Computer Architecture

배경설명

- **Lab0 ~ Lab6에서는 32 bit ARM CPU를 위한 어셈블리 프로그래밍을 공부해왔습니다.**
- **Lab7에서는 64 bit ARM CPU를 위한 어셈블리 프로그래밍을 소개합니다.**
  - 개발환경을 64비트 버전으로 바꾸어 설치합니다. cross compiler 및 QEMU emulator.
  - 64비트 버전의 어셈블리 코드를 사용합니다.
  - Gdbgui를 포함하여 나머지는 그대로 사용합니다.
- **Lab7은 참고용 자료입니다. 향후 64 bit ARM 어셈블리 프로그래밍이 필요하면 찾아보세요. 실제 Lab은 진행하지 않습니다.**

Contents

설치 가이드 - Compiler

- 64bit ARM, AARCH64를 지원하는 gcc 기반의 컴파일러 사용
- 아래와 같이 커맨드를 이용하여 설치

```
$ sudo apt-get install gcc-aarch64-linux-gnu
```

- 컴파일러 설치 확인

```
youngjoon@ubuntu:~/ca_lab/arm64$ aarch64-linux-gnu-gcc
aarch64-linux-gnu-gcc: fatal error: no input files
compilation terminated.
youngjoon@ubuntu:~/ca_lab/arm64$
```

설치 가이드 - Qemu

- Host PC에서 ARM machine을 테스트 할 수 있도록 해주는 가상 에뮬레이터인 Qemu를 설치한다.
- 아래와 같이 커맨드를 이용하여 설치

```
$ sudo apt-get install qemu
```

- Qemu 설치 확인

```
youngjoon@ubuntu:~/ca_lab/arm64$ ls /usr/bin/qemu*
/usr/bin/qemu-aarch64
/usr/bin/qemu-alpha
/usr/bin/qemu-arm
/usr/bin/qemu-armeb
/usr/bin/qemu-cris
/usr/bin/qemu-i386
/usr/bin/qemu-img
/usr/bin/qemu-io
/usr/bin/qemu-m68k
/usr/bin/qemu-microblaze
/usr/bin/qemu-microblazeel
/usr/bin/qemu-mips
/usr/bin/qemu-mips64
/usr/bin/qemu-mips64el
/usr/bin/qemu-system-alpha
/usr/bin/qemu-system-arm
/usr/bin/qemu-system-cris
/usr/bin/qemu-system-i386
/usr/bin/qemu-system-lm32
/usr/bin/qemu-system-m68k
/usr/bin/qemu-system-microblaze
/usr/bin/qemu-system-microblazeel
/usr/bin/qemu-system-mips
/usr/bin/qemu-system-mips64
/usr/bin/qemu-system-mips64el
/usr/bin/qemu-system-mipsel
/usr/bin/qemu-system-moxie
/usr/bin/qemu-system-or32
```

설치 가이드 - Debugger

- gdb-multiarch는 여러 가지 architecture를 지원하는 GNU debugger이다. ARM 바이너리를 디버깅 할 수 있도록 해준다.
- 아래와 같이 커맨드를 이용하여 설치

```
$ sudo apt-get install gdb-multiarch
```

- gdb-multiarch 설치 확인

```
youngjoon@ubuntu:~/ca_lab/arm64$ ls /usr/bin/gdb-multiarch
/usr/bin/gdb-multiarch
```

설치 가이드 - gdbgui

- gdbgui는 앞서 설치한 디버거를 그래픽 환경에서 사용할 수 있도록 도와준다.
- 아래 커맨드를 차례대로 입력

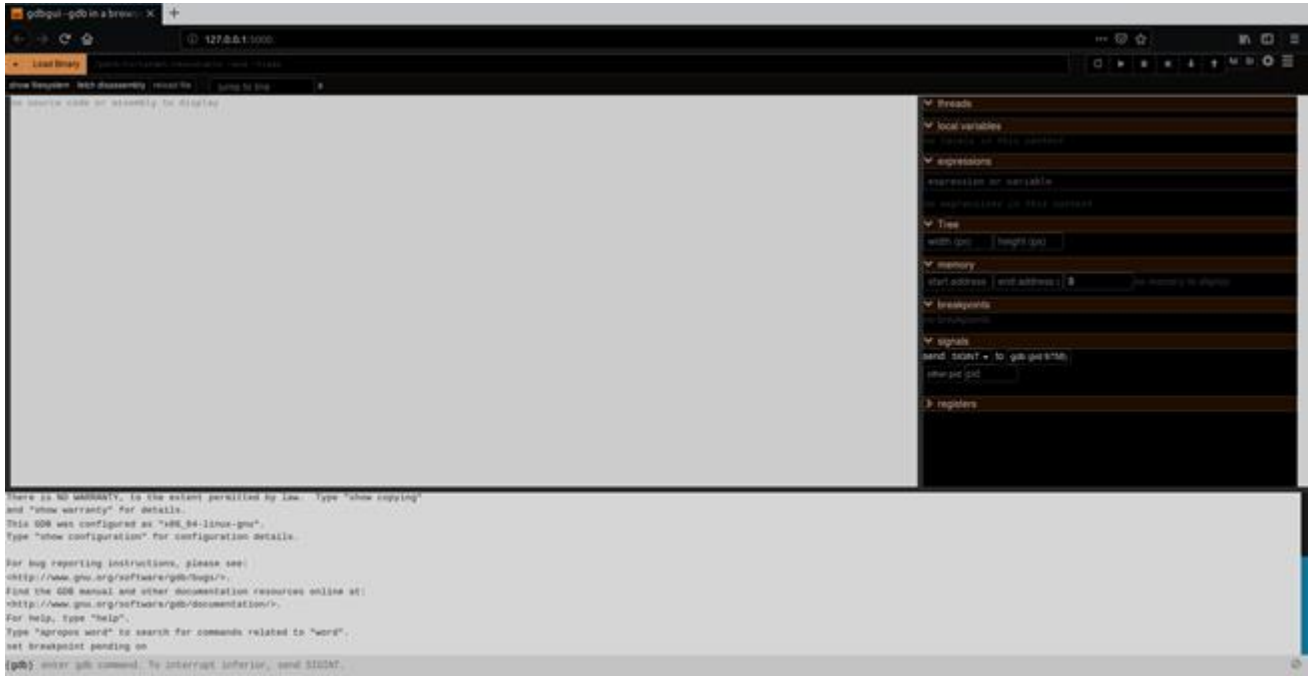
```
$ wget https://bootstrap.pypa.io/get-pip.py -O /tmp/get-pip.py
$ sudo python3 /tmp/get-pip.py
$ pip install --user pipenv
$ pip3 install --user pipenv
$ echo "PATH=$HOME/.local/bin:$PATH" >> ~/.profile
$ source ~/.profile
$ whereis pip
$ sudo pip install gdbgui
```

설치 가이드 - gdbgui

- **gdbgui 설치 확인**

```
$ gdbgui -g gdb-multiarch
```

```
yj@ubuntu:~/lab0$ gdbgui -g gdb-multiarch
Opening gdbgui with default browser at http://127.0.0.1:5000
exit gdbgui by pressing CTRL+C
```



사용법

- **컴파일**

```
$ aarch64-linux-gnu-gcc -o [executable file] [source file] -g -static
```

  - source file: 소스 파일 이름
  - executable file: 원하는 실행 파일 이름

- **Qemu 실행**

```
$ qemu-aarch64 -g 8080 [executable file]
```

  - 실행 커맨드를 입력하면 해당 터미널은 대기 상태가 된다.

- **GDB 실행**

```
$ gdbgui -g gdb-multiarch
```

  - 새로운 터미널을 열어-g 옵션의 인자로 gdb-multiarch를 주고 gdbgui 실행

사용 예제

Network Computing Lab.

컴파일

$ aarch64-linux-gnu-gcc -g -o main main.c -static

```
youngjoon@ubuntu:~/ca_lab/arm64$ aarch64-linux-gnu-gcc -g -o main main.c -static
youngjoon@ubuntu:~/ca_lab/arm64$ ls -l
total 616
-rwxrwxr-x 1 youngjoon youngjoon 620664 Nov 16 21:51 main
-rw-rw-r-- 1 youngjoon youngjoon    117 Nov 16 19:42 main.c
-rw-rw-r-- 1 youngjoon youngjoon    306 Nov 16 20:29 start.sh
youngjoon@ubuntu:~/ca_lab/arm64$
```

qemu 실행

$ qemu-aarch64 -g 8080 ./main

```
youngjoon@ubuntu:~/ca_lab/arm64$ qemu-aarch64 -g 8080 ./main
```

main.c

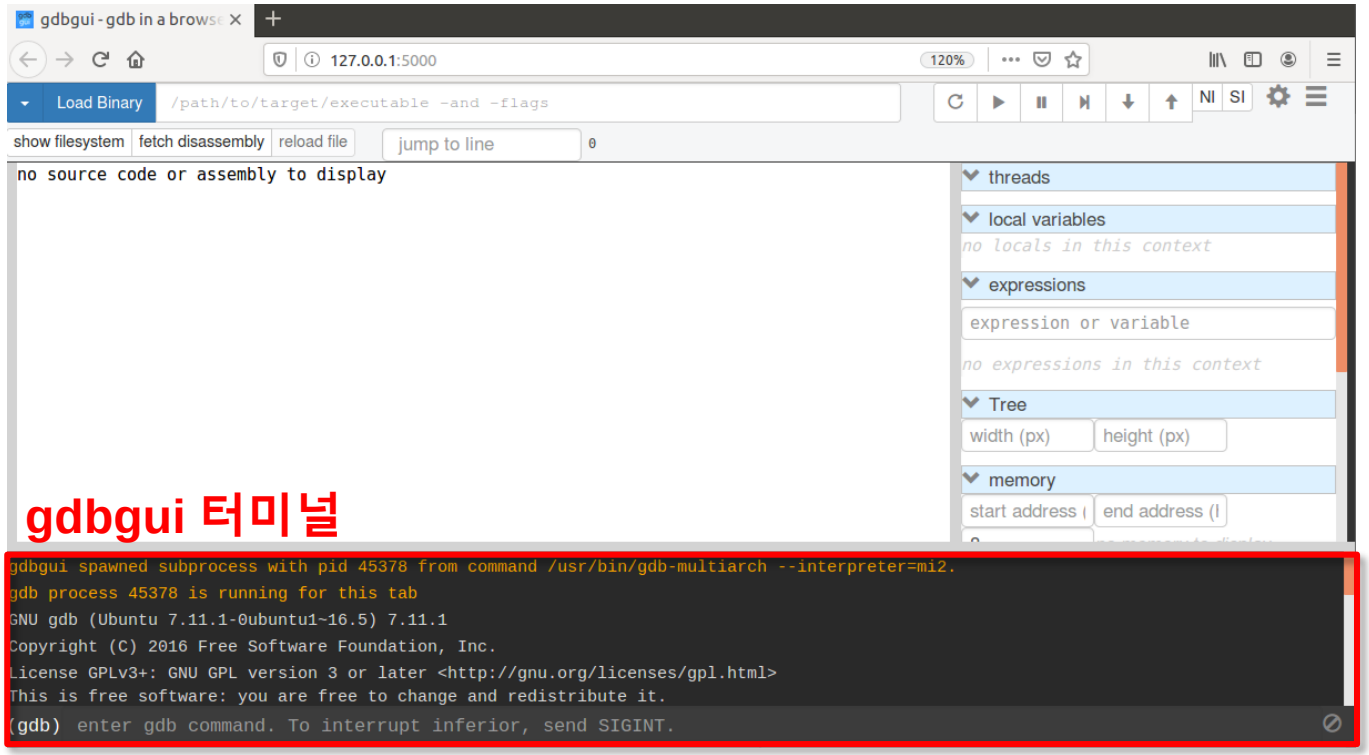
```
1  #include <stdio.h>
2
3  int main()
4  {
5    int i = 0;
6
7    for (i = 0; i < 5; i++)
8      printf("hello world!\n");
9
10   return 0;
11 }
```

사용 예제

- 새로운 터미널에서 gdbgui 실행

```
$ gdbgui -g gdb-multiarch
```

```
youngjoon@ubuntu:~/ca_lab$ gdbgui -g gdb-multiarch
Opening gdbgui with default browser at http://127.0.0.1:5000
exit gdbgui by pressing CTRL+C
```



사용 예제

- **gdbgui 터미널에서 qemu, gdb 연결**

```
$ file main
$ target remote:8080
```

```
Type "apropos word" to search for commands related to "word".
set breakpoint pending on
file main
file main
Reading symbols from main...
done.
(gdb) enter gdb command. To interrupt inferior, send SIGINT.
Reading symbols from main...
done.
target remote:8080
target remote:8080
Remote debugging using :8080
0x00000000004004c8 in _start ()
(gdb) enter gdb command. To interrupt inferior, send SIGINT.
```

사용 예제

- **64-ARM Assembly code**

```
1  #include <stdio.h>
2
3  int main()
4  {
5          int i = 0;
6
7          for (i = 0; i < 5; i++)
8                  printf("hello world!\n");
9
```

```
0x400658 stptx29, x30, [sp,#-32]!          main+0
0x40065c movtx29, sp                       main+4
> 0x400660 strtwzr, [x29,#28]              main+8
0x400678 ldrtw0, [x29,#28]                 main+32
0x40067c addtw0, w0, #0x1                  main+36
0x400680 strtw0, [x29,#28]                 main+40
0x400684 ldrtw0, [x29,#28]                 main+44
0x400688 cmptw0, #0x4                      main+48
0x40068c b.let0x40066c <main+20>           main+52
0x40066c adrptx0, 0x45e000 <free_mem+56>   main+20
0x400670 addtx0, x0, #0xde8                main+24
0x400674 blt0x406f00 <puts>                main+28
```

사용 예제

- 32 비트에서는 32 비트 범용레지스터 15개 사용 (R0 ~ R14)
- 64 비트에서는 64 비트 범용레지스터 31개 사용(X0 ~ X30), 32 비트를 지정할 경우 각 레지스터 하위 32비트를 사용가능(W0~W30)

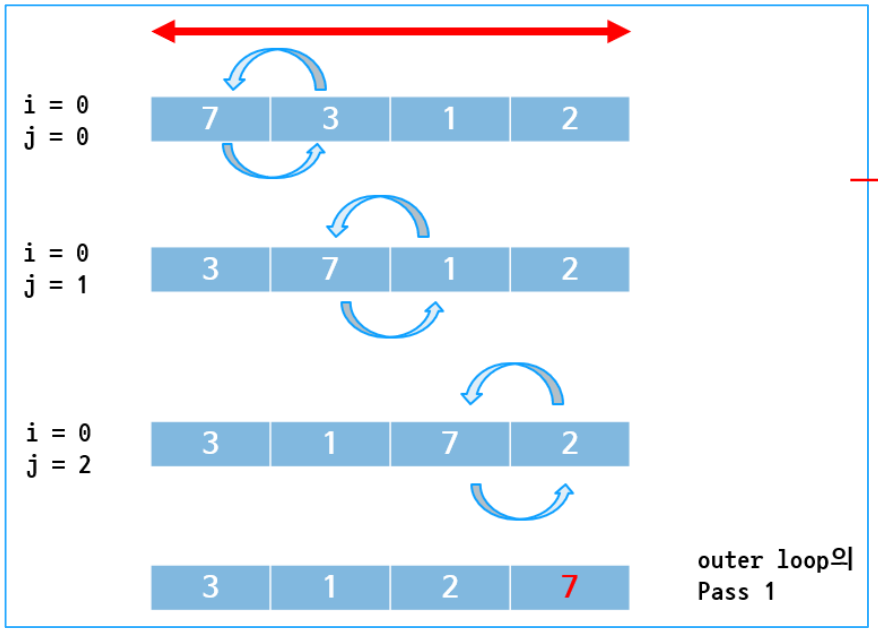
Network Computing Lab.

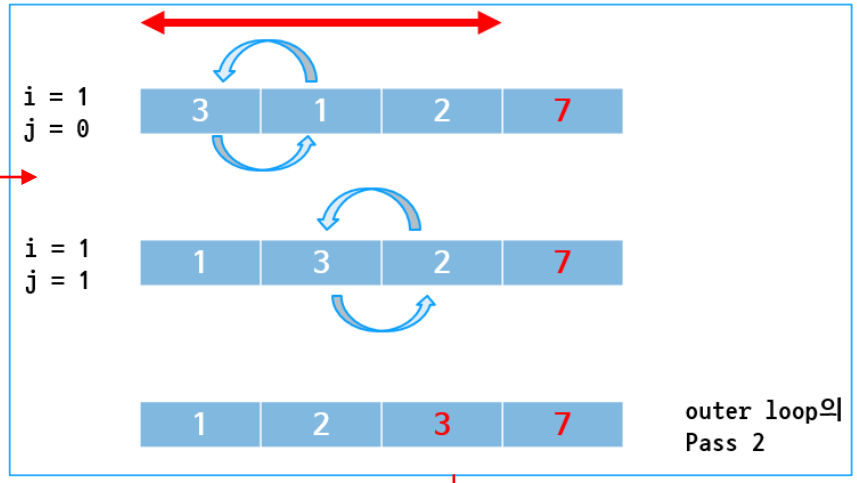
64-ARM register

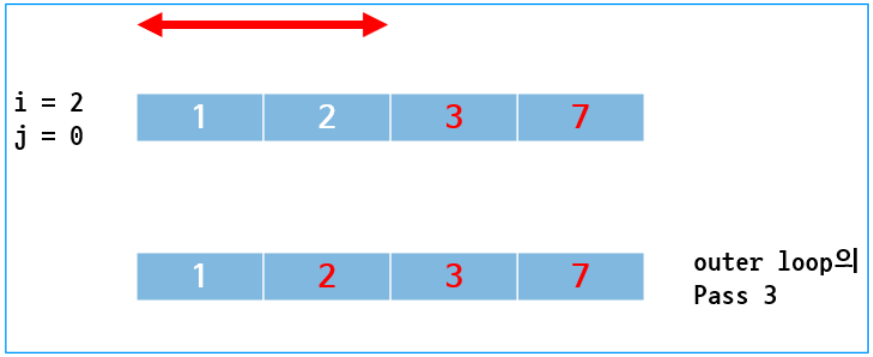
registers

name	value (hex)	value (decimal)
x0	0x1	1
x1	0x40007fffa8	274886295464
x2	0x40007fffb8	274886295480
x3	0x400658	4195928
x4	0x0	0
x5	0x0	0
x6	0x40007ffec8	274886295240
x7	0x40000	262144
x8	0x10000000000000000	18446744073709552000
x9	0x3fffffff	1073741823
x10	0x20000000	536870912
x11	0x40000	262144
x12	0x418ce0	4295904
x13	0x490000	4784128
x14	0x490000	4784128
x15	0x490000	4784128
x16	0x490000	4784128
x17	0x490000	4784128
x18	0x48d000	4771840
x19	0x4001d8	4194776
x15	0x490000	4784128
x16	0x490000	4784128
x17	0x490000	4784128
x18	0x48d000	4771840
x19	0x4001d8	4194776
x20	0x400ce8	4197608
x21	0x400da0	4197792
x22	0x0	0
x23	0x0	0
x24	0x4001d8	4194776
x25	0x0	0
x26	0x0	0
x27	0x0	0
x28	0x0	0
x29	0x40007ffe30	274886295088
x30	0x4008e4	4196580
sp	0x40007ffe30	274886295088
pc	0x400660	4195936
cpsr	0x60000000	1610612736

Lab 7 : 64 bit bubble sort (방법1)

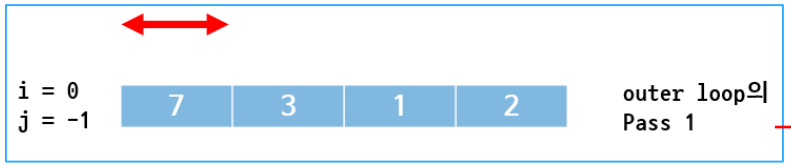




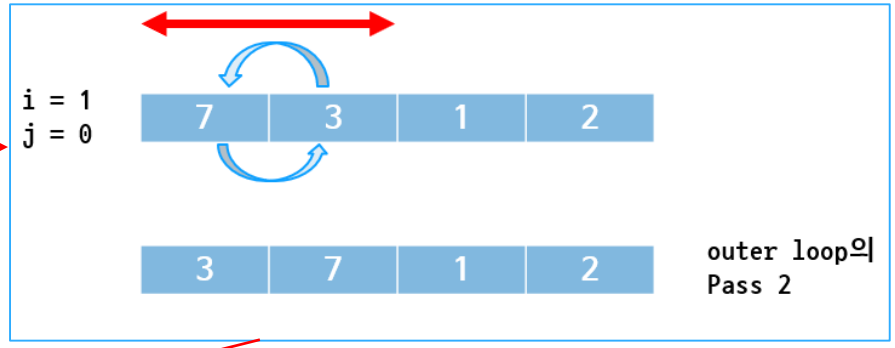


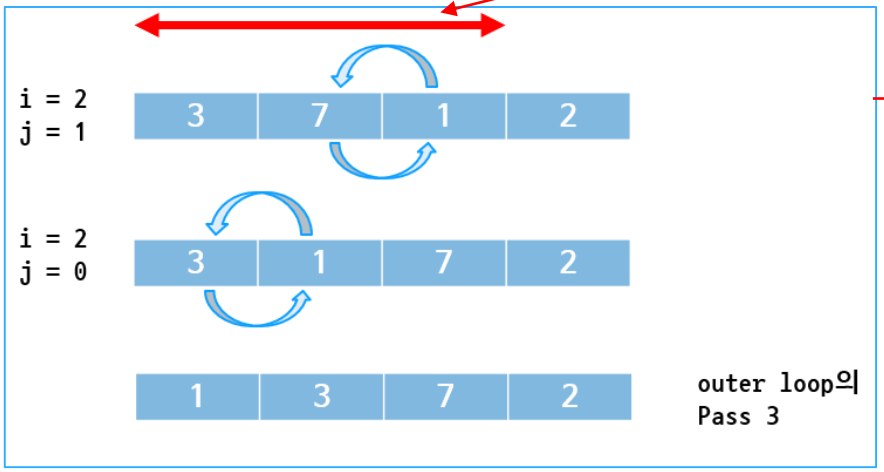

```
void sort (long v[], int n)
{
  int i, j;
  for (i = 0; i < n-1; i ++)

    for (j = 0; j < n-1-i; j ++)

      if (v[j] > v[j + 1]) swap(v,j);
}
```

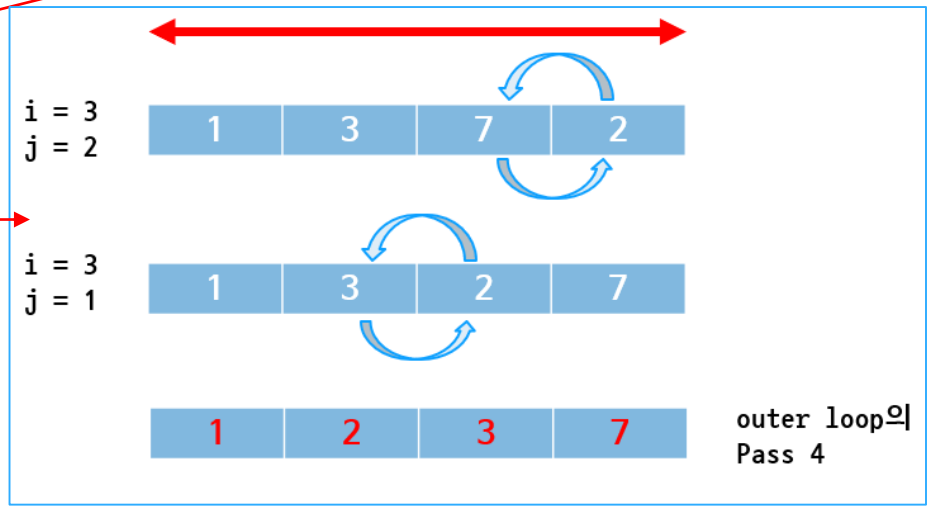
Lab 7 : 64 bit bubble sort (방법2-교재내용)




```
void sort (long v[], int n)
{
  int i, j;
  for (i = 0; i < n; i ++)

    for (j = i - 1; j >= 0 ; j --)

      if (v[j] > v[j + 1]) swap(v,j);
}
```







Register allocation and saving registers for *sort*

Network Computing Lab.

Register allocation

v	**X0**	**; 1st argument address of v**
n	**X1**	**; 2nd argument index n**
i	**X19**	**; local variable i**
j	**X20**	**; local variable j**
vjAddr	**X11**	**; to hold address of v[j]**
vj	**X12**	**; to hold a copy of v[j]**
vj1	**X13**	**; to hold a copy of v[j+1]**
vcopy	**X21**	**; to hold a copy of v**
ncopy	**X22**	**; to hold a copy of n**
	X30	**; same as link register(lr)**

소스코드1: Main 함수(64 비트)

Network Computing Lab.

- **Source code**

```
1  #include <stdio.h>
2
3  extern void bubblesort(long arr[], int index);
4
5
6  int main()
7  {
8          long arr[4] = {7,3,1,2};
9          int i;
10
11         printf("Array before sorting :");
12         for(i = 0; i < sizeof(arr)/sizeof(long); i++)
13                 printf(" %ld ",arr[i]);
14         printf("\n");
15
16         bubblesort(arr, 4);
17
18         printf("Array after sorting :");
19         for(i = 0; i < sizeof(arr)/sizeof(long); i++)
20                 printf(" %ld ",arr[i]);
21         printf("\n");
22
23         return 0;
24 }
```

소스코드2 : 어셈블리 bubble sorting(방법2)

Network Computing Lab.

- Source code

```
1  .text
2  .global bubblesort
3  .global swap
4  .type bubblesort, STT_FUNC
5
6  bubblesort:
7              SUB SP, SP,#56
8              STUR X30, [SP,#32]
9              STUR X22, [SP,#24]
10             STUR X21, [SP,#16]
11             STUR X20, [SP,#8]
12             STUR X19, [SP,#0]
13
14             MOV X21, X0
15             MOV X22, X1
16             MOV X19, #0
17
18         for1tst:
19             CMP X19, X1
20             B.GE exit1
21
22             SUB X20, X19, #1
23
24         for2tst:
25             CMP X20, #0
26             B.LT exit2
27             LSL X10, X20, #3
28             ADD X11, X0, X10
29             LDUR X12, [X11,#0]
30             LDUR X13, [X11,#8]
31             CMP X12, X13
32             B.LE exit2
33
```

```
34
35             STUR X10, [SP,#56]
36             STUR X1, [SP,#48]
37             STUR X0, [SP,#40]
38
39
40             MOV X0, X21
41             MOV X1, X20
42             BL  swap
43
44             LDUR X0, [SP,#40]
45             LDUR X1, [SP,#48]
46             LDUR X10, [SP,#56]
47
48             SUB X20, X20, #1
49             B for2tst
50
51         exit2:
52             ADD X19, X19, #1
53             B for1tst
54
55         exit1:
56             LDUR X19, [SP,#0]
57             LDUR X20, [SP,#8]
58             LDUR X21, [SP,#16]
59             LDUR X22, [SP,#24]
60             LDUR X30, [SP,#32]
61             ADD SP, SP, #56
62
63             BR X30
64
65 .end
66
```

소스코드2 : 어셈블리 코드 비교(1/2)

Network Computing Lab.

- Source code(왼쪽은 64비트, 오른쪽은 32비트)

```
.text
.global bubblesort
.global swap
.type bubblesort, STT_FUNC

bubblesort:
            SUB SP, SP,#56
            STUR X30, [SP,#32]
            STUR X22, [SP,#24]
            STUR X21, [SP,#16]
            STUR X20, [SP,#8]
            STUR X19, [SP,#0]

            MOV X21, X0
            MOV X22, X1
            MOV X19, #0

      for1tst:
            CMP X19, X1
            B.GE exit1

            SUB X20, X19, #1

      for2tst:
            CMP X20, #0
            B.LT exit2
            LSL X10, X20, #3
            ADD X11, X0, X10
            LDUR X12, [X11,#0]
            LDUR X13, [X11,#8]
            CMP X12, X13
            B.LE exit2
```

```
.text
.global bubblesort
.global swap
.type bubblesort, STT_FUNC

bubblesort:
            sub     sp, sp, #20
            str     lr, [sp, #16]
            str     r7, [sp, #12]
            str     r6, [sp, #8]
            str     r3, [sp, #4]
            str     r2, [sp, #0]

            mov     r6, r0
            mov     r7, r1
            mov     r2, #0

for1tst:
            cmp     r2, r1
            bge     exit1
            sub     r3, r2, #1

for2tst:
            cmp     r3, #0
            blt     exit2
            add     r12, r0, r3, LSL #2
            ldr     r4, [r12, #0]
            ldr     r5, [r12, #4]
            cmp     r4, r5
            ble     exit2
```

소스코드2 : 어셈블리 코드 비교(2/2)

Network Computing Lab.

- Source code(왼쪽은 64비트, 오른쪽은 32비트)

```
34
35              STUR X10, [SP,#56]
36              STUR X1, [SP,#48]
37              STUR X0, [SP,#40]
38
39
40              MOV X0, X21
41              MOV X1, X20
42              BL  swap
43
44              LDUR X0, [SP,#40]
45              LDUR X1, [SP,#48]
46              LDUR X10, [SP,#56]
47
48              SUB X20, X20, #1
49              B for2tst
50
51          exit2:
52              ADD X19, X19, #1
53              B for1tst
54
55          exit1:
56              LDUR X19, [SP,#0]
57              LDUR X20, [SP,#8]
58              LDUR X21, [SP,#16]
59              LDUR X22, [SP,#24]
60              LDUR X30, [SP,#32]
61              ADD SP, SP, #56
62
63              BR X30
64
65  .end
66
```

```
32                  stmdb   sp!,{r0,r1,r2,r3,r12}
33                  mov     r0, r6
34                  mov     r1, r3
35                  bl      swap
36                  ldmia   sp!,{r0,r1,r2,r3,r12}
37
38                  sub     r3, r3, #1
39                  b       for2tst
40
41  exit2:
42                  add     r2, r2, #1
43                  b       for1tst
44
45  exit1:
46                  ldr     r2, [sp, #0]
47                  ldr     r3, [sp, #4]
48                  ldr     r6, [sp, #8]
49                  ldr     r7, [sp, #12]
50                  ldr     lr, [sp, #16]
51                  add     sp, sp, #20
52
53                  mov     pc, lr
54
55  .end
```

큰 차이점:
LDR (32비트) -> LDUR(64비트)
STR (32비트) -> STUR(64비트)

소스코드3 : 어셈블리 swap 코드(64비트)

- **Source code 분석**

```
void sort (long v[], int n)
{
  int i, j;
  for (i = 0; i < n; i ++)
    for (j = i - 1; j >= 0 ; j --)
      if (v[j] > v[j + 1]) swap(v,j);
}
```

Swap은 말그대로 배열의 두 원소의 값을 서로의 값으로 바꿉니다.

```
1  .text
2  .global swap
3  .type swap, STT_FUNC
4
5  swap:
6      LSL X10 , X1,#3
7      ADD X10 , X0,X10
8
9      LDUR X9, [X10,#0]
10     LDUR X11, [X10,#8]
11
12     STUR X11, [X10,#0]
13     STUR X9, [X10,#8]
14
15     BR X30
16
17 .end
18
19
20
```

소스코드3 : 어셈블리 코드 비교

- Source code(왼쪽은 64비트, 오른쪽은 32비트)

```
1   .text
2   .global swap
3   .type swap, STT_FUNC
4
5   swap:
6       LSL X10 , X1,#3
7       ADD X10 , X0,X10
8
9       LDUR X9, [X10,#0]
10      LDUR X11, [X10,#8]
11
12      STUR X11, [X10,#0]
13      STUR X9, [X10,#8]
14
15      BR X30
16
17  .end
18
19
20
```

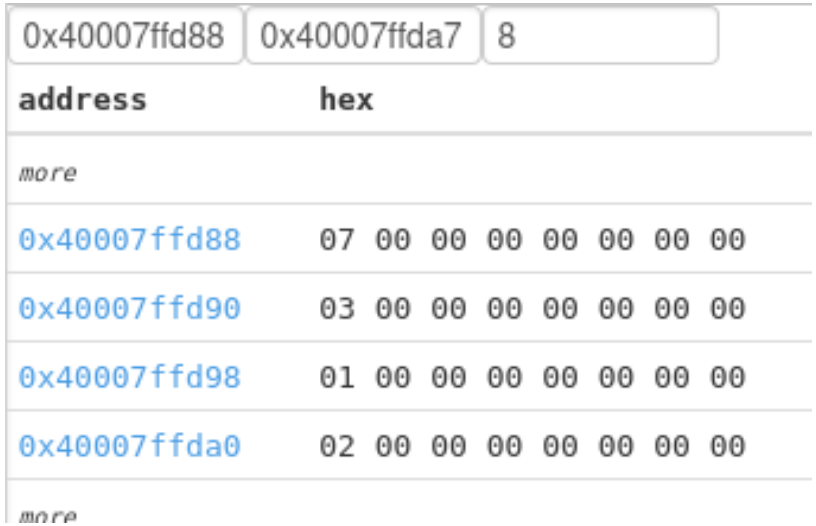
```
.text
.global swap
.type swap, STT_FUNC

swap:
        add     r12, r0, r1, lsl #2

        ldr     r2, [r12, #0]
        ldr     r3, [r12, #4]

        str     r3, [r12, #0]
        str     r2, [r12, #4]

        mov     pc, lr

.end
```

큰 차이점:
LDR (32비트) -> LDUR(64비트)
STR (32비트) -> STUR(64비트)

64비트 명령어

- ldr, ldur 둘다 존재. 차이점 ???
- Ldr r0,[r1,0] // load register r0 from the address pointed to by (r1 + (0 * size)) where size is 8 bytes for 64-bit stores, 4 bytes for 32-bit stores
- Ldur r0,[r1,0] // load register r0 from the address pointed to by (r1 + 0) – the mnemonic means " load unscaled register“
- 64 비트 명령어 리스트는 2020-2-CA-ch2-Part3 P43-47를 보세요.

sort전 array



long type 배열로 선언해서 배열 원소 하나당 8byte씩 메모리를 할당 받습니다.

sort후 array

Network Computing Lab.

0x40007ffd88 | 0x40007ffda7 | 8

address	hex
more	
0x40007ffd88	01 00 00 00 00 00 00 00
0x40007ffd90	02 00 00 00 00 00 00 00
0x40007ffd98	03 00 00 00 00 00 00 00
0x40007ffda0	07 00 00 00 00 00 00 00

배열의 주소를 메모리 윈도우에 가서 입력하거나 Qemu으로 sort후 배열에 저장된 값을을 확인할 수 있습니다.

```
Array before sorting : 7  3  1  2
Array after sorting : 1  2  3  7
```

Lab 7 : bubble sorting(방법 1)로 어셈블리 코드 작성 및 실행하기

- P19에 있는 어셈블리 코드는 bubble sorting (방법 2)으로 구현된 것이다. 이를 bubble sorting (P15에 있는 방법 1)으로 동작하도록 수정한다.
- 컴파일 후 수행한다.
- **원하는 수행결과가 나오는지 확인한다.**
- 만일 원하는 수행결과가 나오지 않으면 P15에 있는 예제에서 outer loop의 pass i에 해당된 결과가 나왔는지 단계별로 확인한다. (Lab 3-1 Outer loop 수행 끝날 때마다 배열 값의 변동 추적 방법을 이용하여 debugging 한다. 그래도 문제가 있으면 Lab 3-2 Inner loop 수행시 배열 값의 변동 추적 방법을 이용하여 debugging 한다)- lab3 참고
- **어셈블리코드 및 수행결과를 화면 캡쳐 한다.**

Network Computing Lab.

- 부록: 어셈블리프로그래밍 Lab을 통해 배울 수 있는 내용 ?

어셈블리프로그래밍을 위한 **Lab** 내용

0. 어셈블리프로그래밍 개발환경 구축 및 디버거 사용법 학습
1. Factorial 함수를 어셈블리 코드로 구현 및 **test**
2. 디버거를 이용한 **C pointer** 동작 및 **stack overflow test**
3. Bubble sorting을 위한 어셈블리 코드 분석 및 **test**
4. 메모리 불량 **test** 및 메모리내 데이터 패턴 검색 코드 구현
5. 매트릭스 곱셈을 어셈블리 코드로 구현 및 **test**
6. Quick sorting을 어셈블리 코드로 구현
7. 64 bit ARM 어셈블리 코드로 작성한 **Bubble sorting**

어셈블리프로그래밍 **Lab**을 통해 배울 수 있는 내용 ?

1. 고급언어 프로그램이 컴퓨터 하드웨어 위에서 어떻게 동작하나 ?
   - 고급언어 프로그램상에서는 변수, 자료구조체, 내제화된 알고리즘에 의한 수행과정이 들어 있슴
   - 어셈블리 프로그램상에서는 변수, 자료구조체는 메모리 주소, 레지스터 주소로 대체되고 내제화된 알고리즘에 의한 수행과정이 메모리 또는 레지스터 참조 동작 + CPU 연산 동작으로 대체
2. 프로그램 성능을 좌우하는 요소들은 무엇인가 ?
   - 명령어의 빠른 수행을 위해 사용가능한 레지스터들을 최대한 활용 (compiler의 역할) – 고급언어 프로그램내 변수를 위해 메모리 또는 레지스터가 할당됨. 자주 사용되는 변수는 레지스터를 할당
   - 명령어 코드수를 최소화 (compiler의 역할) – 다중 loop에서 맨 안쪽의 inner loop 내 코드가 한줄 줄었다고 가정합시다. 이게 수행속도를 높인다.
   - 메모리 참조 횟수를 최소화 (사용하는 자료구조체, 알고리즘에 의해 결정됨)

어셈블리 프로그래밍 – 레지스터 활용

- 성능이 높은 어셈블리 프로그램을 위해서는 매우 중요
- 명령어에서 **operand**는 메모리 또는 레지스터에 있는 데이터를 사용
- 레지스터에 있는 값들을 **ALU** 이용하여 연산하는 동작은 **CPU 1** 클럭 소모.
- 메모리에 있는 값들 연산의 경우는 **1)** 일단 레지스터로 **copy** 해오는 데 **CPU 1** 클럭 소모**, 2)** 레지스터 값들을 **ALU** 이용하여 연산하는 데 **CPU 1** 클럭 소모**.** 총 **CPU 2**개 클럭소모
- 자주 사용하는 변수는 메모리 대신에 레지스터에 위치하게 함**.** 소모하는 **CPU** 클럭 개수를 줄이는 효과
- 똑 같은 일을 하는 프로그램 **A**와 **B** 의 성능 비교
  - 총 기계어 코드 갯수 **–** 갯수가 작을 수록 적은 숫자의 **CPU** 클럭 소모
  - 수행도중 메모리 참조**(memory read** 또는 **memory write)** 총 횟수 **–** 횟수가 작을 수록 적은 숫자의 **CPU** 클럭 소모
  - **compiler** 에서 코드 최적화한다는 의미는 생성된 코드 총 갯수를 줄이면서 메모리참조 총 횟수를 줄이는 것을 의미함
- 어셈블리 프로그래밍한다는 것은 **complier**가 하는 것과 동일한 작업임

C 코드의 수행과정 ?

- **C 프로그램 z= x + y 코드의 하드웨어 위에서 수행과정**
  - 기계어 코드로 변환 **(by compiler)**
  - 기계어 코드를 수행 **(by CPU)**
  - (예) C 코드에서 int x=1, y=2, z; 선언후 z=x + y;
  - (예) ARM 어셈블리(명령어) 코드 예제
    - LDR r0, [r3] ; x 갖고 오기 (r3가 x 저장된 메모리 번지 갖는다고 가정)
    - LDR r1, [r3, #4] ; y 갖고 오기
    - ADD r2, r0, r1
    - STR r2, [r3, #8] ; z에 저장하기
  - ARM 어셈블리 코드에서 각 명령어 수행과정은 ?

명령어 수행과정 ?

1. 명령어 갖고 오기 (**fetch**): (예) 메모리 **1000** 번지에 저장된 명령어를 **CPU**로 읽어온다.
2. 명령어 해독하기(**decode**): **CPU** 내 **control unit**(제어유닛)이 읽어온 명령어를 해독한다. (예) 만일 **"ADD R3, R1, R2"** 이라면 **R1**값과 **R2**값을 더하여 결과를 **R3** 저장하는 덧셈 명령어임을 알게 된다.
3. 명령어 수행하기(**execute**): **ALU**를 이용하여 연산 또는 논리동작을 수행한다. (예) **R1** 값과 **R2** 값을 꺼내어 **ALU** 입력단으로 보내어 덧셈을 수행한다.
4. 수행결과 저장하기(**store**): 수행결과를 레지스터 또는 메모리에 저장한다. (예) 앞에서 수행한 덧셈 결과를 **R3**에 저장한다.

각 단계는 **CPU** 클럭 **1**개씩을 소모