

# Microprocessor Lab Report

## Lab 03: 64-bit ARM Assembly Language

소속: 컴퓨터정보공학부

학번: 2022202090

이름: 남재현

이번 실험의 목표는 라즈베리파이의 우분투 서버에서 어셈블리 언어를 가지고 실습해보고 직접 어셈블리어를 능숙히 조작해보는 것이다.

## 1. 우분투 접속하기

```
nano@nano: ~
PS C:\Users\nwlef> ssh nano@192.168.55.1
nano@192.168.55.1's password:
Welcome to Ubuntu 18.04.5 LTS (GNU/Linux 4.9.201-tegra aarch64)
 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage
This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

16 updates can be applied immediately.
To see these additional updates run: apt list --upgradable

Last login: Mon Aug 30 18:30:37 2021 from 192.168.55.100
nano@nano:~$ ^C
```

## 2. 파일 업로드하기

```
PS C:\Users\nwlef> scp Desktop/MyDrive/광운대/마이크로프로세서/Assignment/3/Lab03_ref/LogicOP_ref.s nano@192.168.55.1:/home/nano/MPLAB/3/
nano@192.168.55.1's password:
LogicOP_ref.s                                100% 851      0.8KB/s   00:00
PS C:\Users\nwlef> scp Desktop/MyDrive/광운대/마이크로프로세서/Assignment/3/Lab03_ref/LogicShift_ref.s nano@192.168.55.1:/home/nano/MPLAB/3/
nano@192.168.55.1's password:
LogicShift_ref.s                            100% 890      55.8KB/s   00:00
PS C:\Users\nwlef> scp Desktop/MyDrive/광운대/마이크로프로세서/Assignment/3/Lab03_ref/Branch.s nano@192.168.55.1:/home/nano/MPLAB/3/
nano@192.168.55.1's password:
Branch.s                                    100% 827      51.8KB/s   00:00
PS C:\Users\nwlef>
```

Window PowerShell 에서 다음 형식의 scp 명령어를 이용하여 어셈블리 코드가 담긴 파일 3 개를 업로드 하였다.

scp 파일경로 username@주소:/폴더경로<sup>1</sup>

---

<sup>1</sup> 출처 : <https://linuxhandbook.com/transfer-files-ssh/>

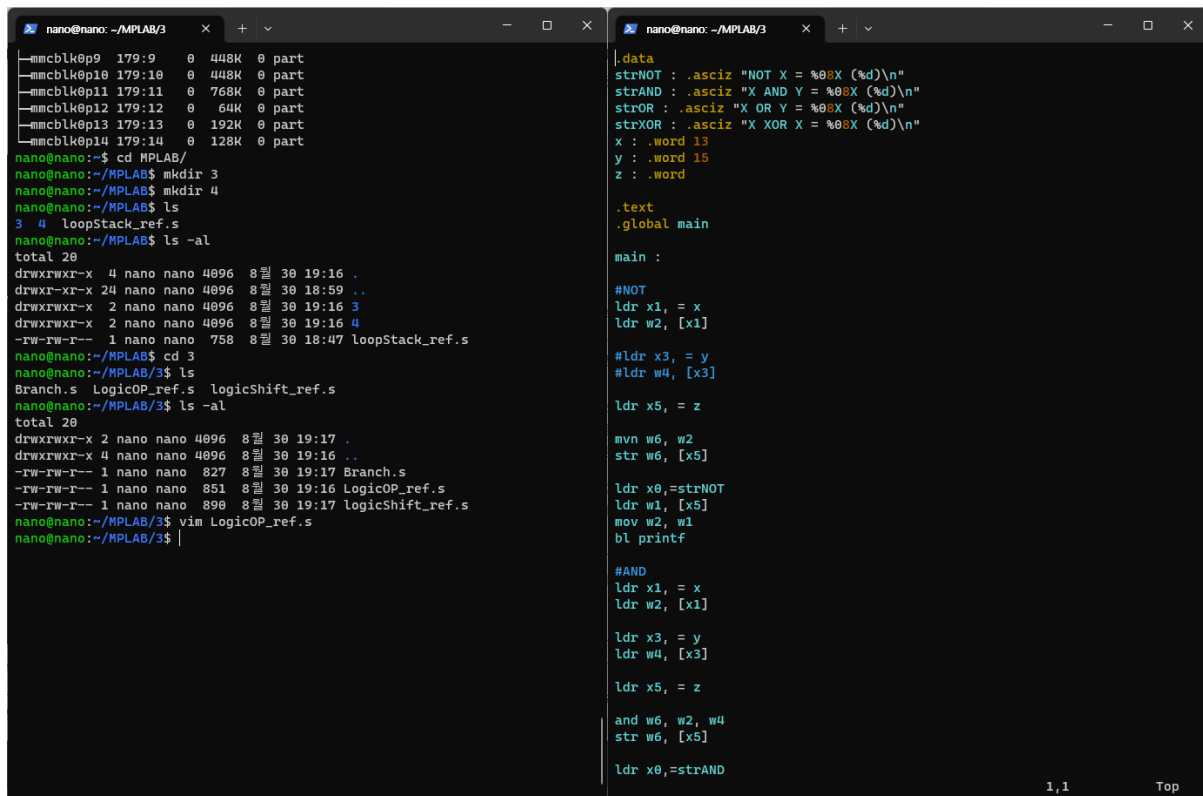
이 명령어는 컴퓨터에서 파일경로에 해당하는 파일을 리눅스 서버의 해당 폴더경로에 업로드한다.

어셈블리 코드가 담긴 파일 3 개의 이름 :

- Branch.s
- LogicOP\_ref.s
- logicShift\_ref.s

### 3. 분석할 어셈블리어 코드

아래 사진은 LogicOP\_ref.s 파일의 내용이다.



The image shows two terminal windows from a nano editor. The left window displays the contents of the LogicOP\_ref.s file, which includes assembly instructions for NOT, AND, and XOR operations. The right window shows the file management commands used to create and organize the files in the MPLAB/3 directory.

```
nano@nano: ~/MPLAB/3
--mmcblk0p9 179:9 0 448K 0 part
--mmcblk0p10 179:10 0 448K 0 part
--mmcblk0p11 179:11 0 768K 0 part
--mmcblk0p12 179:12 0 64K 0 part
--mmcblk0p13 179:13 0 192K 0 part
--mmcblk0p14 179:14 0 128K 0 part
nano@nano:~$ cd MPLAB/
nano@nano:~/MPLAB$ mkdir 3
nano@nano:~/MPLAB$ mkdir 4
nano@nano:~/MPLAB$ ls
3 4 loopStack_ref.s
nano@nano:~/MPLAB$ ls -al
total 20
drwxrwxr-x 4 nano nano 4096 8월 30 19:16 .
drwxr-xr-x 24 nano nano 4096 8월 30 18:59 ..
drwxrwxr-x 2 nano nano 4096 8월 30 19:16 3
drwxrwxr-x 2 nano nano 4096 8월 30 19:16 4
-rw-rw-r-- 1 nano nano 758 8월 30 18:47 loopStack_ref.s
nano@nano:~/MPLAB$ cd 3
nano@nano:~/MPLAB/3$ ls
Branch.s LogicOP_ref.s logicShift_ref.s
nano@nano:~/MPLAB/3$ ls -al
total 20
drwxrwxr-x 2 nano nano 4096 8월 30 19:17 .
drwxrwxr-x 4 nano nano 4096 8월 30 19:16 ..
-rw-rw-r-- 1 nano nano 827 8월 30 19:17 Branch.s
-rw-rw-r-- 1 nano nano 851 8월 30 19:16 LogicOP_ref.s
-rw-rw-r-- 1 nano nano 890 8월 30 19:17 logicShift_ref.s
nano@nano:~/MPLAB/3$ vim LogicOP_ref.s
nano@nano:~/MPLAB/3$
```

```
data
strNOT : .asciz "NOT X = %08X (%d)\n"
strAND : .asciz "X AND Y = %08X (%d)\n"
strOR : .asciz "X OR Y = %08X (%d)\n"
strXOR : .asciz "X XOR X = %08X (%d)\n"
x : .word 13
y : .word 15
z : .word

.text
.global main

main :

#NOT
ldr x1, = x
ldr w2, [x1]

#ldr x3, = y
#ldr w4, [x3]

ldr x5, = z

mvn w6, w2
str w6, [x5]

ldr x0,=strNOT
ldr w1, [x5]
mov w2, w1
bl printf

#AND
ldr x1, = x
ldr w2, [x1]

ldr x3, = y
ldr w4, [x3]

ldr x5, = z

and w6, w2, w4
str w6, [x5]

ldr x0,=strAND
```

```

ldr w1, [x5]
mov w2, w1
bl printf

#OR
ldr x1, = x
ldr w2, [x1]

ldr x3, = y
ldr w4, [x3]

ldr x5, = z

orr w6, w2, w4
str w6, [x5]

ldr x0,=strOR
ldr w1, [x5]
mov w2, w1
bl printf

#XOR
ldr x1, = x
ldr w2, [x1]

ldr x3, = y
ldr w4, [x3]

ldr x5, = z

eor w6, w2, w4
str w6, [x5]

ldr x0,=strXOR

```

```

ldr w1, [x5]
mov w2, w1
bl printf

mov x8, #94
mov x0, #0
svc 0

```

73,11 79% 87,0-1 Bot

우선 .data 아래에는 리터럴 상수 또는 변수가 적혀있다.

.text 아래부터 main 함수와 여러 코드들이 적혀있다.

주석을 추가하였다.

```

.data
strNOT : .asciz "NOT X = %0X (%d)\n"
strAND : .asciz "X AND Y = %0X (%d)\n"
strOR : .asciz "X OR Y = %0X (%d)\n"
x : .word 11 // 32bit integer
y : .word 15 // 32bit integer
z : .word // 32bit integer

.text
.global main
main :
#NOT
ldr x1, = x // x1에 x(=11:4byte)의 메모리 주소를 4byte 크기에 대입
ldr w2, [x1] // w2에 x1의 값(=11:4byte)을 4byte 크기에 대입

#AND
ldr x3, = y // x3에 y(=15:4byte)의 메모리 주소를 4byte 크기에 대입
ldr w4, [x3] // w4에 x3의 값(=15:4byte)을 4byte 크기에 대입

ldr x5, = z // x5에 z(초기화되지 않음)의 메모리 주소를 4byte 크기로 대입

eor w6, w2, w4 // w6에 w2(=11:4byte)와 w4(=15:4byte)를 bit-wise not 연산을 적용시켜 대입
str w6, [x5] // x5의 메모리값에 w6(=11:4byte)을 저장

ldr x0,=strNOT // x0에 strNOT의 주소를 로드
ldr w1, [x5] // w1에 x5의 메모리값(=11:4byte)을 로드
mov w2, w1 // w2에 w1의 값을 부여받기
bl printf // printf 호출

#AND
ldr x1, = x // 위의 값을
ldr w2, [x1] // 위의 값을

ldr x3, = y // 위의 값을
ldr w4, [x3] // 위의 값을

ldr x5, = z // same

and w6, w2, w4 // and연산을 적용(=11)하여 w6에 대입
str w6, [x5] // x5의 메모리값에 w6(=11)을 저장

ldr x0,=strAND // x0에 strAND의 메모리 주소를 로드
ldr w1, [x5] // w1에 x5의 값(=11)을 로드
mov w2, w1 // w2에 w1의 값(=11)을 부여받기
bl printf // printf 호출

#OR
ldr x1, = x //
-- INSERT --

```

```

and w6, w2, w4 // and연산을 적용(=11)하여 w6에 대입
str w6, [x5] // x5의 메모리값에 w6(=11)을 저장

ldr x0,=strAND // x0에 strAND의 메모리 주소를 로드
ldr w1, [x5] // w1에 x5의 값(=11)을 로드
mov w2, w1 // w2에 w1의 값(=11)을 부여받기
bl printf // printf 호출

#XOR
ldr x1, = x //
ldr w2, [x1] //

ldr x3, = y //
ldr w4, [x3] //

ldr x5, = z //

eor w6, w2, w4 // w2와 w4를 exclusive or 연산 후 w6에 저장.(2를 저장)
str w6, [x5] //

ldr x0,=strXOR // strXOR의 메모리 주소를 x0에 로드
ldr w1, [x5] // w1에 x5의 메모리값(=11)을 로드
mov w2, w1 // w2에 w1의 값(=11)을 부여받기
bl printf // 함수 printf 호출

mov x8, #94
mov x0, #0
svc 0

-- INSERT --

```

29,56-65 Top 60,1 Bot

Mnemonic	Instruction	Action	See Section:
SBC	Subtract with Carry	Rd := Rn - Op2 - 1 + Carry	4.5
STC	Store coprocessor register to memory	address := CRn	4.15
STM	Store Multiple	Stack manipulation (Push)	4.11
STR	Store register to memory	<address> := Rd	4.9, 4.10
SUB	Subtract	Rd := Rn - Op2	4.5
SWI	Software Interrupt	OS call	4.13
SWP	Swap register with memory	Rd := [Rn], [Rn] := Rm	4.12
TEQ	Test bitwise equality	CPSR flags := Rn EOR Op2	4.5
TST	Test bits	CPSR flags := Rn AND Op2	4.5

Table 4-1: The ARM Instruction set (Continued)

str instruction 은 store register to memory 이다. 예를 들어서 str w1, [x2]이면 주소가 x2 인 메모리 공간에 w1 을 저장하라는 뜻이다. x2 는 64bit 이고 메모리의 주소 접근도 64bit 이다. 메모리 한 셀의 공간은 32bit 로 w register 를 사용해야한다.<sup>2</sup>

맨 밑에 mov 인스트럭션 2 개와 svc 인스트럭션은 return 0;의 의미인 것 같다.

아래 사진은 logicShift\_ref.s 파일의 내용이다.

```

nano@nano: ~/MPLAB/3
.data
str1 : .asciz "X << Y = %08X (%d)\n"
str2 : .asciz "X >> Y = %08X (%d)\n"
str3 : .asciz "~X << Y = %08X (%d)\n"
str4 : .asciz "~X >> Y = %08X (%d)\n"
x : .word 15
y : .word 3
z : .word

.text
.global main
main :

#X << Y

ldr x1, = x
ldr w2, [x1]

ldr x3, = y
ldr w4, [x3]

ldr x5, = z

lsl w6, w2, w4
str w6, [x5]

ldr x0,=str1
ldr w1, [x5]
mov w2, w1
bl printf

#X >> Y

ldr x1, = x
ldr w2, [x1]

ldr x3, = y
ldr w4, [x3]

ldr x5, = z

lsl w6, w2, w4
str w6, [x5]

ldr x0,=str2
ldr w1, [x5]

mov w2, w1
bl printf

#~X << Y

ldr x1, = x
ldr w2, [x1]
mvn w2, w2

ldr x3, = y
ldr w4, [x3]

ldr x5, = z
#ldr w6, w2, w4
asr w6, w2, w4
str w6, [x5]

ldr x0,=str3
ldr w1, [x5]
mov w2, w1
bl printf

#~X >> Y

ldr x1, = x
ldr w2, [x1]
mvn w2, w2

ldr x3, = y
ldr w4, [x3]

ldr x5, = z
#ldr w6, w2, w4
asr w6, w2, w4
str w6, [x5]

ldr x0,=str4
ldr w1, [x5]
mov w2, w1
bl printf

mov x8, 94
mov x0, 0
svc 0

```

주석을 붙여 설명해보았다.

<sup>2</sup> 출처 : ARM7TDMI-S Data Sheet. 과제 폴더 내의 파일 참고

```

|data
str1 : .asciz "X <= Y = %dX (%d)\n"
str2 : .asciz "X > Y = %dX (%d)\n"
str3 : .asciz "~X <= Y = %dX (%d)\n"
str4 : .asciz "~X > Y = %dX (%d)\n"
x : .word 15 // 8byte 크기의 메모리 주소값 0000 0000 0000 1111
y : .word 3 // 8byte 크기의 메모리 주소값
z : .word // 8byte 크기의 메모리 주소값

.text
.global main
main :
#X <= Y

ldr x1, = x // x의 메모리 주소를 레지스터 x1에 대입
ldr w2, [x1] // 메모리[x1]에 저장된 값(=15)을 w2에 대입

ldr x3, = y // y의 메모리 주소를 레지스터 x3에 대입
ldr w4, [x3] // 메모리[x3]에 저장된 값(=3)을 w4에 대입

ldr x5, = z // z의 메모리 주소를 레지스터 x5에 대입

lsl w6, w2, w4 // 레지스터 w6에 128(= w2 << w4 = 0b1111000)을 대입
str w6, [x5] // 주소가 x5인 메모리에 레지스터 w6값을 저장

ldr x0,=str1 // str1의 메모리 주소를 x0에 로드
ldr w1, [x5] // w1 = *x5
mov w2, w1 // w2 = w1
bl printf // printf(x0, w1, w2)

#X > Y

ldr x1, = x // x1 = x
ldr w2, [x1] // w2 = *x1

ldr x3, = y // x3 = y
ldr w4, [x3] // w4 = *x3

ldr x5, = z // x5 = z

lsl w6, w2, w4 // w6 = w2 << w4
str w6, [x5] // *x5 = w6

ldr x0,=str2 // x0 = str2
ldr w1, [x5] // w1 = *x5
mov w2, w1 // w2 = w1
bl printf // printf(x0, w1, w2)

*logicShift_ref.s* 91L, 2135C
1,1 Top

lsl w6, w2, w4 // w6 = w2 << w4
str w6, [x5] // *x5 = w6

ldr x0,=str3 // char *x0 = str3;
ldr w1, [x5] // int w1 = *x5;
mov w2, w1 // w2 = w1;
bl printf // printf(x0, w1, w2);

#X >= Y

ldr x1, = x // int *x1 = x;
ldr w2, [x1] // int w2 = *x1;
movn w2, w2 // w2 = ~w2;

ldr x3, = y // int *x3 = y;
ldr w4, [x3] // int w4 = *x3;

ldr x5, = z // int *x5 = z;

lsl w6, w2, w4 // int w6 = w2 << w4
str w6, [x5] // *x5 = w6;

ldr x0,=str4 // char *x0 = str4;
ldr w1, [x5] // int w1 = *x5;
mov w2, w1 // w2 = w1;
bl printf // printf(x0, w1, w2);

mov x8, w4
mov x8, 0
svc 0
"
```

중간에 ldr w6, w2, w4 은 무슨 뜻인지 모르겠다. w4 가 오프셋이라면 int w6 = w2 + w4; 정도의 뜻일텐데 맥락상으로 보아 주석인가? 싶다.

이제 규칙을 찾아 볼 수 있는데 (1) int\* 자료형의 크기는 64bit 이므로 x register 가 쓰였다는 것을 알 수 있다. (2) 포인터 변수의 값을 참조하기 위하여 꺾쇠 기호 '[', ']'를 사용한다. (3) bl instruction 은 함수를 호출한다. (4) x0, x1, x2, ...은 함수의 인자로 쓰인다. 이 외에도 더 많은 규칙들을 알아냈다.

아래는 Branch.s 파일의 내용이다.

```
nano@nano: ~/MPLAB/3
.data
num: .asciz "%d"
Enter1: .asciz "Enter n1: "
Enter2: .asciz "Enter n2: "
Enter3: .asciz "Enter n3: "
str4: .asciz "The smallest number is %d\n"

.align
n1: .space 4
n2: .space 4
n3: .space 4
min: .space 4

.text
.global main
main:
#Enter 1
ldr x0, = Enter1
bl printf
ldr x0, = num
ldr x1, = n1
bl scanf

#Enter 2
ldr x0, = Enter2
bl printf
ldr x0, = num
ldr x1, = n2
bl scanf

#Enter 3
ldr x0, = Enter3
bl printf
ldr x0, = num
ldr x1, = n3
bl scanf

ldr x1, = n1
ldr x2, = n2
ldr x3, = n3
ldr x4, = min

ldr w5, [x1]
ldr w6, [x2]
ldr w7, [x3]

#comparing
if1:
cmp --, --
bge else1

if11:
cmp --, --
bge else11
str --, [--]
b prt

else1:
str --, [--]
b prt

else11:
if2:
cmp --, --
bge else21
str --, [--]
b prt

else21:
str --, [--]

prt:
ldr x0, = str4
ldr w1, [x4]
bl printf

mov x8, #94
mov x0, #0
svc 0

"Branch.s" 85L, 827C 35,9 Top 65,6 Bot
```

분석 및 주석은 5 번에 포함

## 4. 어셈블리어 컴파일 후 실행

```
nano@nano: ~/MPLAB/3
nano@nano:~/MPLAB/3$ gcc LogicOP_ref.s -o LogicOP --static
nano@nano:~/MPLAB/3$ ./LogicOP
NOT X = FFFFFFF2 (-14)
X AND Y = 0000000D (13)
X OR Y = 0000000F (15)
X XOR X = 00000002 (2)
nano@nano:~/MPLAB/3$ |
```

위의 사진은 LogicOP\_ref.s 어셈블리어 파일을 실행시킨 결과이다.

```
nano@nano: ~/MPLAB/3
nano@nano:~/MPLAB/3$ gcc logicShift_ref.s -o logicShift --static
nano@nano:~/MPLAB/3$ ./logicShift
X << Y = 00000078 (120)
X >> Y = 00000001 (1)
~X << Y = FFFFFFF80 (-128)
~X >> Y = FFFFFFFE (-2)
nano@nano:~/MPLAB/3$
```

위의 사진은 logicShift\_ref.s 어셈블리어 파일을 실행시킨 결과이다.

이 실행결과의 right shift 에서 ~X 는 MSB 가 1 인 음수(=-16)이고 음수(=-16)를 Y(=3)만큼 right shift 를 했는데 그대로 음수(=-2)이므로 **logical shift 가 아닌 부호를 유지하며 시프트하는 arithmetic shift** 이다. 어셈블리 코드에서 asr 의 니모닉 기호는 arithmetic right shift 로 유추해볼 수 있다.

```
nano@nano: ~/MPLAB/3
nano@nano:~/MPLAB/3$ gcc Branch.s -o Branch --static
Branch.s: Assembler messages:
Branch.s:52: Error: operand 1 must be an integer or stack pointer register -- `cmp --,--'
Branch.s:56: Error: operand 1 must be an integer or stack pointer register -- `cmp --,--'
Branch.s:58: Error: operand 1 must be an integer register -- `str --,[--]'
Branch.s:62: Error: operand 1 must be an integer register -- `str --,[--]'
Branch.s:67: Error: operand 1 must be an integer or stack pointer register -- `cmp --,--'
Branch.s:69: Error: operand 1 must be an integer register -- `str --,[--]'
Branch.s:73: Error: operand 1 must be an integer register -- `str --,[--]'
nano@nano:~/MPLAB/3$
```

위의 사진은 Branch.s 어셈블리어 파일을 컴파일 시킨 결과이다.

Branch.s 의 내용에는 ‘—’라는 빈칸이 있고 ‘—’는 어셈블리어 코드로 해석되지 않는 것 같다.

## 5. Branch.s 분석 및 빈칸 채우기 후 실행



```

nano@nano: ~/MPLAB/3
.data // 리터럴 상수와 변수 모음집
num : .asciz "nd" // const char* (64bit)
Enter1 : .asciz "Enter n1: " // const char* (64bit)
Enter2 : .asciz "Enter n2: " // const char* (64bit)
Enter3 : .asciz "Enter n3: " // const char* (64bit)
str4 : .asciz "The smallest number is %d\n" // const char* (64bit)

.align
n1 : .space 4 // int* (64bit)
n2 : .space 4 // int* (64bit)
n3 : .space 4 // int* (64bit)
min : .space 4 // int* (64bit)

.text // 실행 가능한 코드를 모음집
.global main
main : // int main(){}

#Enter 1
ldr x0, = Enter1 // char *x0 = Enter1;
bl printf // printf(x0); // printf("Enter n1: ");
ldr x0, = num // char *x0 = num; // "nd"
ldr x1, = n1 // int *x1 = n1;
bl scanf // scanf(x0, x1); // scanf("%d", n1); ??? 뭔가 이상한데...

#Enter 2
ldr x0, = Enter2 // char *x0 = Enter2;
bl printf // printf(x0); // printf("Enter n2: ");
ldr x0, = num // char *x0 = num;
ldr x1, = n2 // int *x1 = n2;
bl scanf // scanf(x0, x1); // scanf("%d", n2);

#Enter 3
ldr x0, = Enter3 // char *x0 = Enter3;
bl printf // printf(x0); // printf("Enter n3: ");
ldr x0, = num // char *x0 = num;
ldr x1, = n3 // int *x1 = n3;
bl scanf // scanf(x0, x1); // scanf("%d", n3);

ldr x1, =n1 // int *x1 = n1;
ldr x2, =n2 // int *x2 = n2;
ldr x3, =n3 // int *x3 = n3;
ldr x4, = min // int *x4 = min;

ldr x4, = min // int *x4 = min;

#comparing
if1:
cmp w5, w6 // if(n1 < n2) { }
bge else1 // else { goto else1; }
|
if11:
cmp w5, w7 // if(n1 < n3) { }
bge else11 // else { goto else11; }
str w5, [x4] // *min = w5; // n1이 최소값
b prt // if문 종료

else1:
str w7, [x4] // *min = w7 // n3가 최소값
b prt // if문 종료

else11:
if2:
cmp w6, w7 // if(n2 < n3) { }
bge else21 // else { goto else21; }
str w6, [x4] // *min = w6; // n2가 최소값
b prt // if문 종료

else21:
str w7, [x4] // *min = w7 // n3가 최소값

prt:
ldr x0, = str4 // char *x0 = str4;
ldr w1, [x4] // int w1 = *x4;
bl printf // printf(x0, w1); // printf("The smallest number is %d\n", w1);

mov x8, #04
mov x0, #0
svc 0 // return 0;

~
*Branch.s* 85L, 2089C 12,38 Top 54,0-1 Bot

```

새로 알게 된 사실 : .data 에 적혀있는 값들이 전부 리터럴 상수인줄 알았는데 변수도 있었다. scanf 의 2 번째 인자에 n1 값이 들어가는데 만약에 n1 이 리터럴 상수값이라면 scanf 를 호출해도 값이 바뀌지 않을 것이다.

ldr w5, [x1]과 ldr w6, [x2]와 ldr w7, [x3]를 지우고 w5, w6, w7 을 [x1], [x2], [x3]로 대체하면 인스트럭션 3 개를 더 단축시킬 수 있다.

if(n1 < n2) min = n1;

if(n3 < min) min = n3;

이렇게 두줄이면 인스트럭션을 훨씬 더 단축시킬 수 있다.

```

nano@nano: ~/MPLAB/3
assign.c Branch.s LogicOP LogicOP_ref.s LogicShift LogicShift_ref.s
nano@nano:~/MPLAB/3$ vim LogicOP_ref.s
nano@nano:~/MPLAB/3$ vim LogicOP_ref.s
nano@nano:~/MPLAB/3$ vim LogicShift_ref.s
nano@nano:~/MPLAB/3$ vim LogicShift_ref.s
nano@nano:~/MPLAB/3$ vim Branch.s
nano@nano:~/MPLAB/3$ gcc Branch.s -o Branch --static
nano@nano:~/MPLAB/3$ ./Branch
Enter n1: 1
Enter n2: 2
Enter n3: 3
The smallest number is 1
nano@nano:~/MPLAB/3$ ./Branch
Enter n1: 2
Enter n2: 1
Enter n3: 3
The smallest number is 1
nano@nano:~/MPLAB/3$ ./Branch
Enter n1: 3
Enter n2: 2
Enter n3: 1
The smallest number is 1
nano@nano:~/MPLAB/3$ ./Branch
Enter n1: 1
Enter n2: 3
Enter n3: 2
The smallest number is 1
nano@nano:~/MPLAB/3$
nano@nano:~/MPLAB/3$ 2
-bash: 2: command not found
nano@nano:~/MPLAB/3$ 3
-bash: 3: command not found
nano@nano:~/MPLAB/3$ 1
-bash: 1: command not found
nano@nano:~/MPLAB/3$ ./Branch
Enter n1: 2
Enter n2: 3
Enter n3: 1
The smallest number is 1
nano@nano:~/MPLAB/3$ ./Branch
Enter n1: 3
Enter n2: 1
Enter n3: 2
The smallest number is 1
nano@nano:~/MPLAB/3$ |

```

잘 작동하는 것을 확인할 수 있다.

## 6. Questions

64 비트 연산은 파워를 2 배로 소비한다.(32 비트 연산에 비해) 다른 한편으로는, 대부분의 데이터는 32 비트 워드에 잘 맞는다. 하지만 왜 64 비트 마이크로프로세서를 써야할까?

64 비트 마이크로프로세서를 사용해야 하는 이유는 64 비트 마이크로프로세서는 32 비트 마이크로세서보다 **단순히 2 배 더 많은 전력을 소모하는 것이 아니라 아키텍처를 개선함으로써 더 큰 성능 향상으로 이어지거나 더 좋은 최적화가 가능해지기** 때문이다.

64 비트 마이크로프로세서를 사용해야 하는 이유는 32 비트 마이크로프로세서에서는 4GB 의 램까지 밖에 인덱싱할 수 밖에 없으며, 나노보드에서 많은 데이터를 처리하는 엣지 컴퓨팅에서 부적합할 수 있다.

32 비트 마이크로세서로 호환이 되지 않는 경우에도 64 비트 마이크로프로세서에 이점이 있고, 더 큰 메모리 맵핑이 가능해지면서 보안 기능이 더 뛰어나진다. 64 비트가 업계 표준이 되면서 64 비트 하드웨어를 사용하는 것이 경제적으로 더 이득이 될 수 있다.

## 7. 결과물 다운로드

```
PS C:\Users\nwlef> scp nano@192.168.55.1:/home/nano/MPLAB/3/Branch.s ./Desktop/
nano@192.168.55.1's password:
Branch.s                                     100% 2089      2.0KB/s   00:00
PS C:\Users\nwlef> scp nano@192.168.55.1:/home/nano/MPLAB/3/Branch ./Desktop/
nano@192.168.55.1's password:
Branch                                     100% 610KB    12.6MB/s   00:00
PS C:\Users\nwlef> |
```

scp 명령어를 이용하여 Branch.s 파일과 Branch 파일을 다운로드