어셈블리프로그래밍설계 및 실습 보고서

실험제목: Second Operand & Multiplication, Floating-Point

실험일자: 2023년 10월 17일 (화)

제출일자: 2023년 10월 28일 (토)

(

학 과: 컴퓨터공학과

담당교수: 이준환 교수님

실습분반: 화 6,7 목 5

학 번: 2022202065

성 명: 박나림

1. 제목 및 목적 (3%)

A. 제목

Second Operand & Multiplication

B. 목적

Multiplication operation과 Second operation의 차이점을 이해하고 코드의 성능 차이를 비교해보도록 한다. Floating point number를 공부하고 이에 대한 adder를 구현한다. 이 과정에서 shift를 사용할 때 종류에 따라 left, right를 다르게 하는 걸 이해할 수 있도록 한다.

2. 설계 (Design) (50%)

A. Pseudo code

Problem 1)

```
        AREA ARMex, CODE, READONLY
         ENTRY
start
        MOV r0, #1 ;start, 1!
        MOV r0, r0, LSL #1 ;2!

        MOV r0, r0, LSL #1 ;2x2
        ADD r0, r0, #2 ;3!

        MOV r0, r0, LSL #2 ;2x3x4 =4!

        MOV r0, r0, LSL #2 ;2x3x4x4
        ADD r0, r0, #24 ;5!

        MOV r0, r0, LSL #2 ;2x3x4x5x4
        ADD r0, r0, #240 ;6!

        MOV r1, r0 ;r1=720
        MOV r0, r0, LSL #2 ;2x3x4x5x6x4
        MOV r2, r1, LSL #1 ;r2=720x2
        ADD r2, r2, r1 ;r2=720x3
        ADD r0, r0, r2 ;r0=7!

        MOV r0, r0, LSL #3 ;8!
        MOV r1, r0 ;r1=8!
```

```
        MOV r0, r0, LSL #3 ;8!x8
        ADD r0, r0, r1 ;9!

        MOV r1, r0   ;r1=9!
        MOV r1, r1, LSL #1 ;9!x2
        MOV r0, r0, LSL #3 ;9!x8
        ADD r0, r0, r1 ;10!

        LDR r8, TEMPADDR1
        STR r0, [r8]

TEMPADDR1 & &40000

        MOV pc, lr
        END
```

Problem 2)

```
        AREA ARMex, CODE, READONLY
         ENTRY
start
        MOV r0, #2 ;2!
        MOV r1, #0 ;sum

        MOV r2, #3
        MUL r1, r0, r2 ;3!

        MOV r0, r1 ;r0=sum
        MOV r2, #4
        MUL r0, r1, r2 ;4!

        MOV r2, #5
        MUL r1, r0, r2 ;r1=sum=5!

        MOV r2, #6
        MUL r0, r1, r2 ;r0=sum=6!

        MOV r2, #7
        MUL r1, r0, r2 ;r1=sum=7!

        MOV r2, #8
        MUL r0, r1, r2 ;r0=sum=8!
```

```
        MOV r2, #9
        MUL r1, r0, r2 ;r1=sum=9!

        MOV r2, #10
        MUL r0, r1, r2 ;r0=sum=10!

        LDR r8, TEMPADDR1
        STR r0, [r8]

TEMPADDR1 & &40000

        MOV pc, lr
        END
```

Problem 3-1)

```
        AREA ARMex, CODE, READONLY
         ENTRY
start
        MOV r0, #0 ;sum(17x3)
        MOV r1, #17
        MOV r2, #3

        MUL r0, r1, r2 ;17x3

        LDR r8, TEMPADDR1
        STR r0, [r8]

TEMPADDR1 & &40000

        MOV pc, lr
        END
```

Problem 3-2)

```
        AREA ARMex, CODE, READONLY
         ENTRY
start
        MOV r0, #0 ;sum(3x17)
        MOV r1, #3
        MOV r2, #17

        MUL r0, r1, r2 ;3x17
```

```
        LDR r8, TEMPADDR1
        STR r0, [r8]

TEMPADDR1 & &40000

        MOV pc, lr
        END
```

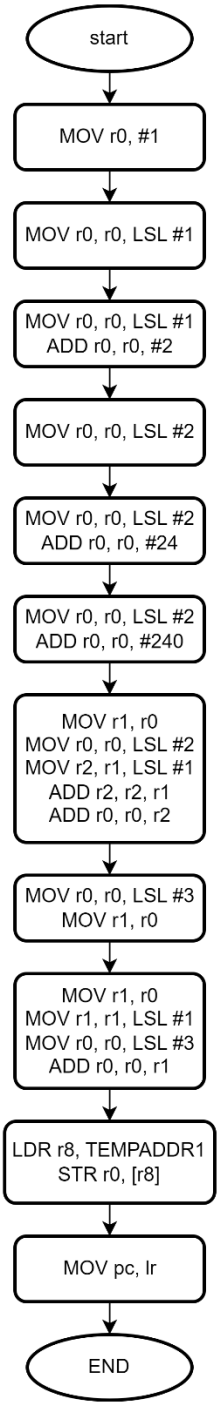
Problem 4-Floating-Point Addition

```
        AREA ARMex, CODE, READONLY
         ENTRY
start
        LDR r1, Val_m1
        LDR r2, Val_m2

        ;Val1 Sign bit
        LDR r3, [r1]
        LSR r3, r3, #31

        ; Val1 Exponent bits
        LDR r4, [r1]
        LSR r4, r4, #23
        AND r4, r4, #0xFF ;8bit

        ; Val1 Fraction Bits
        LDR r5, [r1]
        LSL r5, r5, #9
        LSR r5, r5, #25
        ADD r5, r5, #0x80

        ; Val2 Sign bit
        LDR r6, [r2]
        LSR r6, r6, #31

        ;val_2 Exponent bits
        LDR r7, [r2]
        LSR r7, r7, #23
        AND r7, r7, #0xFF ;8bit

        ; Val2 Fraction Bits
        LDR r8, [r2]
        LSL r8, r8, #9
```

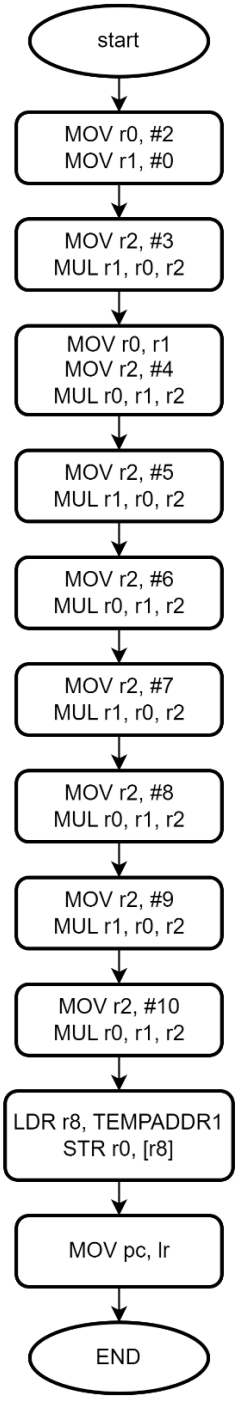
```
        LSR r8, r8, #25
        ADD r8, r8, #0x80

        ;Compare Exponent
        CMP r4, r7
        BLT ManSub1 ;r4 < r7
        BGT ManSub2 ;r7 < r4
        MOVEQ r9, #0 ;r4 = r7

ManSub1
        SUB r9, r7, r4
        LSR r5, r5, r9
        B Normalize

ManSub2
        SUB r9, r4, r7
        LSR r8, r5, r9
        B Normalize

Normalize
        LSL r1, r1, #31 ;Sign
        ADD r9, r9, #127 ;Exponent
        LSL r9, r9, #23
        ADD r10, r5, r8 ;mantissa addition
        LSL r10, r10, #15
        ADD r11, r9, r10

        LDR r12, TEMPADDR1
        STR r11, [r12]
        B EndPro

Val_m1 DCD Val1
Val1 DCI 0x3FC00000

Val_m2 DCD Val2
Val2 DCI 0x40500000

TEMPADDR1 & &40000

EndPro ;End program
        END
```

B. Flow chart 작성

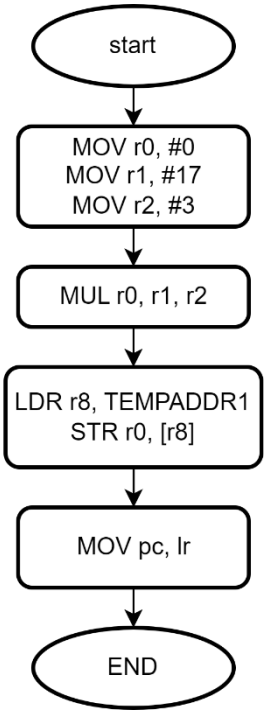
Problem 1)



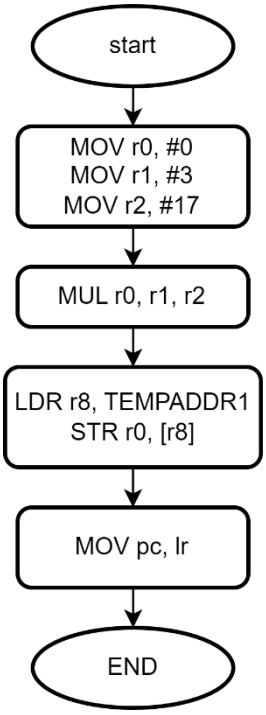
Problem 2)



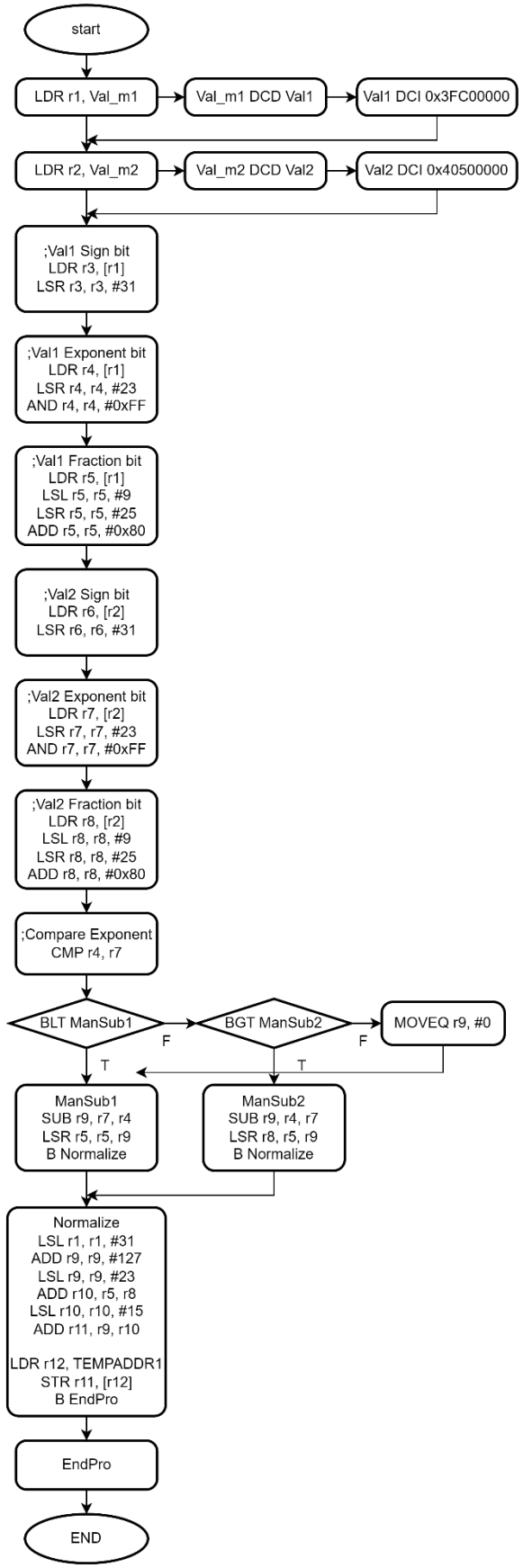
Problem 3-1)



Problem 3-2)



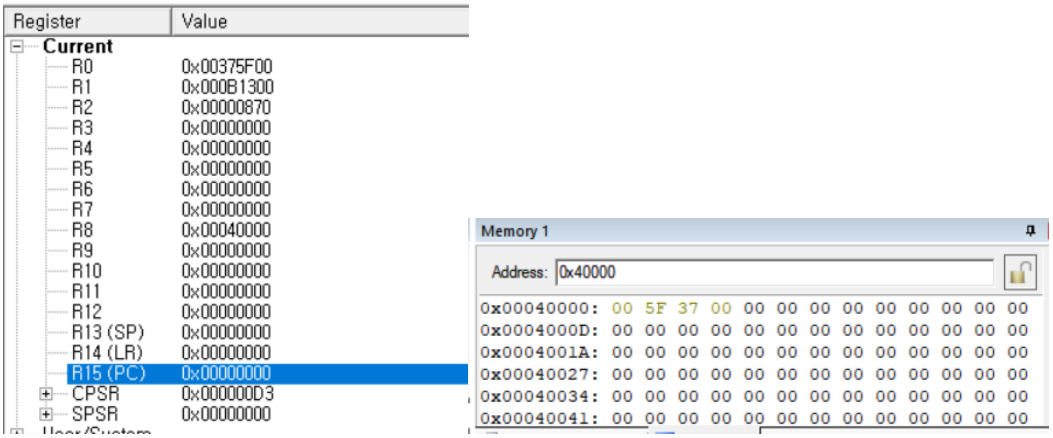
Problem 4)


start
LDR r1, Val_m1
Val_m1 DCD Val1
Val1 DCI 0x3FC00000
LDR r2, Val_m2
Val_m2 DCD Val2
Val2 DCI 0x40500000
;Val1 Sign bit
LDR r3, [r1]
LSR r3, r3, #31
;Val1 Exponent bit
LDR r4, [r1]
LSR r4, r4, #23
AND r4, r4, #0xFF
;Val1 Fraction bit
LDR r5, [r1]
LSL r5, r5, #9
LSR r5, r5, #25
ADD r5, r5, #0x80
;Val2 Sign bit
LDR r6, [r2]
LSR r6, r6, #31
;Val2 Exponent bit
LDR r7, [r2]
LSR r7, r7, #23
AND r7, r7, #0xFF
;Val2 Fraction bit
LDR r8, [r2]
LSL r8, r8, #9
LSR r8, r8, #25
ADD r8, r8, #0x80
;Compare Exponent
CMP r4, r7
BLT ManSub1
F
BGT ManSub2
F
MOVEQ r9, #0
T
T
ManSub1
SUB r9, r7, r4
LSR r5, r5, r9
B Normalize
ManSub2
SUB r9, r4, r7
LSR r8, r5, r9
B Normalize
Normalize
LSL r1, r1, #31
ADD r9, r9, #127
LSL r9, r9, #23
ADD r10, r5, r8
LSL r10, r10, #15
ADD r11, r9, r10
LDR r12, TEMPADDR1
STR r11, [r12]
B EndPro
EndPro
END

C. Result

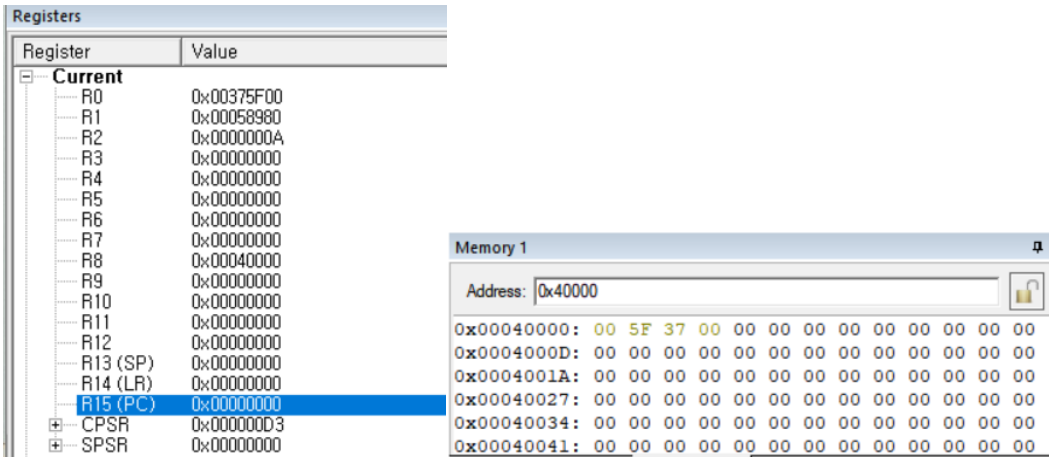
Problem 1)

Second operand만을 사용하여 1!~10!의 값을 계산한 결과이다. R1과 R2 register를 번갈아 사용하여 총 합을 계산한 뒤 나온 결과인 375F00을 R0에 저장하고 이를 메모리에 저장하였다.



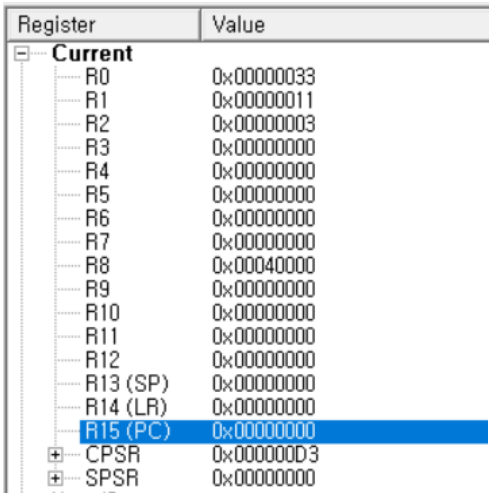
Problem 2)

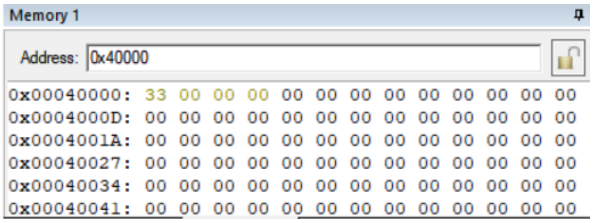
같은 문제를 이번엔 Multiplication operation으로 수정하여 코드를 작성하였다. 마찬가지로 R1과 R2 register를 번갈아 사용하여 총 합을 계산한 뒤 나온 결과인 375F00을 R0에 저장하고 이를 메모리에 저장하였다.



Problem 3-1)

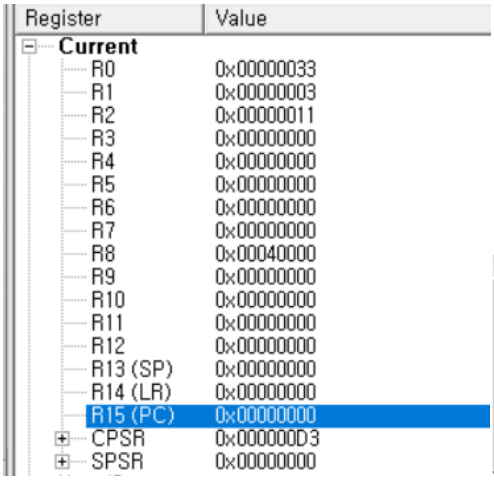
R1에 17, R2에 3을 저장한 뒤 곱한 값을 R0에 넣고 메모리에 저장한 모습이다.

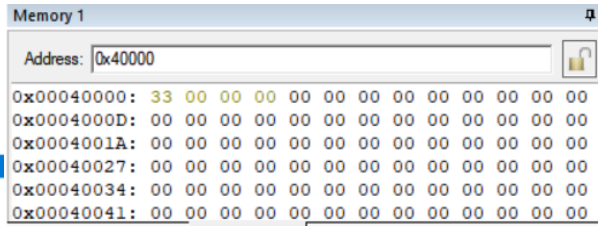




Problem 3-2)

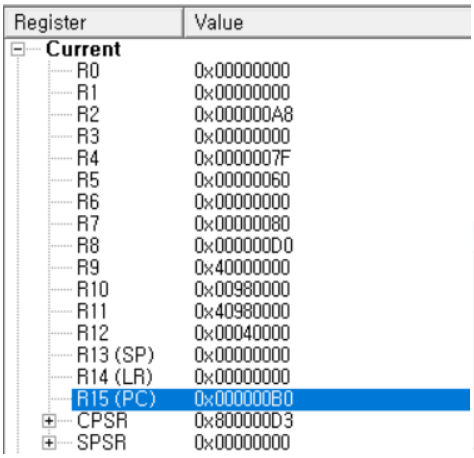
반대 순서로 계산하기 위해 R1에 3, R2에 17을 저장한 뒤 곱한 값을 R0에 넣고 메모리에 저장한 모습이다.

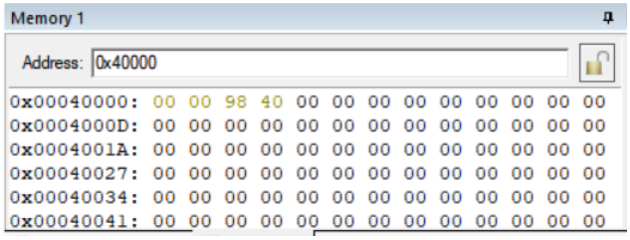




Problem 4-Floating-Point Addition)

임의의 정수 값을 DCI로 메모리에 저장한 뒤, 해당 값을 각각 r1, r2에 load하였다. 이후 r3, r4, r5는 첫번째 값의 sign, exponent, fraction bit을 추출한 값을 저장하였고 두번째 값의 추출 값들은 r6, r7, r8에 저장하여 연산하였다. 지수를 비교하여 그 차이를 r9에 저장한 뒤 r9값 만큼 r5(또는 r8)의 값을 LSR시킨 값으로 나머지 bit들도 전부 Normalize를 한 값이 register에 저장된 모습이다. 최종 덧셈 결과는 r11에 저장되며, 이 값은 메모리에 저장된다.





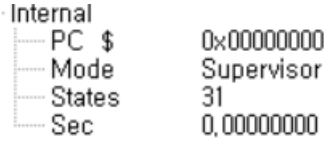
D. Performance

Problem 1)

Code size: 104

```
Program Size: Code=104
".\Objects\test3.axf" -
Build Time Elapsed:  0(
```

State: 31

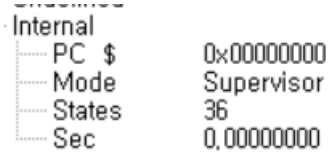


Problem 2)

Code size: 92

```
Program Size: Code=92
".\Objects\test3.axf"
Build Time Elapsed:
```

State: 36

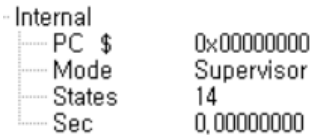


Problem 3-1)

Code size: 32

```
Program Size: Code=32
".\Objects\test3.axf"
Build Time Elapsed:
```

State: 14

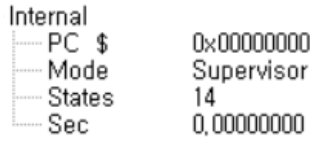


Problem 3-2)

Code size: 32

```
Program Size: Code=32
".\Objects\test3.axf"
Build Time Elapsed:
```

State: 14

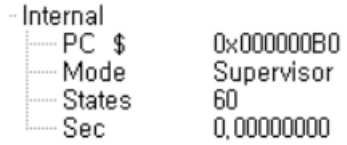


Problem 4-Floating-Point Addition

Code size: 176

```
Program Size: Code=176
".\Objects\test3.axf"
Build Time Elapsed:  0
```

State: 60



3. 고찰 및 결론

A. 고찰 (35%)

Problem3에서는 operand 순서에 따라 성능의 차이를 조사하는 것이었는데, 각각 코드를 작성해도 code size나 state수는 동일하게 나왔다. 이에 Multiplication operation뿐만 아니라 second operand방법도 다 시도해봤으나 전부 동일한 code size와 state수가 나왔다. 이론을 공부해보면 -1같이 전부 1로 채워진 부분에서는 cycle 차이가 발생하였는데, 이렇게 정수로만 곱셈할 때도 차이가 나는지에 대한 것은 더 공부해봐야 될 것 같다.

Problem4에서 Floating point를 만드는 과정 중 Mantissa형태 앞에 1을 붙인다는 것이 처음에는 어떻게 구현해야 할 지 잘 몰랐었다. ARM에서 register에 값을 저장할 때 소수점을 저장할 수는 없으므로, 어떤 식으로 표현해야 될지 고민하다, bit칸을 shift하여 임의적으로 소수점 칸을 정하는 방법을 생각하게 되었다. 그래서 shift한 다음 생긴 공간에 1을 붙여서 연산을 진행하였더니 최종 덧셈 값이 정상적으로 나올 수 있었다.

B. 결론 (10%)

단순히 곱셈을 진행할 때 MUL말고도 Second operand를 통해서도 계산할 수 있다는 점을 공부할 수 있었다. 이러한 형태를 응용하여 나눗셈도 오른쪽으로 shift하는 과정을 통해 계산할 수 있을 것 같다는 생각이 들었다.

Floating point adder를 구현하면서 어떤 부분에서 left, right로 shift하는지 자세히 깨달을 수 있었다. 단순히 값을 추출할 때도 서로 섞어서 shift해야 됐으며 Normalize 과정을 진행할 때에도 지수의 차이에 따라 작은 값을 큰 값에 맞추어서 shift해야 됐었다. 또한 최종적으로 값들을 다 바꿀 때도 처음 추출할 때처럼 섞어가며 shift해야 정상적인 값이 나올 수 있었으므로, 상황에 따라 적절한 연산을 사용해야 된다는 점을 자세히 공부할 수 있었다.

4. 참고문헌 (2%)

이준환 / 어셈블리프로그램설계및실습 강의자료 / 광운대학교(컴퓨터공학과) / 2023