Report – HW 5

1. Problem & Purpose

i. Purpose

예제 코드를 수행하여 기본적인 동작에 대해 이해하고, register 값을 확인하는 법을 숙지하기 위해 해당 과제를 진행한다.

ii. Problem

Problem 1.

- Floating point값 2개를 메모리로 부터 load하여 덧셈 연산을 수행
- 메모리에서 불러오는 값은 실수
  - 양수, 음수 및 0값이 가능
- 결과값은 다음 메모리 주소에 저장
  - 0x40000000

2. Used Instruction

MOV, MOVPL, MOVMI // END // LDR // STR // STRB // LTM // STM // LSL // LSR // ADD // SUB // AND // B // BL

i. MOV Rd, operand2 : operand2에 있는 값을 Rd에 저장한다.

A. MOVPL Rd, operand2 : CPSR에 있는 N flag 가 0일 때 실행한다. Comparison 명령어가 비교한 결과가 0이거나 양수일 때 실행한다.

B. MOVMI Rd, operad2 : CPSR에 있는 N flag 가 1일 때 실행한다. Comparison 명령어가 비교한 결과가 음수일 때 실행한다.

ii. END : Assembly code가 끝났음을 의미하는 Instruction

iii. LDR Rd, addressing : addressing에 있는 값을 Rd에 저장한다.

iv. STR Rd <address> : Rd에 있는 값을 address에 저장한다.

v. STRB Rd <address> : Rd에 있는 값을 바이트 단위로 address에 저장한다.

vi. LDMFD Rn <registers> : Rn에 있는 값을 registers에 저장한다

vii. STMFD Rn <registers> : Rn에 registers를 저장한다.

viii. LSL #number : Number 만큼 왼쪽으로 bit stream을 shift

ix. LSR #number : Number 만큼 오른쪽으로 bit stream을 shift

x. ADD Rd, Rn, N : Rd에 Rn+N 값을 저장한다

xi. SUB Rd, Rn, N : Rd에 Rn-N 값을 저장한다

xii. AND Rd, Rn, N : Rd에 Rn,과 N을 AND 연산한 값을 넣는다

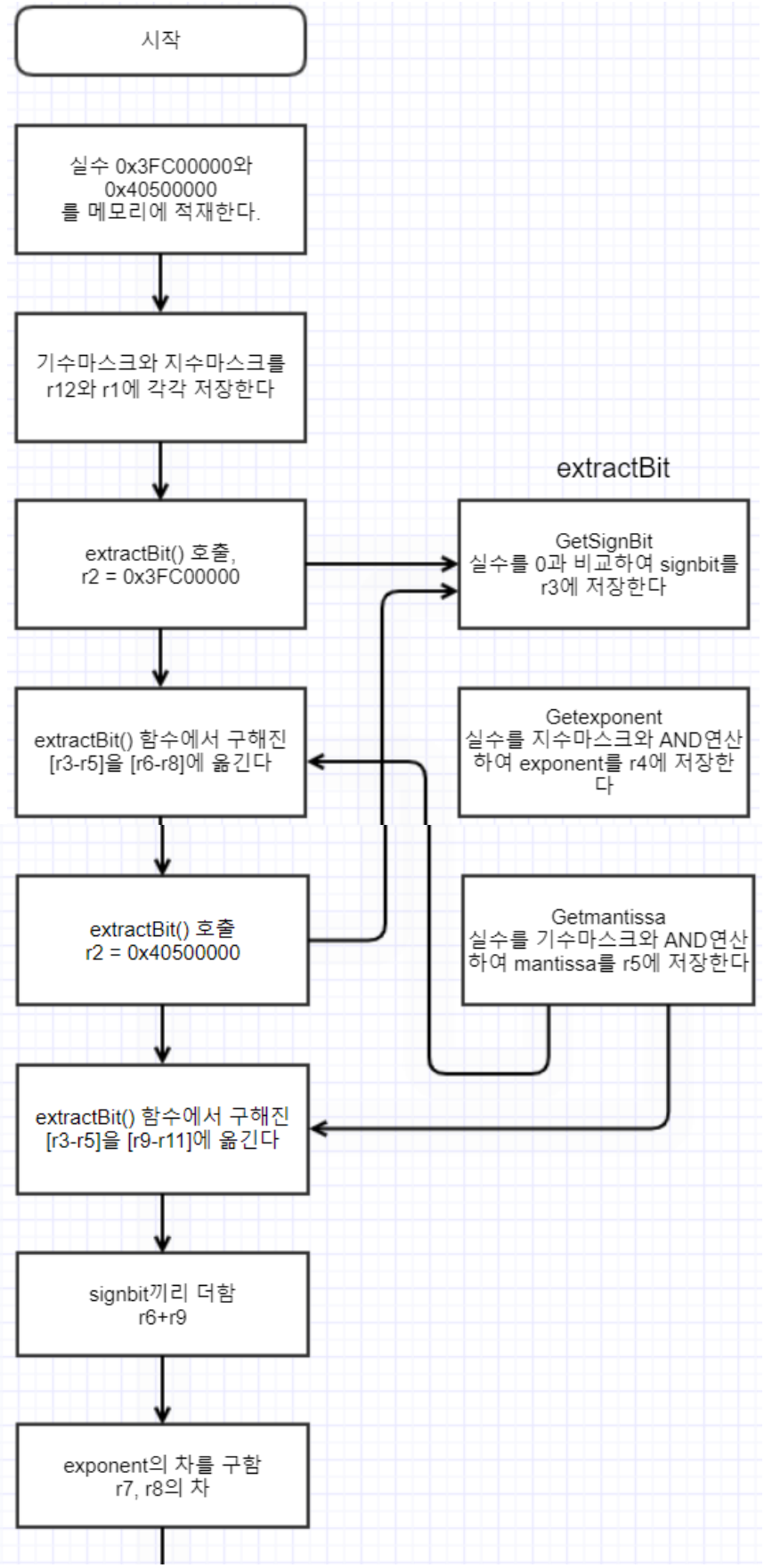
xiii. B label : 작성한 label의 첫 번째 instruction으로 이동

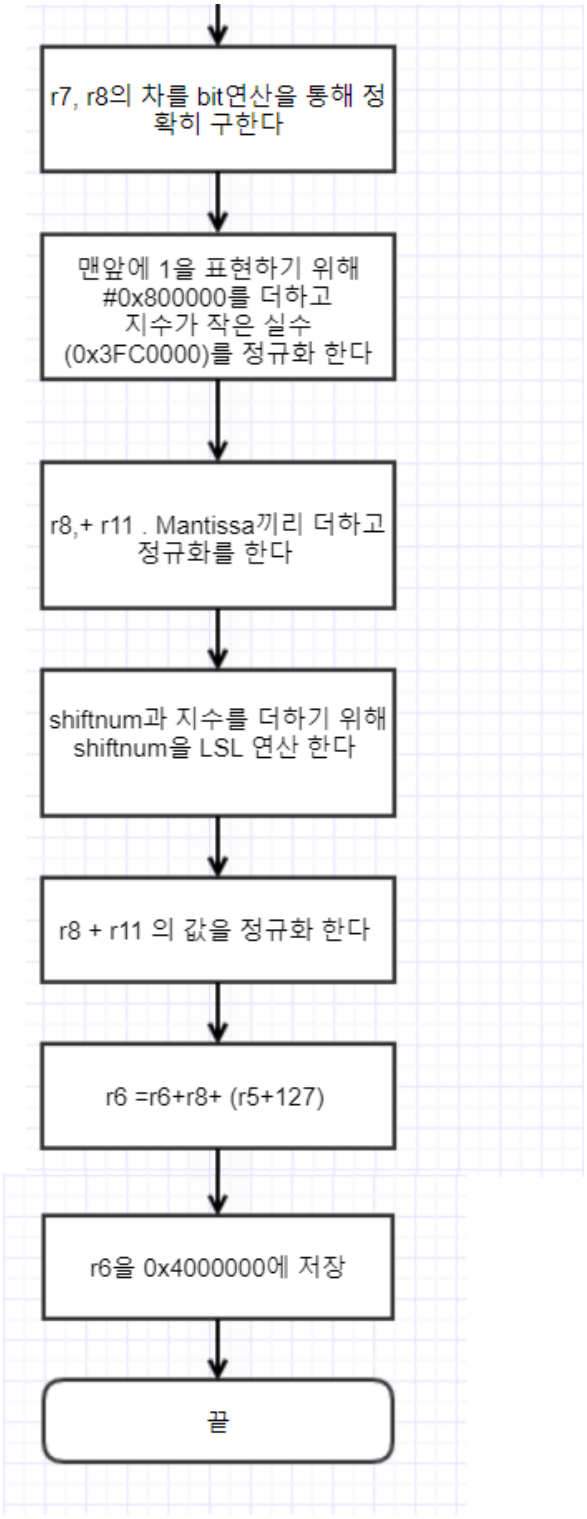
xiv. BL label : LR에 BL 다음에 실행할 명령어의 주소를 저장하고 label의 첫 번째 instruction으로 이동

3. Design(Flow chart)

i. 설계한 내용의 Flow chart

A. Problem 1


시작
실수 0x3FC00000와
0x40500000
를 메모리에 적재한다.
기수마스크와 지수마스크를
r12와 r1에 각각 저장한다
extractBit
extractBit() 호출,
r2 = 0x3FC00000
GetSignBit
실수를 0과 비교하여 signbit를
r3에 저장한다
extractBit() 함수에서 구해진
[r3-r5]을 [r6-r8]에 옮긴다
Getexponent
실수를 지수마스크와 AND연산
하여 exponent를 r4에 저장한
다
extractBit() 호출
r2 = 0x40500000
Getmantissa
실수를 기수마스크와 AND연산
하여 mantissa를 r5에 저장한다
extractBit() 함수에서 구해진
[r3-r5]을 [r9-r11]에 옮긴다
signbit끼리 더함
r6+r9
exponent의 차를 구함
r7, r8의 차



4. Conclusion

대체 숫자에서 어떻게 bit data를 뽑아와야할지 고민을 많이했다. bit단위로도 로드할 수 있는 명령어가 존재하는 것인가? 해서 찾아봤지만 그런 명령어는 없어보였다. 그래서 처음엔 STRB 를 이용해서 Byte 단위로 읽어왔다. 첫 1byte를 뽑아서 sign bit를 해결할 수는 있었다. STRB

로 31bit~24bit 까지 뽑아왔을 때 이 숫자가 2의 7승 보다 크다면 7번째 bit의 자리가 1이란 소리였다. 그렇다면 sign bit가 음수라는 걸 알 수 있었다. 그런데 곰곰히 생각해보니 signbit는 그냥 실수를 0과 비교해서 작으면 1, 크거나 같으면 0을 넣으면 되었다. 괜히 빙빙 꼬아서 생각했다. 나머지 emponent랑 mantissa는 어떻게 해결할까 싶었다. 고민하다보니 LSL을 이용하여 1BIT씩 왼쪽으로 밀면 해결할 수 있을 것 같았다. 그러나 계산을 많이 하고 REGISTER도 많이 사용해야 할 것 같았다.

좀 더 쉬운 방법이 있을 것 같아서 열심히 구글링 한 결과, 기수마스크와 지수마스크라는 개념이 있었다. 마스크를 통해서 기존에 했던 생각했던 비트연산 방식보다 더 쉽게 접근할 수 있을 것 같았다.

Exponent 마스크:

0 11111111 00000000000000000000000

Mantissa 마스크:

0 00000000 11111111111111111111111

이 숫자들을 통해 AND 연산을 이용하려 했다. 처음에 #0b ~~ 형태로 이진수를 input하려 했으나 keli에서는 이진수를 지원하지 않는다고 하여 16진수로 전부 변환했다. 그러나 저 숫자는 ARM에서 roatation으로 지원하지 않는 숫자였다. 수업에서 만약 이런 수를 쓰고싶다면 여러가지 조합을 통해 사용할 수 있다고 배웠다. 두 세가지의 수 조합을 통해 마스크를 만들었다. 조합을 하려고 덧셈연산을 많이 사용하게 되었다. 혹시 BIT연산이 더 코드효율성이 좋을까 라는 생각이 들었다.

실수 두개의 signbit, exponent bit, mantissa bit를 추출해오는 매커니즘은 같기 때문에 브랜치를 이용하여 코드사이즈를 줄일 수 있었다.

Register를 r0~r12, sp, lr, pc 까지 다 쓰는 과제는 처음이었다. Register가 부족하지 않게 재사용을 많이 했어야 했는데, 이럴려면 코드의 흐름을 굉장히 잘 알아야 했다. 어떤 레지스터의 값이 날아가버리면 안되는지 잘 파악해야 했다.

5. Reference

- http://www.hipenpal.com/tool/binary-octal-decimal-hexadecimal-number-converter-in-korean.php#top3

16진수 -> 2진수 변환기

- https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/

  rotate로 숫자를 표현할 수 있는지 확인할 수 있는 사이트

- https://stackoverflow.com/questions/15685181/how-to-get-the-sign-mantissa-and-exponent-of-a-floating-point-number

  비트마스크, 기수마스크 아이디어를 얻어옴

- 강의교안 참고