|  |
| --- |
| 제출일: 2021-04-07; Lab 2  이름: 김민재(2016124036), 김현용(2016124087), 차원범(2016124217); 7조 |

# Assignment 1

## 개요

hamming distance를 계산하는 함수를 작성한다. hamming distance란 비교대상 2개의 숫자를 2진수로 표현한 후, 각 비트의 값을 비교하여 다르다면 1을 증가시킨 값을 누적해 구할 수 있다. 해당 함수를 ARM assembly를 이용해 구현하고, main함수에서 해당 함수를 호출하여 exit code로 반환한다. 반환된 exit code를 확인하여 정상적으로 작동되는지 확인하다.

## 진행 내용

우선 hamming distance를 구현하는 함수를 만든다. vi편집기로 만들 수 있으며 'vi hamming\_distance.s' 명령을 사용하였다. 해당 함수에 들어간 코드를 요약해 정리하면 다음과 같다.

|  |
| --- |
| .global hamming  hamming:  eor r2, r1, r0 ; 입력 인자로 들어올 r1과 r0를 xor하여 r2에 입력한다.  mov r3, #0  LOOP:  cmp r2, #0 ; r2가 0이라면 바로 END문으로 이동한다.  beq END  tst r2, #1  addne r3, r3, #1 ; r2&1을 실시하여 0이 아니라면(즉 1이라면) r3의 값을 1 증가시킨다  mov r2, r2, LSR #1 ; shift right 1 하여 LOOP의 처음으로 돌아간다.  b LOOP  END:  mov r0, r3 ; return register인 r0에 r3값을 보낸다.  bx lr |

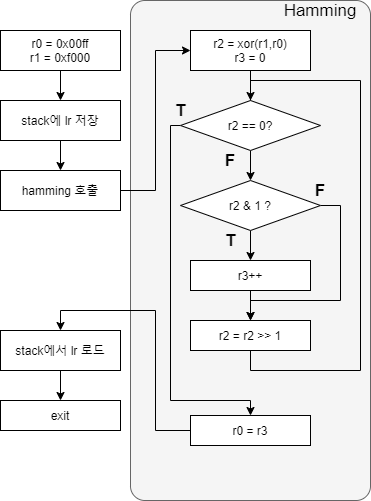
hamming\_distance 함수의 동작 흐름은 입력 인자 두 값을 xor연산한 후, r2에 저장한다. 이로써 각 bit별로 다르다면 1, 같다면 0이 반환된다. 해당 값의 LSB를 보고, 1이라면 r3에 1씩 증가시킨다. 값이 0이 될 때까지 shift right시키며 LSB를 보고 1을 증가시키는 것을 반복하며 0이 된다면 return register인 r0에 값을 넣고 함수가 종료된다.

해당 함수를 구동하는 main함수는 'vi 12.s' 명령을 사용하여 12.s 파일을 생성해 다음과 같이 코드를 작성하였다.

|  |
| --- |
| .global main  main:  mov r0, #0x00ff  mov r1, #0xf000  str lr, [sp, #-4]! ; hamming으로 진입하기 전에 lr 레지스터의 값을 stack으로 미리 저장시켜 놓는다.  bl hamming  ldr lr, [sp], #+4 ; lr 레지스터의 값을 복구한다  bx lr |

main에서는 r0와 r1에 0x00ff와 0xf000을 입력 인자로 주어 hamming함수를 호출하였다. 호출할 때 lr 레지스터의 값이 바뀌므로 원래 lr 레지스터의 값을 보존하기 위해 함수 호출 전에 stack에 저장하고, 호출 후에 로드하도록 하였다. r0로 return된 값은 main에서도 그대로 r0로 표현되며, 해당 r0값이 main 함수의 return값으로 바로 반환되어 함수가 종료된다.

위의 함수를 순서도로 표현한다면 다음과 같이 표현할 수 있다.



이어서 만든 함수를 Assemble하는 과정은 다음과 같다.

as -o 12.o 12.s

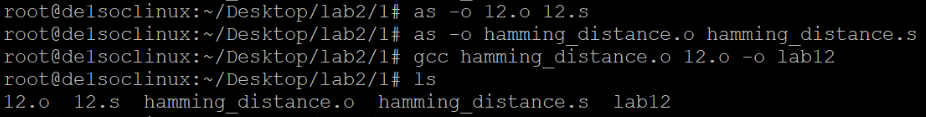
as -o hamming\_distance.o hamming\_distance.s

위와 같이 Assemble하여 목적파일을 만들고, 다시 gcc를 사용해 linking과정까지 하여 최종적인 실행파일까지 만들 수 있다.

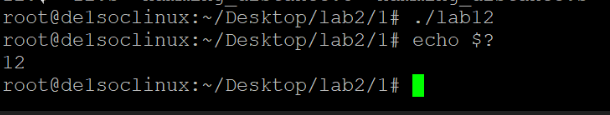
gcc hamming\_distance.o 12.o -o lab12

이렇게 lab12 라는 실행파일을 만들었으며, 해당 실행파일은 './lab12'를 사용해 실행할 수 있다. 실행 파일이 종료될 때 exit code로 r0 레지스터의 값을 return하므로 'echo $?' 를 사용해 어떤 값이 return되었는지 확인할 수 있다.

## 진행 결과



vi 에디터로 만든 12.s와 hamming\_distance.s의 목적파일을 만들고, 다시 gcc를 사용해 linking까지 하는 과정이다. ls 명령을 사용해 성공적으로 12.o와 hamming\_distance.o가 생성되었으며, 이들을 다시 linking하여 생성된 실행파일인 lab12까지 생성된 것을 확인하였다.



최종적으로 해당 실행파일을 실행해 본 결과, 별도의 printf 등이 사용되지 않았으므로 출력 없이 함수가 종료되었다. 설정한 return값을 확인해보기 위해 echo $? 명령을 사용해본 결과 값 12가 나왔다. 값 12는 앞서 입력으로 준 0x00ff와 0xf000의 hamming distance인데, 해당 값은 2진수 표현으로 0x0000 0000 1111 1111와 0x1111 0000 0000 0000이다. 값이 다른 bit의 수가 12개이므로 정상적으로 출력되었음을 확인할 수 있었다.

## 결과 분석 및 팀원 간 토의 사항

어셈블리어로 함수를 호출하여 시행하는 법 뿐만 아니라 return값까지 모두 설정하여 확인해 보는 과정을 익혔다. r0, r1, r2, r3레지스터를 입력 인자로 넘겨주며, scratch register, variable register등의 사용법 등을 알게 되었다. 또한 main 함수의 return 값은 항상 0으로 설정하였기에 0이 아닌 다른 값으로 반환하여 확인해보는 방법을 익힐 수 있었는데 해당 값을 확인하는 것까지 해봄으로써 main 함수의 return값을 사용해 볼 수 있는 사례가 되었다.

# Assignment 2

## 개요

ARM 어셈블리언어를 사용하여 1부터 1000까지의 정수 합계를 계산하는 함수를 구현한다. 그리고 C언어를 사용하여 동일한 기능을 구현한 다음에, C언어로 호출 프로그램을 작성하여 해당 기능들을 호출하고 결과를 비교 분석한다. ls -al을 이용해 프로그램의 용량을 비교하고 각 프로그램에 대해서 효율성에 대해 논의한다.

## 진행 내용

arm 어셈블리 언어로 1부터 n까지의 정수 합계를 계산하는 함수를 작성하였다. 해당 함수는 vi 편집기를 사용해 만들었으며, 'vi sum.s' 명령을 통해 생성하였다. 구현된 sum.s는 loop를 이용해 입력된 정수를 1씩 빼면서 계속 더하는 방법이다.

|  |
| --- |
| .global sum  sum:  mov r1, r0 ;입력값 r1으로 이동  mov r0, #0 ; r0을 0으로 초기화  loop:  add r0, r0, r1 ;입력된 수부터 1씩빼면서 덧셈  sub r1, r1, #1  cmp r1, #0 ;r1이 아직 0이아니라면 loop  bne loop  bx lr |

주석과 같이 입력 인자로 받는 레지스터 r0을 사용해 해당 값을 1씩 빼면서 값을 누적한다. 0이 될 때까지 loop를 순회하며, 0이 된다면 loop를 탈출한다. 최종적으로 r0를 사용해 값을 반환한다.

다음으로 sum.c를 sum.s와 동일한 동작을 하도록 c언어로 구현을 하였다. 위와 같이 vi에디터를 사용해 작성할 수 있다.

|  |
| --- |
| #include<stdio.h>  int sum(int n){ ;입력값 n을 받음  int result=0, i;  for(i=1; i<=n; i++){ ;n번 for문 반복  result += i; ;1부터 n까지의 덧셈  }  return result; ;결과값 반환  } |

for문을 이용해 1부터 n까지의 정수 합계를 출력하는 방법이다. ARM assembly에서는 n부터 1까지 누적하지만, c언어에서는 1부터 n까지 누적을 하는 것으로 바꿔 진행하였고, 결과값은 같다.

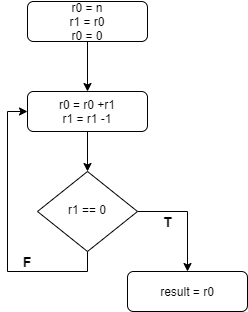
이제 sum.s, sum.c 함수를 수행시킬 main 함수를 C언어로 작성해보면 다음과 같다. 'vi main.c' 를 사용해 생성하였다.

|  |
| --- |
| #include<stdio.h>  int sum(int);  int main(void){  printf("1부터 %d까지의 합은? = %d\n", 1000, sum(1000));  } |

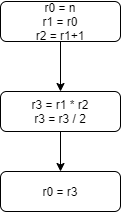
입력 값으로 1000을 입력해 주었고 printf를 통해 sum함수의 결과를 프린트하도록 구현하였다. 그리고 추가적인 방법으로 1부터 n까지의 합은 n\*(n+1)/2 의 식으로 변형 시킬 수 있게 되는데 위 알고리즘을 활용하여 sum2.s로 어셈블리 code작성을 시도하였다. 입력 받은 수와 그 수에 1을 더한 값을 곱해주고 arithmetic shift right로 1칸 이동하여 나누기2까지 연산하는 형태로 구현하였다. 좀더 간결한 알고리즘으로 프로그램의 용량을 줄일 수 있는지 확인해보기 위해서 진행하였다.

|  |
| --- |
| .global sum  sum:  mov r1, r0 ;입력값 r1으로 이동  add r2, r1, #1 ;r2 = n+1  mul r3, r2, r1 ; n\*(n+1)  asr r3, r3, #1 ; n\*(n+1)/2  mov r0, r3 ;r0에 결과 반환  bx lr |

두 방법에 대해서 코드의 전체 흐름을 순서도로 나타내면 아래와 같다.



loop를 활용한 방법의 순서도



간결한 알고리즘을 사용한 함수의 순서도

이어서 구현한 Assembly 코드의 경우 Assemble를 진행하여 목적 파일을 다음과 같은 방법으로 만들 수 있다.

as -o sum.o sum.s

하지만 C언어로 작성한 파일의 경우 arm ISA에 맞도록 컴파일 해야 하므로, 맨 뒤에 marm옵션을 붙여 컴파일해야 한다.

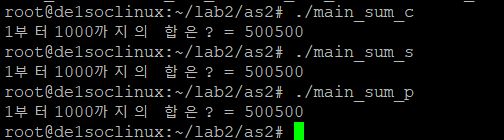
gcc -c main.c -o main.o -marm

위와 같이 컴파일하여 DE1-SOC보드의 ARM 시스템에서 동작할 수 있도록 컴파일한다. 마지막으로 생성된 목적파일간 linking은 다음과 같이 진행한다.

gcc sum.o main.o -o main\_sum\_c

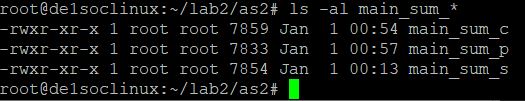
위와 같이 main파일과 필요로 하는 목적 파일들간의 linking 과정을 통해 최종적인 실행파일을 생성할 수 있다. 생성된 실행파일은 ./main\_sum\_c 와 같이 실행하여 그 결과를 확인하였다.

## 진행 결과



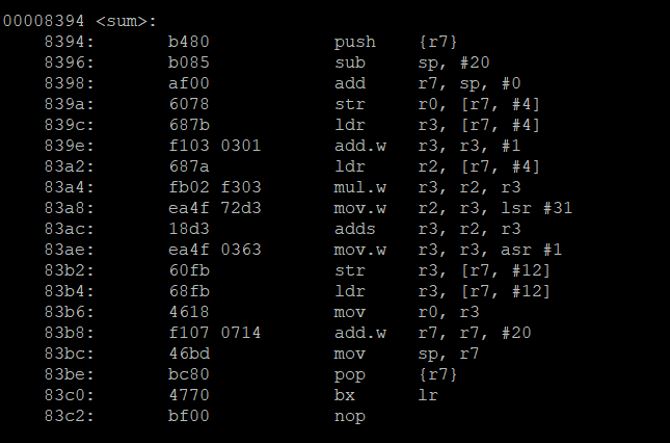
실행결과 화면으로, 위에서부터 순서대로 c언어로 작성하여 컴파일하였을 때, 어셈블리언어로 작성하여 컴파일하였을 때, 간단한 알고리즘을 활용한 어셈블리 언어 함수를 컴파일 하였을 때의 결과이다. 모두 main c함수를 통해 1000의 입력 값을 받았고 모든 결과가 500500으로 똑같이 나왔음을 확인할 수 있었다.

또한 ls명령어에 -al옵션을 주어 파일의 권한, 용량, 생성시간 등을 확인할 수 있는데, 파일의 용량 비교를 위하여 해당 명령을 사용했다.

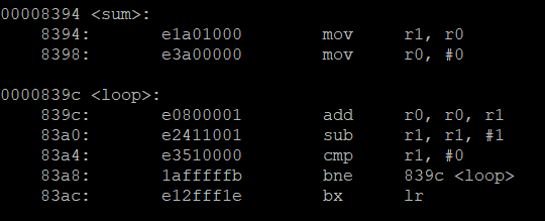


먼저 반복문을 활용한 c언어와 어셈블리 언어 함수를 비교해 보면 5바이트 정도로 차이를 보이고 있다. 결과적으로 어셈블리 언어로 작성하였을 시 같은 동작에서 용량을 단축시킬 수 있는 효과를 볼 수 있었다. 그리고 좀 더 용량 단축을 할 수 있을 것 같다는 생각으로 추가적으로 작성한 간단한 알고리즘의 어셈블리 언어 함수는 같은 어셈블리언어 함수와 비교해 21바이트나 단축되었다. 그래서 같은 결과를 보인다고 한다면 간단한 알고리즘을 찾아 프로그램의 용량을 더 줄일 가능성도 있는 것을 확인할 수 있었다.

추가로 용량 뿐만 아니라 내부 코드의 어떠한 부분에서 차이가 있는지 확인하기 위해 Binutils의 objdump를 사용해 disassemble을 시행할 수 있다. 'objdump -d 실행파일or목적파일' 와 같이 명령을 주어 내부 코드를 기계어 단위로 확인할 수 있다. 다음은 objdump로 살펴본 결과화면이다.



c언어로 작성된 sum함수의 목적파일의 내용이 disassemble된 결과이다.



어셈블리언어로 작성된 sum함수의 목적파일의 내용이 disassemble된 결과이다. C로 함수를 작성하였을 때, 컴파일러가 번역한 어셈블리언어는 어셈블리언어로 직접 코딩했을 때보다 확실히 길어지게 되는데, 입력된 값이 커질수록 거치는 코드의 길이도 길어 지기 때문에 실행시간도 더욱 길어지는 부분도 발생할 것이라고 볼 수 있다.

## 결과 분석 및 팀원 간 토의 사항

1부터 n까지의 합을 계산해주는 함수를 만들기 위해서 먼저 c언어로 for문을 이용하여 함수를 작성해 보고 이후에 어셈블리 언어로 구현하면서 둘의 언어와 동작을 비교해가며 함수를 구현하였다. 그 과정에서 arm 어셈블리 언어의 cmp와 branch 언어들에 대해 이해해가면서 활용해 볼 수 있었다. C의 for문 같은 경우는 어셈블리에서 cmp와 branch로 계속 procedure call을 하면서 loop를 돌도록 만들어 줄 수가 있었다. 그리고 c언어와 어셈블리언어로 작성된 둘 함수의 결과를 비교해보고 같은 결과&동작에서 프로그램의 용량이 어떻게 차이가 나는지 확인하였다. 2번 문제와 같은 간단한 함수는 어셈블리언어로 작성했을 때, 용량이 더 작은 실행파일이 생성되어 효율성이 더 좋은 모습을 보였다. 추가적인 실험으로 간단한 알고리즘으로 구현하였을 때, 용량을 더욱 단축할 수 있는 가능성도 확인할 수 있었다.

# Assignment 3

## 개요

Assignment 2에서 수행했던 과정을 Sigma가 아니라 Fibonacci 수열을 재귀함수 형식으로 하여 똑같이 반복해본다. 여기서 Fibonacci 수열이란, n=0, n=1 일 때는 1을 return하고 나머지 n에 대해서는 Fib(n) = Fib(n-1)+Fib(n-2)을 return하는 수열이다. 이 Fibonacci 수열에 대해서 n=15로 하고 C 언어와 ARM assembly로 작성해본 뒤 각각 컴파일하여 실행 파일의 용량을 비교해 효율성을 따져본다.

## 진행 내용

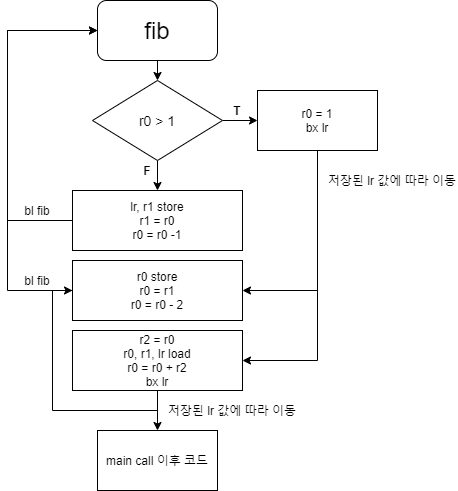
먼저, Fibonacci 함수를 수행시킬 main 함수를 C언어로 작성해보면 다음과 같다. 이 역시 모두 vi 에디터를 사용하였다.

|  |
| --- |
| #include <stdio.h>  extern int fib(int);  int main(void){  printf("fibonacci (%d)=%d\n", 15, fib(15));  } |

main 함수에서는 단순히 프린트문과 함께 fib함수를 호출해준다.  
이제 fib 함수를 어셈블리로 작성해보면 다음과 같다.

|  |
| --- |
| .global fib  fib:  cmp r0, #1 ;r0-1 수행 후 update condition flags  bgt .L2 ;r0>1이면 .L2로 branch  mov r0, #1 ;r0<=1이면 r0 = 1  bx lr  .L2:  str lr, [sp, #-4]!  str r1, [sp, #-4]! ;r1 store (n값 저장용도)  mov r1, r0 ;r0 = r1  sub r0, r0, #1 ;r0--  bl fib  str r0, [sp, #-4]! ;r0 store (f(n-1)결과 저장)  mov r0, r1  sub r0, r0, #2  bl fib  mov r2, r0 ;r2 = r0 = f(n-2)의 결과  ldr r0, [sp], #+4 ;저장해 둔 f(n-1) load  ldr r1, [sp], #+4 ;n++  ldr lr, [sp], #+4  add r0, r0, r2 ;r0 = f(n-1) + f(n-2)  bx lr |

fib는 먼저 매개변수로 받은 r0와 1을 비교해 condition flags를 update한다. 그리고, gt는 NZCV Flags에서 Z=0이고, N=V이면 명령어를 수행한다. 따라서 bgt .L2는 r0와1 비교 결과 r0>1이면 .L2로 branch 명령어를 수행한다. 코드의 전체 흐름을 순서도로 나타내면 아래와 같다.



이제 fib 함수를 C언어로 작성해 컴파일한 뒤 실행 파일의 용량들을 비교해 본다. 이 때 fib 함수를 C언어로 작성한 것은 아래와 같다.

|  |
| --- |
| #include <stdio.h>  int fib(int n){  if(n==0) return 1;  if(n==1) return 1;  return fib(n-1) + fib(n-2);  } |

이 코드는 n이 0,1일 때는 1을 return 하고, 나머지 n의 경우에는 fib(n-1)+fib(n-2)를 return 하도록 작성했다.  
컴파일 과정은 아래와 같이 목적 파일을 만든 뒤 두 목적 파일을 링크하여 실행파일을 만들었다.  
gcc -c main.c -o main.o -marm #main.o 생성

as -o fib\_s.o fib.s #fib\_s.o 생성

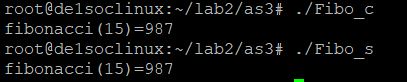
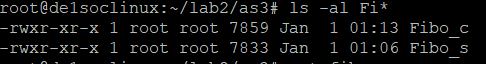
gcc main.o fib\_s.o -o Fibo\_s #fib\_s.o와 main.o 링크하여 실행파일 Fibo\_s 생성

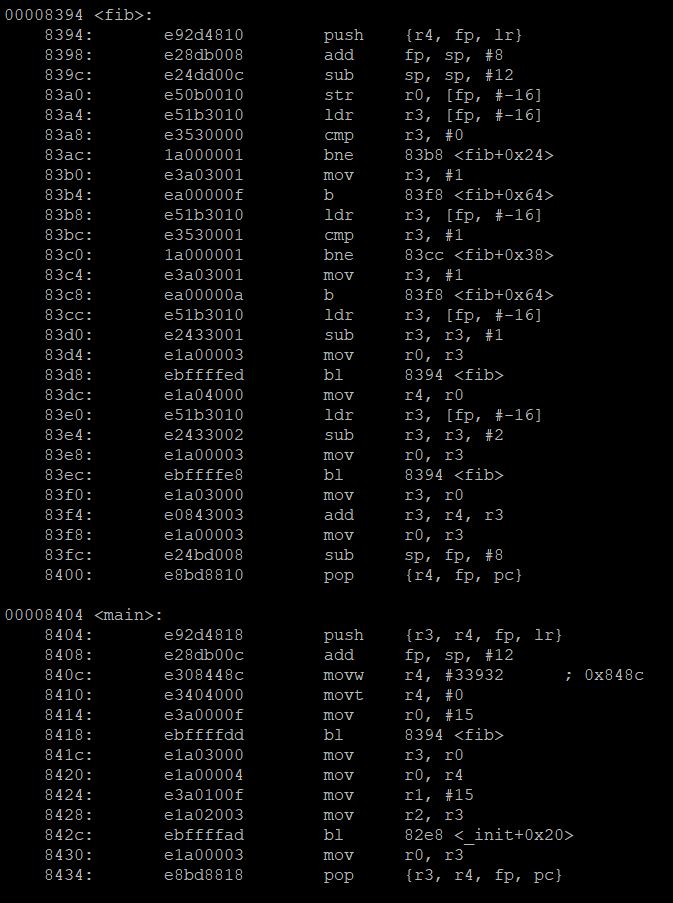
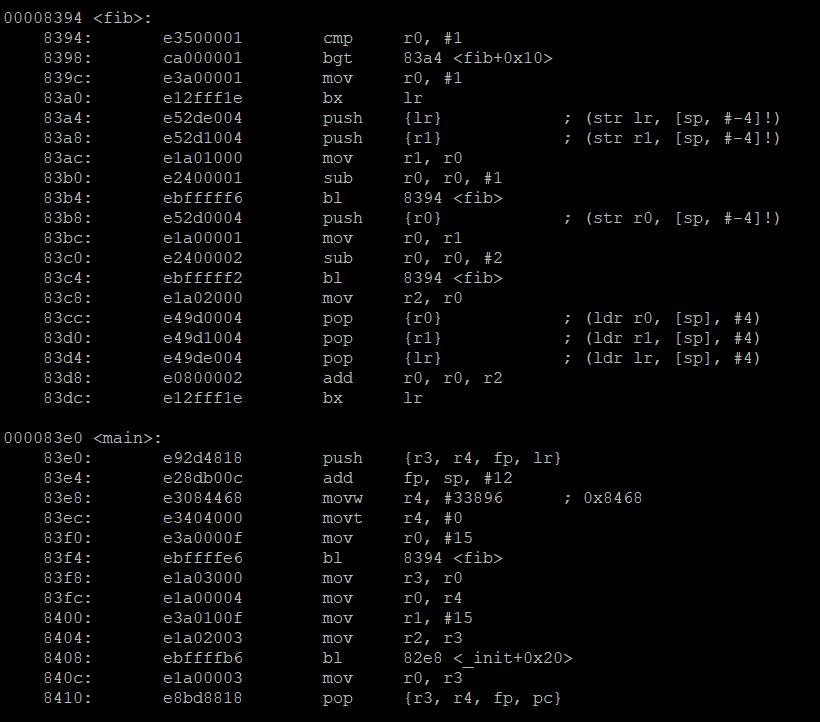
gcc -c fib.c -o fib\_c.o -marm #fib\_c.o 생성

gcc main.o fib\_c.o -o Fibo\_c #fib\_c.o와 main.o 링크하여 실행파일 Fibo\_c 생성

## 진행 결과

먼저, 위에서 생성한 두가지 실행파일 Fibo\_s와 Fibo\_c를 실행한 결과는 다음과 같다.

  
Fib(15)의 값이 둘 다 동일하게 잘 출력된 것을 확인할 수 있었다. 추가적으로 ls 명령에 -al 옵션을 주어 파일의 용량 비교를 진행하였다.  
  
두 실행 파일의 용량을 확인해 본 결과 어셈블리로 작성한 목적 파일로 링크한 것이 대략 26바이트 정도 적은 것을 확인할 수 있었다. 이에 대해서 더 자세하게 알아보기 위해서 두 실행파일에 대해서 objdump -d 명령을 실행해보았더니 다음과 같았다.  
아래의 두 사진 중 위의 사진이 어셈블리어로 만든 실행 파일이며, 아래 사진이 C언어로 만든 실행 파일이다.

  
어셈블리어로 작성한 경우는 작성한 어셈블리를 거의 그대로 실행하는 것을 알 수 있었지만, C언어로 작성한 경우는 컴파일러가 번역해서 생성한 어셈블리어로 실행되는 것을 알 수 있다. 하지만, 코드 길이가 9줄 정도 차이가 나는데, 재귀 함수로 축적되면 n값이 커지면 커질수록 실행 시간도 느려질 것이라는 것을 알 수 있다. 하지만, 컴파일러보다 프로그래머가 짠 어셈블리어가 코드 길이가 더 길어지면 오히려 반대의 상황이 나올 수도 있다는 것을 알 수 있다. 따라서 효율성을 따져보면, 컴파일러의 성능과 프로그래머의 역량에 따라 효율성이 달라지겠지만, 대체적으로 어셈블리어로 작성하는 것이 더 효율적인 것을 알 수 있다.

## 결과 분석 및 팀원 간 토의 사항

먼저 Fibonacci 수열을 재귀 함수 형태로 코딩 하는 부분에 있어서 ARM assembly 작성 능력을 더욱 향상시킬 수 있었다. 특히, ARM에서의 Store 및 Load에 대해 이해도가 높아졌고, NZCV와 같은 Conditional code flags에 대해 깊이 알 수 있게 되었다. 또한, objdump -d 의 리눅스 명령어를 사용해 봄으로써, 목적 파일의 내용을 disassemble된 view로 볼 수 있었고, 이를 기반으로 용량이 적게 나온 원인을 분석해 볼 수 있었다.

# 조원 별 기여 사항 및 느낀점

|  |  |  |  |
| --- | --- | --- | --- |
| 이름 | 기여도 (0 - 100%) | 기여 사항 | 느낀점 |
| 김민재 | 33.3% | - Assignment 1번의 Return Value 및 echo 통한 return 값 검토  - Assignment 2번의 ARM Assembly 작성 및 검토  - Assignment 3번에 대한 피보나치 수열 알고리즘 탐구  - 보고서 작성 | arm 어셈블리 언어의 여러 명령어들을 익히고 공부하면서 활용해보는 시간이었다. c언어와 동작을 비교해보면서 같은 동작에서도 더 좋은 효율성을 비교해보는 과정에서 어셈블리 언어에 대한 공부의 필요성을 느낄 수 있었다. 어셈블리 명령어의 형태들을 더 파악하고 효율적인 사용을 위해 더욱 노력이 필요하다고 생각한다. |
| 김현용 | 33.3% | - Assignment 1번 알고리즘 개선 및 문제 의도 파악  - Assignment 2번의 개선된 알고리즘 제안 및 작성, 검토  - Assignment 3번의 재귀함수 작동 원리 및 Stack push / pop 적용  - 보고서 작성 | 이번 LAB을 진행하면서 Assembly로 짠 코드가 더 효율적이라는 것을 알 수 있는 계기가 되었고, Assembly 코드 또한 얼마나 효율적으로 짜는지에 따라 실행파일의 용량과 시행시간에서 차이가 있음을 알게 되어 더욱 뛰어난 프로그래머가 되어야겠다고 생각했다. |
| 차원범 | 33.4% | - Assignment 1번의 Return Register의 보존 필요성 제안 및 코드 작성  - Assignment 2번의 용량 및 Disassemble 시행 및 분석 비교  - Assignment 3번 버그 수정 및 분석 비교 시행  - 보고서 작성 | Stack을 사용해 재귀함수에서 레지스터를 어떻게 보존하고 어떻게 다시 사용하는지 등에 대해 배울 수 있었다. 또한 용량 또는 시행시간이 중요한 시스템에서는 어셈블리로 코드를 더욱 최적화하여 프로그래밍할 수 있다는 것을 배울 수 있는 계기가 되었다. |