 즉 레지스터를 이용해서 parameter를 passing하는 방식이 가장 기본적인 방식이다. 하지만 레지스터의 수가 제한되어 있어 많은 parameter를 passing하기에는 어려움이 있다.

Table 9-1 APCS registers

Register	APCS name	APCS role
r0	a1	argument 1/scratch register/result
r1	a2	argument 2/scratch register/result
r2	a3	argument 3/scratch register/result
r3	a4	argument 4/scratch register/result
r4	v1	register variable
r5	v2	register variable
r6	v3	register variable
r7	v4	register variable
r8	v5	register variable
r9	sb/v6	static base/register variable
r10	sl/v7	stack limit/stack chunk handle/register variable
r11	fp/v8	frame pointer/register variable
r12	ip	scratch register/new-sb in inter-link-unit calls
r13	sp	lower end of the current stack frame
r14	lr	link register/scratch register
r15	pc	program counter

그림 42 ARM Procedure Call Standard(APCS)

다음으로 stack을 이용해서 parameter의 value를 passing을 하는 방법이 있다. Subroutine을 호출하기 전에 stack에 passing하려는 parameter를 push하고 함수를 호출한 뒤 현재의 sp값과 offset을 통해서 stack에 저장된 값을 접근하여 사용하는 방식이다. 이 방식을 사용할 때의 장점은 stack 공간에 여유만 있다면 원하는 데이터를 전부 stack에 저장하여 subroutine에 전달할 수 있다는 장점이 있다. 단 전달하려는 데이터를 전부 stack에 저장해야 함으로 시간의 소요가 클 수 있다는 단점이 있다.

마지막으로 stack을 이용해서 parameter의 address를 passing하는 방법이 있다. 원하는 데이터가 저장되어 있는 공간의 주소를 stack에 push하고 subroutine 내부에서 해당 주소를 stack에서 참조하여 데이터가 저장되어 있는 공간으로 직접 접근해 데이터를 사용하는 방식이다. 이 방식의 장점은 주소 하나만 전달하여 수많은 데이터에 접근할 수 있도록 연결할 수 있음으로 직접 stack에 저장하지 않은 데이터를 접근할 수 있고 만약 주어진 parameter를 수정한 경우 해당 값이 직접 수정된다는 장점이 있다.

즉 레지스터를 이용한 방식은 가장 간단하고 빠른 방식이지만 자원이 한정되어 있고 stack을 통해 value passing은 stack에 직접 값을 저장해야 해서 느리지만 레지스터에 저장할 때 보다 많은 parameter를 넘길 수 있다는 장점이 있다. 마지막으로 stack을 통한 reference passing은 stack에 접근하려는 데이터의 주소를 parameter로 passing함으로써 많은 데이터에 접근할 수 있다는 장점이 있다.

5. 추가 실험

1) Fibonacci Sequence and Recursion Function

Subroutine의 장점은 parameter를 주고 그 값을 가지고 동작을 수행하는 코드를 한번의 선언으로 자유롭게 호출할 수 있다는 점이다. 이를 극대화해서 보여주는 예가 재귀함수이다. 재귀함수란 같은 함수 내에서 동일한 함수를 계속해서 호출하는 형태의 함수이다. 이때 가장 중요한 것이 정확한 parameter를 전달하는 것과 종료 조건을 잘 선언하는 것이다. 이러한 점에 유의해서 재귀함수를 가지고 피보나치 수열을 구하는 프로그램을 아래 [그림 43]과 같이 어셈블리어를 통해 작성해 보았다.

```

1  STACK_BASE EQU 0x20000100
2      area lab6_5,code
3      entry
4  __main proc
5      export __main [weak]
6  start ldr r0, data1
7      ldr sp, =STACK_BASE
8      bl fibo
9  stop  b stop
10     endp
11  fibo proc
12      push {r1, lr}
13      cmp r0, #1
14      blt return0
15      beq return1
16      mov r1, r0
17      sub r0, r1, #1
18      bl fibo
19      push {r0}
20      sub r0, r1, #2
21      bl fibo
22      pop {r1}
23      add r0, r1
24      pop {r1, pc}
25  return0 mov r0, #0
26          pop {r1, pc}
27  return1 mov r0, #1
28          pop {r1, pc}
29      align
30  data1 dcd 10
31      end

```

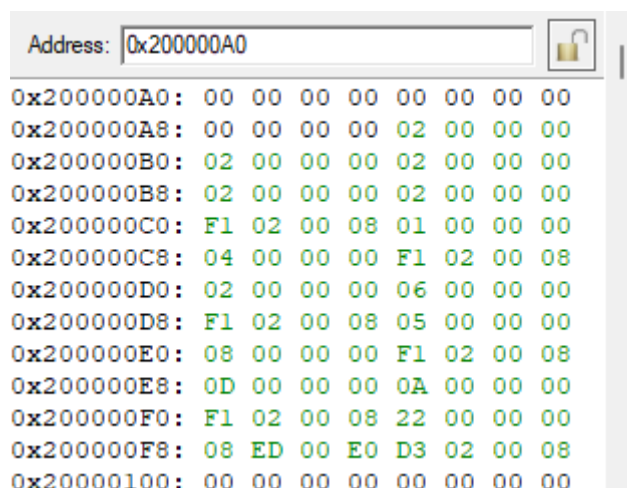
그림 43 직접 작성한 lab6_5.s 코드

이 코드를 간략히 설명하면 data1에 들어간 수를 이용해서 피보나치 수를 구하는 코드이다. fibo라는 subroutine이 피보나치 수를 구하는 함수인데 알고리즘은 간단하다. Subroutine을 호출할 때 r0를 통해서 피보나치 수를 구하는 파라미터를 전달하는데 r0의 값이 1보다 작으면 0임으로 0을 return하고 1과 같으면 1을 리턴한다. 그리고 나머지 경우는 피보나치 수를 구하기 위해서 n-2의 피보나치 수와 n-1의 피보나치 수를 구할 수 있도록 r1에 임시로 값을 저장해두고 r0에 n-1과 n-2를 각각 넣어서 subroutine을 재귀로 호출하는 방식을 사용하고 있다. 이때 subroutine에서 r1을 사용함으로 subroutine을 시작할 때 r1을 push하고 subroutine 종료 후 원래의 흐름으로 잘 돌아가기 위해서 lr도 push한다. 그리고 마지막에 r1, pc로 각각 pop하게 된다. 이러한 알고리즘을 이용해서 n=10일 때의 피보나치 수 55를 구한 결과는 아래 [그림 44]를 통해서 확인할 수 있다. 레지스터의 값이 16진수로 표현되기 때문에 r0의 0x37즉 55가 잘 구해진 것을 알 수 있다.

R0	0x00000037
R1	0xE000ED08

그림 44 10번째 피보나치 수를 구한 결과

이때 함수를 계속 재귀적으로 호출하다 보면 push를 계속 하게 되어서 stack overflow가 발생할 여지가 있다. 따라서 특정 n 에 대해서 stack이 얼마나 필요한지 확인해보기 위해 우선 $n=10$ 인 경우를 살펴보면 아래 [그림 45]가 $n=10$ 일 때 프로그램 수행 후 스택 메모리 공간의 데이터이다. 스택에서 pop을 하더라도 데이터가 사라지지 않기 때문에 가장 최대로 데이터를 저장했던 위치를 보면 0x200000AC로 0x20000100보다 0x54 즉 84byte만큼의 공간이 필요했다. 이에 대해 식을 구해보면 n 을 구할 때 subroutine을 최대 10번 호출되고 그 때 마다 8byte씩 stack이 증가한다. 그리고 stack 내부에서 4byte를 한 번 더 push하기 때문에 최대로 중첩되면 $(8*n+4)$ byte 만큼의 공간이 필요하게 된다. $n=10$ 인 경우를 계산해보면 $8*10+4$ 로 정확히 84byte가 나오는 것을 계산할 수 있다. 물론 이때 $n=0$ 이나 $n=1$ 인 경우 중첩으로 subroutine을 호출하지 않기 때문에 8byte의 stack만 필요하다. 즉 stack overflow없이 안전하게 stack을 사용하기 위해서는 최소 $(8*n+4)$ byte만큼의 stack 여유공간이 필요하다.



Address	00	00	00	00	00	00	00	00
0x200000A0:	00	00	00	00	00	00	00	00
0x200000A8:	00	00	00	00	02	00	00	00
0x200000B0:	02	00	00	00	02	00	00	00
0x200000B8:	02	00	00	00	02	00	00	00
0x200000C0:	F1	02	00	08	01	00	00	00
0x200000C8:	04	00	00	00	F1	02	00	08
0x200000D0:	02	00	00	00	06	00	00	00
0x200000D8:	F1	02	00	08	05	00	00	00
0x200000E0:	08	00	00	00	F1	02	00	08
0x200000E8:	0D	00	00	00	0A	00	00	00
0x200000F0:	F1	02	00	08	22	00	00	00
0x200000F8:	08	ED	00	E0	D3	02	00	08
0x20000100:	00	00	00	00	00	00	00	00

그림 45 프로그램 수행 후 스택 공간

2) C code, Assembly code Linking

어셈블리 코드는 기계어를 제외하고 가장 기계어와 가까운 언어이다. 여기서 가깝다는 의미는 CPU의 동작을 내가 원하는 대로 정확히 컨트롤 할 수 있다는 의미이다. 우리가 익히 아는 C언어 같은 경우는 어셈블리보다 높은 레벨의 언어로 우리가 작성한 C코드를 컴파일러가 번역해서 기계어로 만들어 주는데 아무래도 그 과정에서 100% 효율적으로 컴파일 하기에는 무리가 있다. 아래 [그림 46]은 아래 참고자료의 논문에서 가져온 그림으로 이 그림에서 확인할 수 있는 것처럼 어셈블리어가 C언어보다 runtime performance가 더 좋은 것을 확인할 수 있다.

따라서 최대의 performance를 위해서는 가능하다면 기계어와 가장 가까운 어셈블리어로 프로그래밍을 하는 것이 좋다. 하지만 어셈블리어로 코드를 작성하기에는 너무 많은 시간이 소요되고 코드의 생산성이 떨어진다는 단점이 있다. 따라서 각각의 장점과 단점을 잘 보완하며 필요에 따라 선택하여 코드를 작성해야 한다. 이를 위해서는 C로 작성한 코드와 어셈블리어로 작성한 코드를 각각 컴파일하여 링킹하는 과정이 필요한데 이번 추가실험에서 이에 관해서 탐구해보았다.

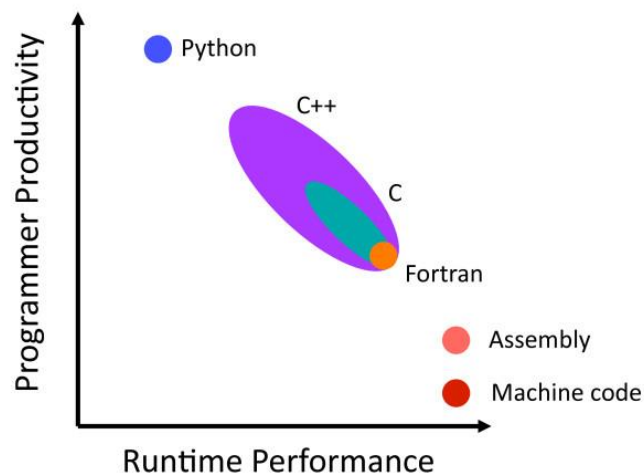
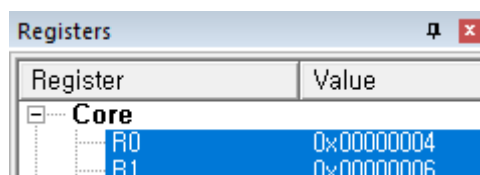


그림 46 어셈블리 vs high level 언어 성능 비교

<pre> 1 #include <stdio.h> 2 extern int add(int x, int y); 3 extern int fibo(int n); 4 5 int main(void) { 6 int x = 4; 7 int y = 6; 8 9 x = add(x, y); 10 x = fibo(x); 11 12 return 0; 13 }</pre>	<pre> 1 area lab6_6,code 2 add proc 3 export add 4 push {lr} 5 add r0,lr 6 pop {pc} 7 endp 8 9 fibo proc 10 export fibo 11 push {r1, lr} 12 cmp r0, #1 13 blt return0 14 beq return1 15 mov r1,r0 16 sub r0,r1,#1 17 bl fibo 18 push {r0} 19 sub r0,r1,#2 20 bl fibo 21 pop {r1} 22 add r0,r1 23 pop {r1, pc} 24 return0 mov r0, #0 25 pop {r1, pc} 26 return1 mov r0, #1 27 pop {r1, pc} 28 end</pre>
--	---

그림 47 직접 작성한 C, 어셈블리 코드

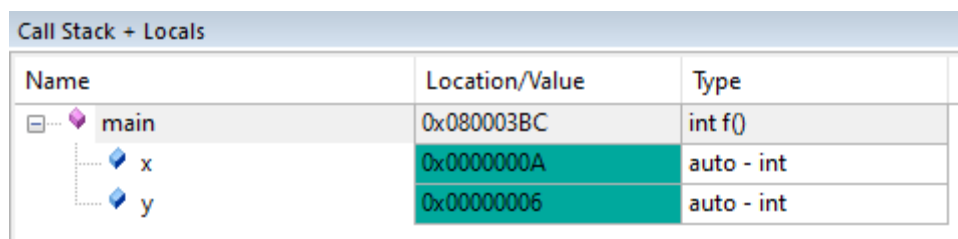
[그림 47]과 같이 C코드와 어셈블리 코드를 작성하였다. 먼저 C코드에서는 외부에 있는 함수를 사용하기 위해서 extern 명령어를 통해 prototyping을 해주고 main함수에서 해당 함수를 호출해서 사용하도록 해주었다. 그리고 어셈블리 코드로는 두 숫자를 더하는 add함수와 추가실험1에서 진행하였던 피보나치를 구하는 함수를 다시 재사용하였다. 이때 외부 파일에서 해당 함수들을 사용할 수 있도록 모두 export directive를 해주었다.



Register	Value
R0	0x00000004
R1	0x00000006

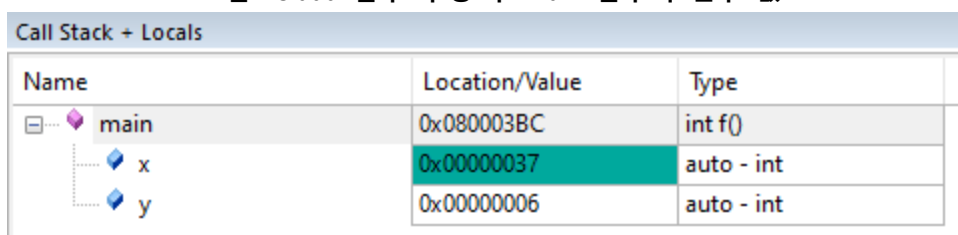
그림 48 add 함수 호출 후 레지스터 값

C코드의 main함수에서 어셈블리 코드에서 작성한 함수명과 똑같이 해서 함수를 호출하면서 x와 y의 값을 parameter passing해주었다. 그 결과 위의 exercises 4번에서 설명한 대로 APCS에 따라서 각 parameter가 r0, r1에 저장되어 전달된 것을 위 [그림 48]을 통해서 확인할 수 있다. 그리고 r0와 r1을 더해 r0에 저장하고 원래 함수로 return 한 결과 아래 [그림 49]와 같이 x값에 함수의 return값이 그대로 들어간 것을 확인할 수 있다. 그리고 동일하게 C코드를 통해서 원하는 데이터를 fibo함수의 parameter로 넘겨주었더니 피보나치 수를 구해서 결과를 x값에 저장한 것을 아래 [그림 50]을 통해서 확인할 수 있다.



Name	Location/Value	Type
main	0x080003BC	int f()
x	0x0000000A	auto - int
y	0x00000006	auto - int

그림 49 add 함수 수행 후 main 함수의 변수 값



Name	Location/Value	Type
main	0x080003BC	int f()
x	0x00000037	auto - int
y	0x00000006	auto - int

그림 50 fibo 함수 수행 후 main 함수의 변수 값

이번 추가실험을 통해서 어셈블리어와 C코드를 같이 사용해서 코드를 작성하고 프로그램을 수행하는 방법을 알아보았다. 원래 근본적으로는 C파일을 오브젝트 파일로 컴파일 하고, 어셈블리파일도 동일하게 어셈블러를 통해 오브젝트 파일로 컴파일 한 뒤, 해당 오브젝트 파일을 map파일을

통해 배치시킨 뒤 링킹해주는 과정이 필요하지만 우리가 사용하는 Keil IDE에서는 자동으로 이 과정을 진행해주었기 때문에 편하게 코드를 합칠 수 있었다. 따라서 앞으로 C언어로 코드를 작성하다가 필요에 따라서 더 최적화시키고 싶은 부분은 직접 어셈블리어로 작성하여 코드의 효율성을 높일 수 있게 되었다.

3) Stack Space Collision

```
1  STACK_BASE EQU 0x20000100
2      area lab6_7,code
3      entry
4  __main proc
5      export __main [weak]
6  start ldr sp,=STACK_BASE
7      ldr r0,=data
8      push {r0}
9      bl copy
10 stop b stop
11      endp
12
13 copy proc
14     push{r0-r2,lr}
15     ldr r1,[sp,#0x10]
16     adr r0,input
17 loop ldr r2,[r0]
18     cmp r2,#0
19     beq return
20     str r2,[r1]
21     add r1,#4
22     add r0,#4
23     b loop
24 return pop {r0-r2,pc}
25     endp
26 danger proc
27     mov r0,#0
28     mov r1,#0
29     mov r2,#0
30     b .
31     ltorg
32 input dcd 0x080002F1, 0x080002F1, 0x080002F1,
33     area lab6data,data
34 data space 20
35     end
```

그림 51 직접 작성한 lab6_7.s 코드

Stack을 사용하면서 주의해야 할 점으로는 반드시 스택 영역이 침범당하지 않도록 주의해야 한다는 것이다. 만약 stack 영역이 침범당하는 경우에 발생할 수 있는 문제점을 [그림 51]과 같이 코드를 통해서 구현해 보았다. 간단히 설명하면 copy라는 함수의 목적은 4byte input의 데이터를 복

사해서 data영역에 저장하는 것이다. 따라서 copy함수를 호출하면서 data의 주소를 parameter로 전달하였다. 하지만 이때 만약 input데이터가 4byte를 넘어 이번 경우에는 100byte의 데이터를 input으로 넣어주었더니 copy함수가 종료되고 return하려고 stack에 저장해두었던 데이터가 수정되었다. 따라서 원래 돌아가려 했던 주소를 잃어버리고 0x080002F1번지로 돌아가게 되었고 그 영역은 이번 프로그램에서 사용되지 않고 선언만 되어 있던 danger 영역으로 들어가게 되었다. 이에 대한 결과는 [그림 52]와 [그림 53]을 통해서 확인 가능하다.

Memory 1	Memory 1
Address: 0x20000000	Address: 0x20000000
0x20000000: 00 00 00 00 00 00 00 00	0x20000000: F1 02 00 08 F1 02 00 08
0x20000008: 00 00 00 00 00 00 00 00	0x20000008: F1 02 00 08 F1 02 00 08
0x20000010: 00 00 00 00 00 00 00 00	0x20000010: F1 02 00 08 F1 02 00 08
0x20000018: 00 00 00 00 00 00 00 00	0x20000018: F1 02 00 08 F1 02 00 08
0x20000020: 00 00 00 00 00 00 00 00	0x20000020: F1 02 00 08 F1 02 00 08
0x20000028: 00 00 00 00 00 00 00 00	0x20000028: F1 02 00 08 F1 02 00 08
0x20000030: 00 00 00 00 00 00 00 00	0x20000030: F1 02 00 08 F1 02 00 08
0x20000038: 00 00 00 00 00 00 00 00	0x20000038: F1 02 00 08 F1 02 00 08
0x20000040: 00 00 00 00 00 00 00 00	0x20000040: F1 02 00 08 F1 02 00 08
0x20000048: 00 00 00 00 00 00 00 00	0x20000048: F1 02 00 08 F1 02 00 08
0x20000050: 00 00 00 00 00 00 00 00	0x20000050: F1 02 00 08 F1 02 00 08
0x20000058: 00 00 00 00 00 00 00 00	0x20000058: F1 02 00 08 F1 02 00 08
0x20000060: 00 00 00 00 00 00 00 00	0x20000060: F1 02 00 08 F1 02 00 08
0x20000068: 00 00 00 00 00 00 00 00	0x20000068: F1 02 00 08 F1 02 00 08
0x20000070: 00 00 00 00 00 00 00 00	0x20000070: F1 02 00 08 F1 02 00 08
0x20000078: 00 00 00 00 00 00 00 00	0x20000078: F1 02 00 08 F1 02 00 08
0x20000080: 00 00 00 00 00 00 00 00	0x20000080: F1 02 00 08 F1 02 00 08
0x20000088: 00 00 00 00 00 00 00 00	0x20000088: F1 02 00 08 F1 02 00 08
0x20000090: 00 00 00 00 00 00 00 00	0x20000090: F1 02 00 08 F1 02 00 08
0x20000098: 00 00 00 00 00 00 00 00	0x20000098: F1 02 00 08 F1 02 00 08
0x200000A0: 00 00 00 00 00 00 00 00	0x200000A0: F1 02 00 08 F1 02 00 08
0x200000A8: 00 00 00 00 00 00 00 00	0x200000A8: F1 02 00 08 F1 02 00 08
0x200000B0: 00 00 00 00 00 00 00 00	0x200000B0: F1 02 00 08 F1 02 00 08
0x200000B8: 00 00 00 00 00 00 00 00	0x200000B8: F1 02 00 08 F1 02 00 08
0x200000C0: 00 00 00 00 00 00 00 00	0x200000C0: F1 02 00 08 F1 02 00 08
0x200000C8: 00 00 00 00 00 00 00 00	0x200000C8: F1 02 00 08 F1 02 00 08
0x200000D0: 00 00 00 00 00 00 00 00	0x200000D0: F1 02 00 08 F1 02 00 08
0x200000D8: 00 00 00 00 00 00 00 00	0x200000D8: F1 02 00 08 F1 02 00 08
0x200000E0: 00 00 00 00 00 00 00 00	0x200000E0: F1 02 00 08 F1 02 00 08
0x200000E8: 00 00 00 00 00 00 00 20	0x200000E8: F1 02 00 08 F1 02 00 08
0x200000F0: 08 ED 00 E0 00 10 02 40	0x200000F0: F1 02 00 08 F1 02 00 08
0x200000F8: D5 02 00 08 00 00 00 20	0x200000F8: F1 02 00 08 00 00 00 20
0x20000100: 00 00 00 00 00 00 00 00	0x20000100: 00 00 00 00 00 00 00 00

그림 52 데이터 침범 전/후 메모리 영역

```

26: danger proc
0x080002EE BD07 POP {r0-r2,pc}
27: mov r0,#0
0x080002F0 F04F0000 MOV r0,#0x00
28: mov r1,#0
⇒ 0x080002F4 F04F0100 MOV r1,#0x00
29: mov r2,#0
0x080002F8 F04F0200 MOV r2,#0x00
30: b .

```

그림 53 danger 함수가 수행된 그림

4) Nonlocal Jump

Nonlocal jump는 일반적으로 subroutine을 호출하고 리턴하는 것과 별개로 현재의 진행상황에서 다른 함수로 바로 이동하는 것을 말한다. 일반적으로 다시 돌아오려는 흐름을 버퍼에 저장해 두었다가 나중에 필요할 때 호출하여 다시 원래의 흐름으로 돌아온다. 보통 함수의 함수의 함수를 계속해서 들어가다가 오류가 발생하거나 하여 다시 맨 처음의 흐름으로 돌아오려고 할 때 nonlocal jump를 이용해서 계속해서 return하지 않고 한 번에 돌아올 수 있도록 해준다.

```
1  STACK_BASE EQU 0x20000100
2  MY_BUFFER EQU 0x20000128
3      area lab6_8,code
4      entry
5  __main proc
6      export __main [weak]
7  start ldr sp,=STACK_BASE
8      mov r0,#0
9      bl setj
10     bl subl
11 stop  b stop
12     endp
13 setj proc
14     push{r0,lr}
15     mov r0,sp
16     ldr sp,=MY_BUFFER
17     push{r0-r8,lr}
18     mov sp,r0
19     pop{r0,pc}
20     endp
21 longj proc
22     ldr sp,=MY_BUFFER
23     sub sp,#0x28
24     pop{r0-r8,pc}
25     endp
26 subl proc
27     cmp r0,#5
28     beq longj
29     add r0,#1
30     bl sub2
31     endp
32 sub2 proc
33     cmp r0,#5
34     beq longj
35     add r0,#1
36     bl sub3
37     endp
38 sub3 proc
39     cmp r0,#5
40     beq longj
41     add r0,#1
42     bl subl
43     endp
44 end
```

그림 54 직접 작성한 lab6_8.s 코드

위 [그림 54]의 코드를 보면 setj는 MY_BUFFER라는 공간에 새로 stack을 받아서 nonlocal jump만을 위한 buffer를 만들어 주었다. 그리고 setj함수는 setj 함수가 호출되었을 때의 레지스터 값과 lr을 모두 MY_BUFFER에 push해준다. 그리고 나중에 longj함수가 호출되면 해당 버퍼에서 값을 다시 pop해서 가져온다. 그리고 간단한 테스트를 위해 임시로 sub1, sub2, sub3를 호출해서 서로가 서로를 계속해서 호출하도록 만들었다.

그리고 동작을 테스트하기 위해 위 코드에서 약간 수정을 가해서 sub1함수로 들어가서 100번동안 서로 함수를 호출하도록 해주었다. 그 결과 nonlocal jump가 있을 때의 원래 흐름으로 돌아오는 소요시간은 아래 [그림 55]와 같고 함수마다 pop을 해서 돌아온 경우의 소요시간은 아래 [그림 56]과 같다. 이를 통해서 필요한 경우에 따라 nonlocal jump를 이용하면 약간의 추가적인 공간만 있다면 빠르게 원래의 흐름으로 돌아올 수 있는 것을 확인할 수 있었다.

Internal	Thread
Mode	Privileged
Privilege	MSP
Stack	891
States	0,00003140
Sec	

그림 55 nonlocal jump 없이 return시 소요시간

Internal	Thread
Mode	Privileged
Privilege	MSP
Stack	1481
States	0,00003686
Sec	

그림 56 nonlocal jump로 return시 소요시간

5) Queue

스택과 유사한 자료구조로 queue가 있다. Queue는 FIFO(First In First Out)의 구조를 가진 자료구조로 먼저 삽입된 데이터가 먼저 나오는 자료구조이다. 큐를 이용하면 간단한 OS 스케줄링이나 BFS와 같은 알고리즘에 사용되거나 프린터와 같이 먼저 들어온 순서대로 출력해야 하는 경우에 활용할 수 있는 매우 유용한 자료구조이다.

큐를 구현하기 위해서는 데이터가 들어가는 부분과 가장 먼저 들어간 데이터를 가리키고 있는 부분이 필요하다. 이를 위해서 아래 [그림 57]과 같이 코드를 작성하였다. 코드에서 r7은 데이터가 들어가는 부분의 위치를 담당하고 있고 r8은 데이터가 나가는 부분의 위치를 담당하고 있다. enqueue함수는 input에서 데이터를 순서대로 읽어들이며 큐에 저장하고 deque함수는 큐에 저장된 데

이터에서 순서대로 data영역에 저장하는 역할을 하고 있다. 이때 스택과 다르게 r7과 r8모두 둘 다 같은 방식의 address를 증가시키거나 감소시켜야 한다. 프로그램에서 enqueue와 deque의 순서를 섞어서 사용하더라도 정상적으로 데이터가 복사된 것을 [그림 58]을 통해서 확인할 수 있다. 이를 통해서 기본적인 큐의 동작을 살펴보고 나중에 필요에 따라 enqueue와 deque를 수정해서 사용하면 된다.

```
1  QUEUE_BASE EQU 0x20000100
2      area lab6_9,code
3      entry
4  __main proc
5      export __main [weak]
6  start ldr r7,=QUEUE_BASE
7      ldr r8,=QUEUE_BASE
8      adr r0,input
9      ldr r1,=data
10     bl enqueue
11     bl enqueue
12     bl enqueue
13     bl deque
14     bl enqueue
15     bl deque
16     bl deque
17     bl deque
18 stop b stop
19     endp
20 enqueue proc
21     ldr r2,[r0],#4
22     stmia r7!,{r2}
23     bx lr
24     endp
25 deque proc
26     ldmbia r8!,{r2}
27     str r2,[r1],#4
28     bx lr
29     endp
30     align
31 input dcd 0x12345678, 0x23456789, 0x3456789a, 0x456789ab
32     area lab6data,data
33 data space 16
34     end
```

그림 57 직접 작성한 lab6_9.s 코드

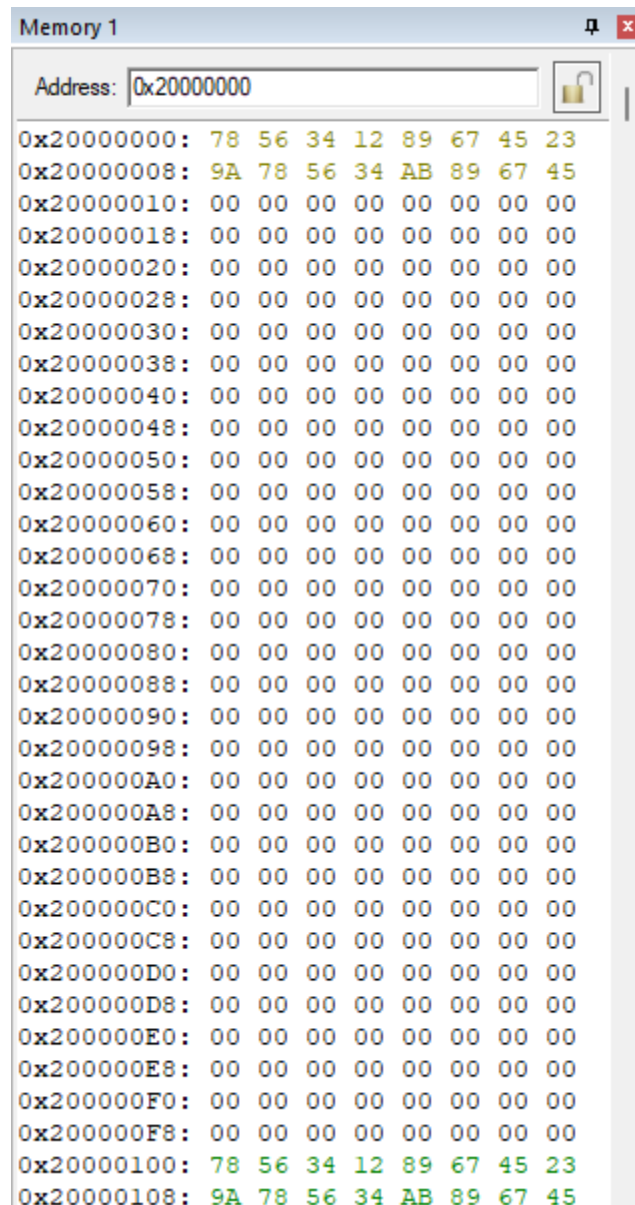


그림 58 프로그램 수행 후 메모리 값

6. 결론

이번 6주차 실험을 통해서 프로그래밍에서 가장 중요하다고도 생각할 수 있는 subroutine에 대해서 배웠다. 이와 관련해 stack을 다루는 방법부터 stack이 어떻게 동작하는지 stack을 사용할 때 주의해야 할 점까지 모두 실험을 통해서 배울 수 있었다. 또한 subroutine뿐만 아니라 다른 파일을 하나의 프로그램으로 합쳐서 동작하도록 하는 것을 배웠고 어셈블리 + 어셈블리뿐만 아니라 추가 실험에서 어셈블리 + C언어 코드도 합쳐서 수행을 해보았다. 가장 많이 배운 부분은 보통 고급언어로 작성하다 보니 쉽게 지나쳤던 subroutine이 호출될 때마다 stack에 레지스터에 들어있

던 내용을 push하고 pop하는 코드까지 직접 작성하였는데 이를 통해서 stack의 중요성에 대해 배울 수 있었고 subroutine의 동작도 정확하게 이해할 수 있었다.

7. 참고문헌

서강대학교 전자공학과 (2023). 마이크로프로세서개론.

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

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

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

ARM. (2011). ARM Architecture Reference Manual Armv7-A and ARMv7-R edition.

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

히언. (2021). EMBEDDED RECIPES.

Ralf W Grosse-Kunsteve & Thomas Charles Terwilliger & Nicholas Sauter & Paul D Adams. (2012).

Automatic Fortran to C++ conversion with FABLE.