Report – HW 4

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.

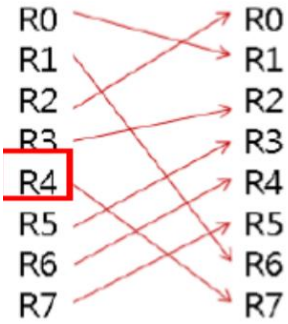
- Factorial은 1부터 n까지의 곱을 수행하는 것으로 '!'으로 표시한다
  - Ex) 4!=4x3x2x1=24
- 다음과 같이 재귀 함수를 이용하여 구현할 수 있음

```
int factorial (int n) {
    if (n <= 1)
        return 1;
    else
        return n * factorial(n-1);
}
```

- 10!의 값(3628800)을 스택과 재귀 함수를 이용하여 구현!
  - 결과는 메모리 4000번지에 저장
  - 반드시 스택과 재귀 함수 개념을 이용할 것
  - 프로그램은 MOV pc, #0으로 끝낼 것

Problem 2.

- 아래 그림에 주어진 대로 r0로 부터 r7까지 레지스터 간의 값을 바꾸는(왼편의 값을 오른편의 레지스터에 저장시키는) 코드를 작성하라.
- 단, 반드시 stack 혹은 블록 복사 명령어를 활용R0=1, R1=2, R2=3, R3=4



Problem 3.

- 다음 동작을 하는 프로그램을 작성하라
- Register r0-r7에는 10~17의 값이 저장되어 있음
- 함수 doRegister() 호출
  - 문제 2와 같이 register들의 값 copy
  - 각 register의 값과 register 의 index(예를 들어, r0의 index는 0, 더한 후, r0-r7의 값들의 합을 return
- 함수 doGCD()
  - 위에서 구해진 r0-r7의 합과 160과의 최대 공약수 구하기
- 구해진 GCD와 register들이 doRegister()를 통해 값이 변경되기 전의 r4를 더 하여 메모리에 저장
- 프로그램 종료

2. Used Instruction

MOV, MOVLS // END // CMP // LDR // STRB // LDMFD // STMFD // ADD // SUB, SUBGT, SUBLT // MUL // BL // BNE // DCD

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

A. MOVLS Rd operand 2 : CPSR에 있는 C clear 거나 Z flag set 되었을 때 실행한다. Comparison 명령어가 비교한 결과가 부호가 없고 0보다 같거나 작을 때 실행한다.

ii. END : Assembly code가 끝났음을 의미하는 Instruction

iii. CMP Rn operand2 : Rn와 operand2에 있는 값을 빼서 그 결과에 따라 CPSR에 있는 N, Z, C, V flag를 설정한다.

iv. LDR Rd addressing : addressing에 있는 값을 Rd에 저장한다.

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

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

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

viii. ADD Rd, Rn, N : Rd에 Rn+N 값을 저장한다

ix. SUB RD, Rn, N : Rd에 Rn-N 값을 저장한다

A. SUBGT : CPSR에 있는 Z flag가 clear, N and V flag가 같을 때 뺄셈을 한다. Comparison 명령어가 비교한 결과가 부호가 있고 0보다 크거나 같을 때 실행한다.

B. SUBLT : CPSR에 있는 N and V flag가 서로 다를 때 뺄셈을 한다. Comparison 명령어가 비교한 결과가 부호가 있고 0보다 작을 때 실행한다.

x. MUL Rd, Rm, Rs : Rm에 저장된 값과 Rs값을 곱하여 Rd에 저장한다

xi. BNE label : CPSR에 있는 Z flag가 clear되었을 때 작성한 label의 첫 번째 instruction으로 이동

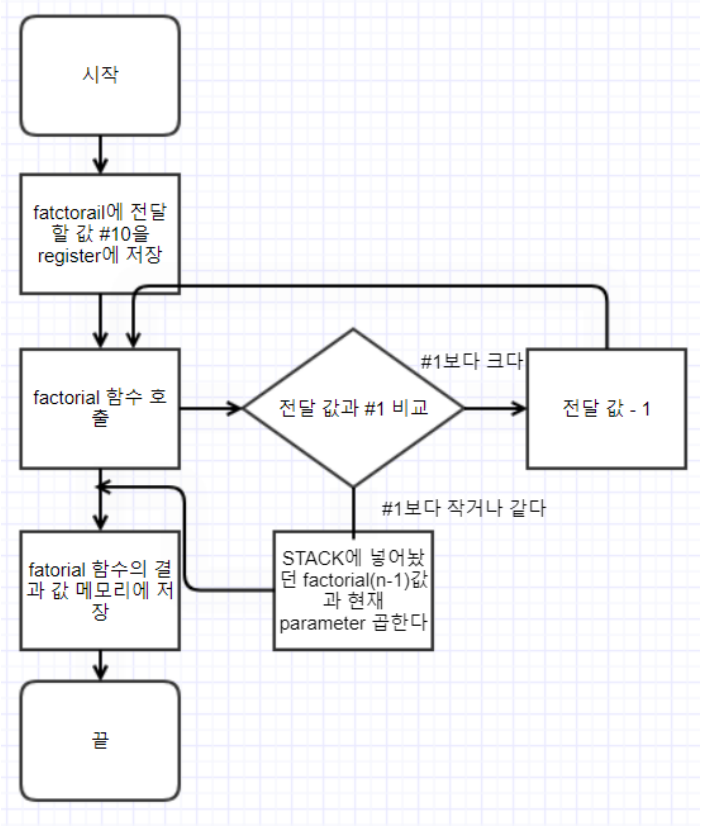
xii. BL label : LR에 BL 다음에 실행할 명령어의 주소를 저장하고 label의 첫 번째 instruction으로 이동

xiii. {label} DCD exer{,expr} or {label} & expr{,expr} : 4 byte 단위로 메모리를 할당 및 해당 값으로 초기화{

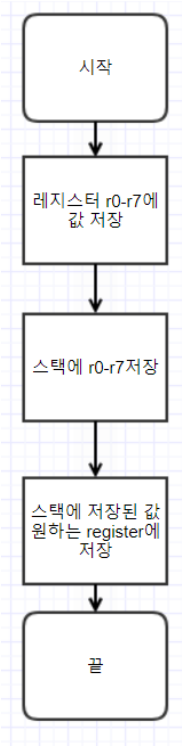
3. Design(Flow chart)

i. 설계한 내용의 Flow chart

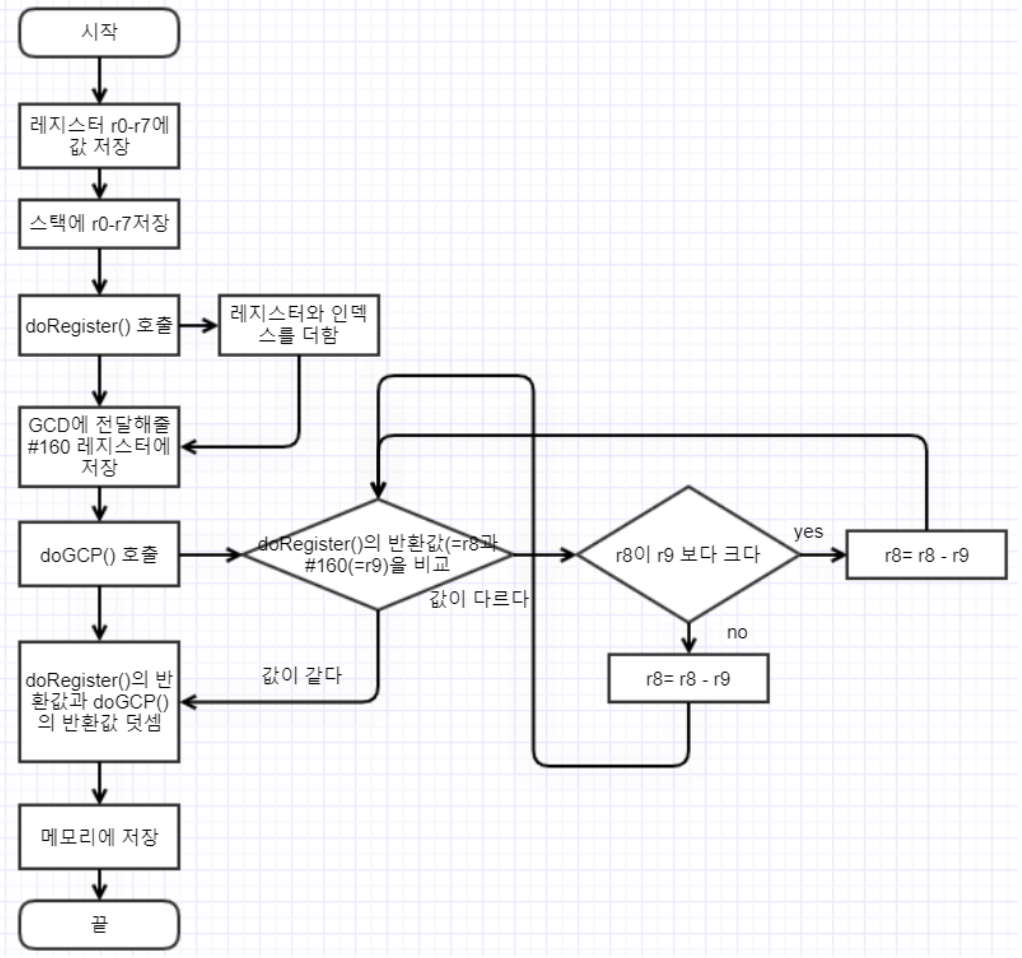
A. Problem 1



B. Problem 2



C. Problem 3



4. Conclusion

과제 3의 1번 문제(팩토리얼 값 구하기) 재귀함수로 풀어봤다. Subroutine call을 그냥 고급언어 (java, C)에서 사용했을 때는 단수히 함수가 함수를 호출하나 보다 하고, 결과값에 의존하여서 코드를 꾸역꾸역 짜냈다. 수업을 들으면서 PC와 LR의 관계를 알고, 현재 프로그램의 context를 저장할 수 있는 STACK의 개념을 배웠고, 그를 통해 어셈블러로 재귀함수를 구현하니 정확히 어떻게 작동하는지 알게 되었다

수업만 들었을 땐 막연했던 링크 레지스터(LR)도 실습을 통해 어떤 개념인지 알았고, 현재 프로그램의 context를 저장하는 것이 중요하다고 생각했다. 서브루틴을 호출 하기 전에, 다시 돌아올 곳을 저장해야 안전하게 프로그램이 돌아갈 수 있다.

(Problem 1)

처음에 짰을 땐, 불필요한 레지스터의 개수가 많았다. 고급언어에서는 프로그램의 유지보수와 협업성을 위해 변수를 일일히 지정하고 재사용하지는 않았다. 그런데 어셈블리어에서는 효율

을 중요시 하다보니, 레지스터에 저장되어 있는 값을 더 이상 사용하지 않는다면 얼마든지 덮어서 쓸 수 있었다. 과제를 해결하면서 이 점을 깨달아서 code size를 줄일 수 있었다.

브랜치를 이용하면 좋을까 CONDITINAL EXECUTION으로 하면 좋을까를 많이 생각했다. 필자는 CONDITINAL EXECUTION으로 작성했다. 브랜치보다는 아무래도 짜는게 쉽기도 했고, 서브루틴 자체도 브랜치인데 브랜치 안에서 브랜치를 호출하는게 이게 서브루틴이 맞는가 싶었기 때문이다. 근데 구글링을 통해 다른 사람들이 어셈블러로 서브루틴을 짜는 것을 보니깐 서브루틴 안에서 브랜치를 호출했다. 하긴 고급 언어에서도 함수안에 반복문 쓰고 조건문 쓰는거 보니 서브루틴 안에서도 분기명령어를 쓰는게 적합하겠다. 조건절이 만족할 때 실행할 문장이 3줄 이하면 CONDITINAL EXECUTION을 쓰라고 했었는데 필자의 코드는 3줄이 넘었다. 그렇다면 브랜치문이 성능에 더 좋을 듯 싶다. Branch를 이용해서 짜봐야겠다.

(Probem 2)

2번은 1번보다는 간단하여서 기분 좋게 해결했다. 낮은 레지스터는 낮은 주소로 간다는 것만 알고, LDM과 STM을 잘 응용할 수 있다면 해결할 수 있는 문제였다.

(Problem 3)

3번에서 변경되기 전의 r4를 구할 때, doRegistser()에서 r4를 인덱스 값과 저장 되어있는 값을 더해서 덮어 쓰기 전에 스택에 옮겨 놓을까 생각했었다. 이렇게 되면 사용하는 레지스터의 개수는 줄어들지만, 메모리에 access 하는 빈도는 높아져서 더 효율이 안 좋아질 것 같았다. 그래서 r4는 MOV연산으로 다른 레지스터(=r10)에 저장해 놓았다. 고급언어에서 전역변수에 저장한 것과 비슷한 개념인 것 같다.

최대 공약수 함수를 구현하는데 애를 썼다. 관련 책을 찾아보던 중 비슷한 개념이 있어서 참고를 했다. 고급언어로 쓰여진 최대공약수 함수를 어셈블러로 바꾸려고 시도했다. 고급언어에서는 반복문, IF문, % 연산을 썼는데 어셈블러에서 %연산을 어떻게 구현하나 싶었다. 반복문이나 IF문은 가능하지만... 그래서 %연산을 안 쓰는 다른 방법을 통해서 구현했다. 조건절은 3문장 이하를 만족시켜서 conditional execution이 적합해 보였다.

처음에 BL instruction을 사용할 때 LR과 PC의 관계가 너무 헷갈려서 강의도 다시 들어보고 교안의 exercise도 다시 다 해보고, 구글링도 해보고 관련 책도 많이 찾아봤다. Exercise에 BL을 쓰는 예시가 더 있었으면 좋겠다. 이번 과제를 통해 함수의 호출과 피호출 명령어 사이의 관계를 알게 되었다. subroutine에는 꼭 return문이 있다는 걸 잊으면 안된다. return에선 나를 호출했던 명령어의 다음 주소로 꼭 돌아 가야한다! 이걸 잊지 않고 LR에는 pc의 다음주소가 저장된다는 것을 알면 시간은 좀 걸리더라고 서브루틴을 구현할 수 있다. 근데 LR=PC-4라고 강의에 있었는데 직접 해보니 PC+4가 들어갔었던 것 같다. 강의를 다시 한번 들어야겠다..

5. Reference

강의 실습 교안

강의와 pdf 교안

ARM 920T를 이용한 ARM 시스템 프로그래밍 – 조상영

http://www.cburch.com/books/armsub/