

강의명 : 마이크로프로세서

실습 번호 : 4

실습 제목 : Assembler programming

학생 이름 : 이학민

학번 : 201910906

## 1. ARM 어셈블러 함수 - 1

### 1.1

memset\_asm:

```
.L0 : cmp r0, r1
      bgt .L1
      strb r2, [r0]
      add r0, r0, #1
      b .L0
```

```
.L1 : mov r0, r2
      bx lr
```

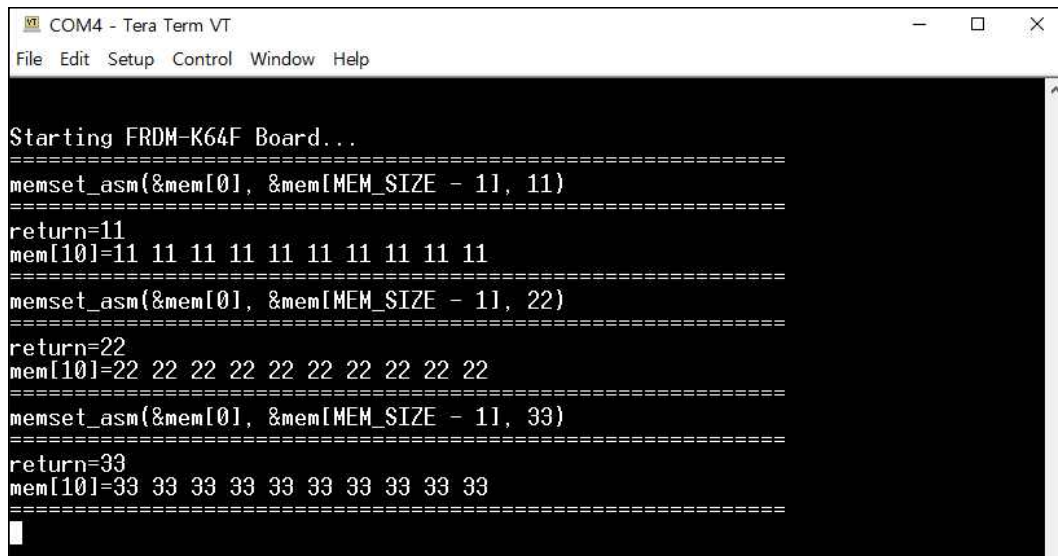
#### <알고리즘 설명>

우선 r0는 start의 주소를 담는 용도와 c의 값을 반환하는 용도, r1은 end의 주소를 담는 용도, r2는 c의 값을 담는 용도로 사용하였다. memset\_asm:은 label이다. r0와 r1을 비교하는 과정을 통해 start의 주소와 end의 주소 중 어떤 값이 더 큰지 판단한다. 만약 start의 주소값이 end의 주소값보다 크다면 c를 return 해야하므로 .L1으로 바로 이동하도록 한다. mov r0, r2를 통해 c의 값을 r0에 넣고 bx lr 명령어를 통해 c의 값을 return 할 수 있다. 만약 start의 주소값이 end의 주소값보다 작다면 r0가 가리키는 메모리에 가서 r2의 값을 저장한다. 또한 add 명령어를 통하여 r0를 1 증가시킨다. 앞서 설명한 과정을 .L1으로 이동하는 조건이 만족할 때까지 되풀이 해야하므로 branch 명령어를 수행하여 .L0로 돌아갈 수 있도록 한다.

### 1.2

함수를 프로그래밍 하는 과정에서 r0, r1, r2 총 3개의 레지스터를 사용하였다. 이 3개의 레지스터는 인수 및 return 값을 저장하는 레지스터로 계산에 사용이 가능하다. 이 레지스터들은 다른 특별한 용도로 사용되지 않으므로 corrupt하여 사용자의 용도에 맞게 사용이 가능하다. 또한 함수 return 시 r0가 return 값을 가지고 있는데, r0를 올바르게 return 하였음을 실습 결과를 통해 알 수 있었다. 따라서 사용한 모든 레지스터는 ATPCS의 기준에 따라 바르게 사용되었다.

### 1.3



```
COM4 - Tera Term VT
File Edit Setup Control Window Help

Starting FRDM-K64F Board...
=====
memset_asm(&mem[0], &mem[MEM_SIZE - 11], 11)
=====
return=11
mem[10]=11 11 11 11 11 11 11 11 11 11
=====
memset_asm(&mem[0], &mem[MEM_SIZE - 11], 22)
=====
return=22
mem[10]=22 22 22 22 22 22 22 22 22 22
=====
memset_asm(&mem[0], &mem[MEM_SIZE - 11], 33)
=====
return=33
mem[10]=33 33 33 33 33 33 33 33 33 33
=====
```

## 2. ARM 어셈블러 함수 - 2

### 2.1

multi\_asm :

```
    push {r4}
    mov r4, #2
    mov r2, r0
    mov r3, r1

.L0 : cmp r4, r3
      bgt .L1
      add r2, r2, r0
      add r3, r3, #-1
      b .L0

.L1 : mov r0, r2
      pop {r4}
      bx lr
```

#### <알고리즘 설명>

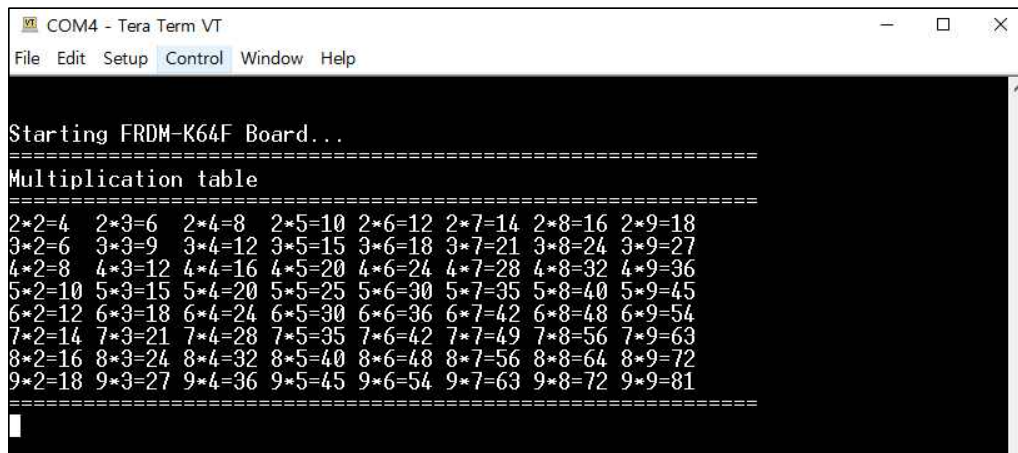
우선 r0는 i의 값을 담는 용도와 곱셈의 결과를 반환하는 용도, r1은 j의 값을 담는 용도, r2는 변화하는 i값을 저장하는 용도, r3는 변화하는 j의 값을 저장하는 용도, r4는 임의의 상수 값을 저장하는 용도로 사용하였다. 첫 줄의 multi\_asm : 은 label이다. MUL 명령어를 사용하지 않는 조건이므로 덧셈을 여러 번 반복하는 과정을 통해 곱셈의 결과를 도출하는 방식을 택하였다. r4를 사용하기 위해서 push {r4}를 통해 이전 함수에서 사용되던 레지스터 값을 스택에 저장해야 하고 함수 return 시 pop {r4}를 수행하여 스택에 저장해 두었던 레지스터 값을 복구한다. 다음 r4에 상수 2를 저장한다. 상수 2(r4)와 변화하는 j의 값(r3)을 이용하여 r4가 더 커질 때 .L1으로 이동하도록 하였다. 만약 그렇지 않다면  $r2 = r2 + r0$ 를 통하여 덧셈 연산을 수행한다. 덧셈을 한번 수행하였으므로 j의 값을 1 차감한다. branch 명령어를 통해서 다시 .L0로 돌아가서 앞서 설명한 과정을 반복한다.  $j=1$ , 즉 r3가 1이 될 때 r4보다 작아지므로 .L1으로 이동할 것이며 이 때 저장된 r2의 값이 곱셈의 결과와 같으므로 r2의 값을 r0에 저장하고 r0를 return한다.

### 2.2

함수를 프로그래밍 하는 과정에서 r0, r1, r2, r3, r4 총 5개의 레지스터를 사용하였다. 이 중 r0~r3 4개의 레지스터는 인수 및 return 값을 저장하는 레지스터로 계산에 사용이 가능하다. 이 레지스터들은 다른 특별한 용도로 사용되지 않으므로 corrupt하여 사용자의 용도에 맞게 사용이 가능하다. 또한 함수 return 시 r0가 return 값을 가지고 있는데, r0를 올바르게 return 하였음을 실습 결과를 통해 알 수 있었다. 하지만 r4의 경우 함수 내에서 corrupt 되기 때문에 corrupt를 방지하기 위하여 push {r4}를 통해 이전 함수에서 사용되던 레지스터

값을 스택에 저장하였고 함수 return 시 pop {r4}를 수행하여 스택에 저장해 두었던 레지스터 값을 원래대로 복구하였다. 결과적으로 사용한 모든 레지스터는 ATPCS의 기준에 따라 바르게 사용되었다.

## 2.3



```
COM4 - Tera Term VT
File Edit Setup Control Window Help

Starting FRDM-K64F Board...
=====
Multiplication table
=====
2*2=4 2*3=6 2*4=8 2*5=10 2*6=12 2*7=14 2*8=16 2*9=18
3*2=6 3*3=9 3*4=12 3*5=15 3*6=18 3*7=21 3*8=24 3*9=27
4*2=8 4*3=12 4*4=16 4*5=20 4*6=24 4*7=28 4*8=32 4*9=36
5*2=10 5*3=15 5*4=20 5*5=25 5*6=30 5*7=35 5*8=40 5*9=45
6*2=12 6*3=18 6*4=24 6*5=30 6*6=36 6*7=42 6*8=48 6*9=54
7*2=14 7*3=21 7*4=28 7*5=35 7*6=42 7*7=49 7*8=56 7*9=63
8*2=16 8*3=24 8*4=32 8*5=40 8*6=48 8*7=56 8*8=64 8*9=72
9*2=18 9*3=27 9*4=36 9*5=45 9*6=54 9*7=63 9*8=72 9*9=81
=====
```

### 3. ARM 어셈블러 함수 - 3

#### 3.1

fact\_asm:

```
    push {lr}
    push {r4-r7}
    mov r4, r0
    mov r7, r1

.L0 : cmp r4, r7
      bgt .L3
      mov r5, #2
      mov r6, #1
      ldr r0, =string

.L1 : cmp r5, r4
      bgt .L2
      mul r6, r6, r5
      add r5, r5, #1
      b .L1

.L2 : mov r1, r4
      mov r2, r6
      bl printf
      add r4, r4, #1
      b .L0

.L3 : pop {r4-r7}
      pop {pc}
```

string :

```
.asciz "fact(%d)=%d\n"
```

#### <알고리즘 설명>

fact\_asm:은 레이블이다. printf 함수를 호출하면 link register가 override되므로 함수 시작 시 push {lr}을 수행하고 함수 return 시 pop {pc}를 수행한다. 현재 함수에서 r4, r5, r6, r7을 사용할 것이므로 push {r4-r7}을 수행하여 이전 function에서 사용되던 값을 stack에 저장하고 함수 return 시 pop {r4-r7}을 수행하여 이들 값을 복구한다. 함수의 첫 번째 인수 from의 값을 담고 있는 r0와 두 번째 인수 to의 값을 담고 있는 r1은 corrupt 될 것이므로 corrupt를 방지하기 위한 역할을 수행하는 레지스터인 r4와 r7에 각각 저장해둔다. 다음은

팩토리얼 계산과 출력 과정에 대한 설명이다.

첫 번째로 .L0에 대한 설명이다. r4는 from의 값을 담은 레지스터로, factorial 결과를 출력하고 나면 1이 더해진다. r4의 값이 계속 커지다가 to의 값을 담은 레지스터인 r7의 값보다 커져버리면 .L3로 jump한다. r5와 r6은 factorial 계산을 하기 위한 레지스터로 사용된다. 특히 r6은 factorial 계산의 최종 결과값을 저장하는 역할을 동시에 맡는다. 만약 jump 조건을 만족하지 않을 경우, r5에 상수 2를 대입, r6에 상수 1을 대입하여 factorial 계산을 위한 준비를 한다. 또한 출력 시 printf에서 첫 번째 인수로 fact(%d)=%d\n을 넣어주어야 하므로 ldr 명령어를 수행한다.

두 번째로 .L1에 대한 설명이다. 이 부분은 factorial 값을 계산하는 역할이다. r5는 초기에 상수 2의 값을 가지고, factorial 계산이 부분적으로 수행될 때마다 1씩 증가한다. r5가 증가하다가 r4의 값을 넘어 분기 조건을 만족하면 .L2로 jump한다. 만약 jump 하지 않을 경우 factorial의 일부분을 계산하고 결과 값을 r6에 저장한다. 예를 들어 r4=3 이라고 가정하면,  $2=1 \times 2$ ,  $6=2 \times 3$ 의 과정을 통해 fact(3)=6임을 도출해낸다.

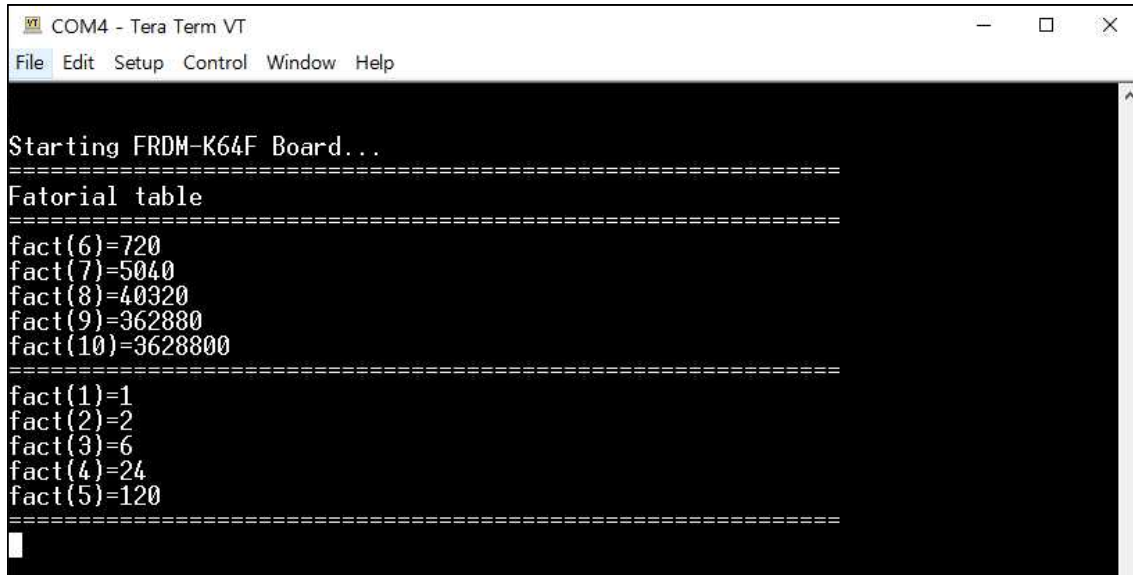
세 번째로 .L2에 대한 설명이다. .L1에서 factorial 계산이 끝까지 완료되면 출력을 하는 역할을 담당한다. printf의 첫 번째 인수에는 string을 이미 .L0에서 대입해주었으므로 두 번째와 세 번째 인수를 불러와야 한다. printf의 두 번째 인수는 r1인데 이 값은 현재 r4에 저장되어 있으므로 r4의 값을 r1에 대입한다. printf의 세 번째 인수는 r2인데 이 값은 현재 r6에 저장되어 있으므로 r6의 값을 r2에 대입한다. bl printf를 통해 출력하고자 하는 문구를 출력한다. 다음 factorial 계산을 위해 r4값에 1을 더해주고 .L0로 돌아간다.

마지막으로 .L3에 대한 설명이다. 모든 factorial 계산이 끝나고 출력을 마치면 함수를 종료하기 위한 역할을 담당한다. pop {r4-r7}을 통해 r4-r7 레지스터의 값을 복구하고 pop {pc}를 통해 함수를 종료한다.

### 3.2

함수 내에서는 r0, r1, r2, r4, r5, r6, r7, r14(=lr)가 corrupt 되었다. 그 중 r2는 factorial 계산을 다시 수행하여 결과가 바뀔 때마다 계속 값이 변하는 레지스터로 사용되었다. r2의 특별한 사용 용도가 없기 때문에 자유롭게 r2를 corrupt하여 타 용도로 사용할 수 있다. r0는 printf 실행 시 string을 가지고 있어야 하므로 corrupt 되는데 원래의 r0 값을 r4에 복사해 두어 이용함으로써 발생할 수 있는 문제를 예방하였다. r1도 마찬가지로 printf 실행 시 다른 값을 가지고 있어야 하므로 corrupt 되는데 원래의 r1 값을 r7에 복사해 이용하였다. r5와 r6도 임의의 상수를 대입하는데 사용하였다. 따라서 r4-r7의 corrupt를 방지하기 위하여 이전 function에서 사용되던 이 레지스터의 값들을 push {r4-r7}을 통해 스택에 저장해두고 함수 return 시 pop {r4-r7}을 수행하여 원래 값을 복구하였다. bl printf가 수행되면 r14(=lr)가 override되므로 push {lr}을 수행하여 스택에 원래의 값을 저장해두고 함수 return 시 pop {pc}를 통해 원래의 값을 복구하였다. 이와 같이 사용한 모든 레지스터는 ATPCS의 기준에 따라 바르게 사용되었다.

### 3.3



A screenshot of a Tera Term VT window titled "COM4 - Tera Term VT". The window has a menu bar with "File", "Edit", "Setup", "Control", "Window", and "Help". The main display area is black with white text. The text shows the process of starting an FRDM-K64F board and displaying a factorial table. The table lists factorial values for numbers 1 through 10, with values for 6 through 10 separated by dashed lines from the values for 1 through 5.

```
Starting FRDM-K64F Board...  
=====
```

Fatorial table	
fact(6)=720	
fact(7)=5040	
fact(8)=40320	
fact(9)=362880	
fact(10)=3628800	
=====	
fact(1)=1	
fact(2)=2	
fact(3)=6	
fact(4)=24	
fact(5)=120	
=====	

끝.